ПРОГРАММИРОВАНИЕ
введение в профессию

А.В.СТОЛЯРОВ

3 ПАРАДИГМЫ

Любое использование данного файла означает ваше согласие с условиями лицензии (см. ниже). Электронные версии этой и других книг автора бесплатно доступны на сайте http://www.stolyarov.info

© Андрей Викторович Столяров, 2026

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Программирование: введение в профессию» в трёх томах, ранее публиковавшееся в издательстве МАКС Пресс в 2016–2021 гг., а также в издательстве ДМК Пресс в 2021 г., называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1) воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2) копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искажённых экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей (включая файлообменные и любые другие сервисы, организованные в Сети Интернет коммерческими компаниями, в том числе бесплатные), а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные децентрализованные файлообменные P2P-сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. СТОЛЯРОВ

ПРОГРАММИРОВАНИЕ
ВВЕДЕНИЕ В ПРОФЕССИЮ

издание 2,92 **(электронное)**

в трёх томах

Том III: ПАРАДИГМЫ

Белград — 2026

УДК 519.683+004.2+004.45
ББК 32.97
С81

Столяров, Андрей Викторович.
С81 Программирование: введение в профессию. – Изд. 2,92 (электронное), с испр. : в 3 томах / А. В. Столяров. – Белград : 2026.

Том III : Парадигмы. – 233 с. : ил.

Учебник «Программирование: введение в профессию» ориентирован на самостоятельное изучение и предполагает использование систем семейства Unix (в т. ч. Linux) в роли сквозной среды для обучения.

Третий том учебника посвящён высокоуровневому программированию и возможным подходам к осмыслению программы и её исполнения, известным как парадигмы программирования. Рассматриваются языки Си++, Лисп, Scheme, Пролог, Хоуп и Tcl; дополнительно показано создание графических пользовательских интерфейсов с помощью библиотеки FLTK для Си++ и фреймворка Tcl/Tk.

Для школьников, студентов, преподавателей и всех, кто интересуется программированием.

УДК 519.683+004.2+004.45
ББК 32.97

© Столяров А. В., 2020
© Столяров А. В., 2026,
с изменениями

Оглавление

9. Парадигмы в мышлении программиста **7**
9.1. Загадочное слово «парадигма» 7
9.2. Рекурсия как пример парадигмы 17
9.3. Парадигмы и языки программирования 24
9.4. Примеры частных парадигм 31

10. Язык Си++, ООП и АТД **34**
10.1. От Си к Си++ 34
10.2. О выборе подмножества Си++ 35
10.3. Методы, объекты и защита 35
10.4. Абстрактные типы данных в Си++ 39
10.5. Обработка исключительных ситуаций 59
10.6. Наследование и полиморфизм 64
10.7. Шаблоны 74
10.8. Снова о парадигмах 83
10.9. Особенности оформления кода на Си++ 84
10.10. Пример: TCP-сервер 86
10.11. О графических интерфейсах пользователя 91
Си++: что дальше 106

11. Неразрушающие парадигмы **108**
11.1. Язык Лисп и его S-выражения 108
11.2. Scheme: Лисп, но не совсем 133
11.3. Об оформлении кода на лиспоподобных языках 140
11.4. Логическое и декларативное программирование 141
11.5. Ленивые вычисления 163

12. Компиляция, интерпретация, скриптинг **178**
12.1. Характерные особенности скриптовых языков 178
12.2. Язык Tcl 179
12.3. Интерпретатор Tcl и язык Си 190
12.4. Графические интерфейсы на Tcl/Tk 193
12.5. Стратегии выполнения как парадигмы 203
Приложение 5. Сравнение файлов и наложение изменений 215
Приложение 6. Системы контроля версий 217

Заключение 226

Список литературы 227
Предметный указатель 229

Содержание

9. Парадигмы в мышлении программиста **7**
9.1. Загадочное слово «парадигма» . . . 7
9.1.1. Терминологические трудности . . . 8
9.1.2. Парадигмы вне программирования . . . 8
9.1.3. Обзор основных стилей . . . 10
9.2. Рекурсия как пример парадигмы . . . 17
9.2.1. Пример с обходом дерева . . . 17
9.2.2. Виды рекурсии . . . 18
9.2.3. Редукция (свёртка) последовательностей . . . 19
9.2.4. Остаточная рекурсия . . . 21
9.2.5. Рекурсивное мышление . . . 23
9.3. Парадигмы и языки программирования . . . 24
9.3.1. О роли языка программирования . . . 24
9.3.2. Концептуальное отличие Си от Паскаля . . . 26
9.3.3. Два подхода к сравнению языков . . . 30
9.4. Примеры частных парадигм . . . 31
9.4.1. Callback-функции . . . 31
9.4.2. Программирование в терминах явных состояний . . . 31
9.4.3. Метапрограммирование . . . 33

10. Язык Си++, ООП и АТД **34**
10.1. От Си к Си++ . . . 34
10.2. О выборе подмножества Си++ . . . 35
10.3. Методы, объекты и защита . . . 35
10.3.1. Функции-члены (методы) . . . 35
10.3.2. Неявный указатель на объект . . . 36
10.3.3. Защита. Понятие конструктора . . . 36
10.3.4. Зачем нужна защита . . . 37
10.3.5. Классы . . . 38
10.3.6. Деструкторы . . . 39
10.4. Абстрактные типы данных в Си++ . . . 39
10.4.1. Перегрузка имён функций; декорирование . . . 39
10.4.2. Переопределение символов стандартных операций . . . 41
10.4.3. Конструктор умолчания. Массивы объектов . . . 42
10.4.4. Конструкторы преобразования . . . 42
10.4.5. Ссылки . . . 43
10.4.6. Ссылки на константы . . . 44
10.4.7. Ссылки как семантический феномен . . . 44
10.4.8. Константные методы . . . 45
10.4.9. Операции работы с динамической памятью . . . 46
10.4.10. Конструктор копирования . . . 46
10.4.11. Временные и анонимные объекты . . . 47
10.4.12. Значения параметров по умолчанию . . . 48
10.4.13. Описание метода вне класса. Области видимости . . . 48
10.4.14. «Подставляемые» функции (inline) . . . 49
10.4.15. Инициализация членов класса в конструкторе . . . 50
10.4.16. Перегрузка операций простыми функциями . . . 50
10.4.17. Дружественные функции и классы . . . 51
10.4.18. Переопределение операций присваивания . . . 52
10.4.19. Методы, возникающие неявно . . . 52
10.4.20. Переопределение операции индексирования . . . 53
10.4.21. Переопределение операций `++` и `--` . . . 54
10.4.22. Переопределение операции `->` . . . 55
10.4.23. Переопределение операции вызова функции . . . 56
10.4.24. Переопределение операции преобразования типа . . . 56
10.4.25. Пример: разреженный массив . . . 56
10.4.26. Статические поля и методы . . . 58
10.5. Обработка исключительных ситуаций . . . 59
10.5.1. Ошибочные ситуации и проблемы их обработки . . . 60
10.5.2. Общая идея механизма исключений . . . 61
10.5.3. Возбуждение исключений . . . 61
10.5.4. Обработка исключений . . . 62
10.5.5. Обработчики с многоточием . . . 63
10.5.6. Объект класса в роли исключения . . . 63
10.5.7. Автоматическая очистка . . . 64
10.5.8. Преобразования типов исключений . . . 64
10.6. Наследование и полиморфизм . . . 64
10.6.1. Иерархические предметные области . . . 64
10.6.2. Наследование структур и полиморфизм адресов . . . 65
10.6.3. Методы и защита при наследовании . . . 66
10.6.4. Конструирование и деструкция наследника . . . 66
10.6.5. Виртуальные функции . . . 67
10.6.6. Чисто виртуальные методы и абстрактные классы . . . 69
10.6.7. Виртуальность в конструкторах и деструкторах . . . 70
10.6.8. Наследование ради конструктора . . . 70
10.6.9. Виртуальный деструктор . . . 71
10.6.10. Ещё о полиморфизме . . . 71
10.6.11. Приватные и защищённые деструкторы . . . 72
10.6.12. Перегрузка функций и сокрытие имён . . . 72
10.6.13. Вызов в обход механизма виртуальности . . . 72
10.6.14. Операции приведения типа . . . 73
10.6.15. Иерархии исключений . . . 74
10.7. Шаблоны . . . 74
10.7.1. Шаблоны функций . . . 75
10.7.2. Шаблоны классов . . . 76
10.7.3. Специализация шаблонов . . . 78
10.7.4. Пример: свёртка последовательностей . . . 79
10.7.5. Константы в роли параметров шаблона . . . 81
10.8. Снова о парадигмах . . . 83
10.8.1. Спектр парадигм в Си++ . . . 83
10.8.2. ООП и АТД . . . 83
10.8.3. Полиморфизм без мистики . . . 83
10.8.4. Наследование как сужение множества . . . 84
10.9. Особенности оформления кода на Си++ . . . 84
10.9.1. Соглашения об именах . . . 84
10.9.2. Форматирование заголовков классов . . . 85
10.9.3. Форматирование заголовка конструктора . . . 85
10.9.4. Тела функций в заголовке класса . . . 86
10.10. Пример: TCP-сервер . . . 86
10.11. О графических интерфейсах пользователя . . . 91
10.11.1. Знакомимся с библиотекой FLTK . . . 91
10.11.2. Простые кнопки и реакция на них . . . 93
10.11.3. Другие виды кнопок . . . 95
10.11.4. Виджеты для ввода текста . . . 96
10.11.5. Обрамление и метки . . . 98
10.11.6. Виджеты для вывода . . . 100
10.11.7. Окна, допускающие изменение размера . . . 101
10.11.8. Обработка аргументов командной строки . . . 102
10.11.9. Обзор нерассмотренных возможностей . . . 104
10.11.10. FLTK и парадигма ООП . . . 104
10.11.11. (*) Если нужен свой главный цикл . . . 106
Си++: что дальше . . . 106

11. Неразрушающие парадигмы 108
11.1. Язык Лисп и его S-выражения . . . 108
11.1.1. Немного истории . . . 108
11.1.2. SBCL, GCL и ECL . . . 109
11.1.3. S-выражения: гетерогенная модель данных . . . 112
11.1.4. Вычисление S-выражений . . . 113
11.1.5. Пользовательские функции . . . 115
11.1.6. Разрушающие функции . . . 116
11.1.7. Функция `eval` . . . 118
11.1.8. Функции как объекты обработки . . . 119
11.1.9. Фунарг-проблема и лексическое связывание . . . 120
11.1.10. О фунарг-проблеме в других языках . . . 123
11.1.11. Редукция списков . . . 125
11.1.12. Груз устаревших парадигм . . . 126
11.1.13. Возможности, которых лучше бы не было . . . 127
11.1.14. Ввод-вывод . . . 128
11.1.15. Неутешительное заключение . . . 132
11.2. Scheme: Лисп, но не совсем . . . 133
11.2.1. Chicken Scheme . . . 133
11.2.2. Видимые отличия Scheme от обычного Лиспа . . . 134
11.2.3. Ввод-вывод в Scheme . . . 135
11.2.4. Континуации (продолжения) . . . 137
11.2.5. (*) Continuation-passing style . . . 140
11.3. Об оформлении кода на лиспоподобных языках . . . 140
11.4. Логическое и декларативное программирование . . . 141
11.4.1. Язык Пролог и его история . . . 142
11.4.2. SWI-Prolog . . . 142
11.4.3. Модель данных в Прологе . . . 144
11.4.4. Операция унификации термов . . . 147
11.4.5. Отношения, предикаты и факты . . . 147
11.4.6. Правила и процедуры . . . 149
11.4.7. Обращения к процедурам, прототипы и инверсия . . . 150
11.4.8. Отрицание и отсечение: убийцы логики . . . 153
11.4.9. Арифметика . . . 155
11.4.10. Ввод-вывод . . . 157
11.4.11. Анализ атомов и термов . . . 159
11.4.12. Списки решений . . . 160
11.4.13. Работа с базой данных . . . 161
11.4.14. Оформление кода на Прологе . . . 162
11.5. Ленивые вычисления . . . 163
11.5.1. Две возможные стратегии вычисления выражений . . . 163
11.5.2. Хоуп и другие «ленивые» языки . . . 163
11.5.3. Интерпретатор Hopeless . . . 164
11.5.4. Лексика и синтаксис языка Хоуп . . . 165
11.5.5. Модель данных и система типов . . . 165
11.5.6. Функции в Хоупе . . . 166
11.5.7. Операции `if` и `let` . . . 167
11.5.8. Полиморфные функции . . . 168
11.5.9. Безымянные функции и функционалы . . . 169
11.5.10. Операция `letrec` . . . 170
11.5.11. Редукция списков . . . 171
11.5.12. Бесконечные структуры данных. Примеры . . . 172
11.5.13. Ввод-вывод в Хоупе . . . 173
11.5.14. Карринг . . . 174
11.5.15. Комбинатор неподвижной точки . . . 175

12. Компиляция, интерпретация, скриптинг 178
12.1. Характерные особенности скриптовых языков . . . 178
12.2. Язык Tcl . . . 179
12.2.1. Интерпретатор и простейшие программы . . . 179
12.2.2. Переменные и их значения . . . 180
12.2.3. Ветвления, циклы... и строки . . . 181
12.2.4. Процедуры и видимость переменных . . . 184
12.2.5. Обработка особых ситуаций . . . 186
12.2.6. Файлы, потоки и внешние команды . . . 187
12.2.7. Ассоциативные массивы . . . 189
12.2.8. Если сравнить Tcl с Лиспом . . . 189
12.3. Интерпретатор Tcl и язык Си . . . 190
12.3.1. Встраиваемый Tcl . . . 190
12.3.2. Расширение набора команд tclsh . . . 193
12.4. Графические интерфейсы на Tcl/Tk . . . 193
12.4.1. Библиотека Tk и интерпретатор `wish` . . . 193
12.4.2. Взаимодействие с оконным менеджером . . . 195
12.4.3. Основные виджеты . . . 196
12.4.4. Положение и размеры . . . 198
12.4.5. Управление шрифтами . . . 200
12.4.6. Обработка событий . . . 201
12.4.7. «Ресурсы» и русификация . . . 202
12.5. Стратегии выполнения как парадигмы . . . 203
12.5.1. И всё же — что такое скрипт . . . 203
12.5.2. Дихотомия Оустерхаута и её противники . . . 204
12.5.3. И всё же что такое «интерпретация»? . . . 206
12.5.4. Интерпретация как парадигма . . . 207
12.5.5. Зависимости и самодостаточность . . . 209
12.5.6. Миф о переносимости . . . 211
12.5.7. Когда интерпретация всё же допустима . . . 211
12.5.8. (*) Размышления о чистой компиляции . . . 213

Приложение 5. Сравнение файлов и наложение изменений . . . 215
Приложение 6. Системы контроля версий . . . 217
Система CVS . . . 218
Система `git` . . . 221

Заключение . . . 226
Список литературы . . . 227
Предметный указатель . . . 229

Список донэйторов, поддержавших проект создания этой книги в ходе подготовки двух первых изданий (кроме пожелавших сохранить анонимность):

Gremlin, Grigoriy Kraynov, Шер Арсений Владимирович, Таранов Василий, Сергей Сетченков, Валерия Шакирзянова, Катерина Галкина, Илья Лобанов, Сюзана Тевдорадзе, Иванова Оксана, Куликова Юлия, Соколов Кирилл Владимирович, jeckep, Кулёва Анна Сергеевна, Ермакова Марина Александровна, Переведенцев Максим Олегович, Костарев Иван Сергеевич, Донцов Евгений, Олег Французов, Степан Холопкин, Попов Артём Сергеевич, Александр Быков, Белобородов И. Б., Ким Максим, artyrian, Игорь Эльман, Илюшкин Никита, Кальсин Сергей Александрович, Евгений Земцов, Шрамов Георгий, Владимир Лазарев, eupharina, Николай Королев, Горошевский Алексей Валерьевич, Леменков Д. Д., Forester, say42, Аня «canja» Ф., Сергей, big_fellow, Волканов Дмитрий Юрьевич, Танечка, Татьяна 'Vikora' Алпатова, Беляев Андрей, Лошкины (Александр и Дарья), Кирилл Алексеев, kopish32, Екатерина Глазкова, Олег «burunduk3» Давыдов, Дмитрий Кронберг, yobibyte, Михаил Аграновский, Александр Шепелёв, G.Nerc=Y.uR, Василий Артемьев, Смирнов Денис, Pavel Korzhenko, Руслан Степаненко, Терешко Григорий Юрьевич 15e65d3d, Lothlorien, vasiliandets, Максим Филиппов, Глеб Семёнов, Павел, unDEFER, kilolife, Арбичев, Рябинин Сергей Анатольевич, Nikolay Ksenev, Кучин Вадим, Мария Трофимова, igneus, Александр Чернов, Roman Kurynin, Власов Андрей, Дергачёв Борис Николаевич, Алексей Алексеевич, Георгий Мошкин, Владимир Руцкий, Федулов Роман Сергеевич, Шадрин Денис, Панфёров Антон Александрович, os80, Зубков Иван, Архипенко Константин Владимирович, Асирян Александр, Дмитрий С. Гуськов, Тойгильдин Владислав, Masutacu, D.A.X., Каганов Владислав, Анастасия Назарова, Гена Иван Евгеньевич, Линара Адылова, Александр, izin, Николай Подонин, Юлия Корухова, Кузьменкова Евгения Анатольевна, Сергей «GDM» Иванов, Андрей Шестимеров, vap, Грацианова Татьяна Юрьевна, Меньшов Юрий Николаевич, nvasil, В. Красных, Огрызков Станислав Анатольевич, Бузов Денис Николаевич, capgelka, Волкович Максим Сергеевич, Владимир Ермоленко, Горячая Илона Владимировна, Полякова Ирина Николаевна, Антон Хван, Иван К., Сальников Алексей, Щеславский Алексей Владимирович, Золотарев Роман Евгеньевич, Константин Глазков, Сергей Черевков, Андрей Литвинов, Шубин М. В., Сыщенко Алексей, Николай Курто, Ковригин Дмитрий Анатольевич, Андрей Кабанец, Юрий Скурский, Дмитрий Беляев, Баранов Виталий, Новиков Сергей Михайлович, maxon86, mishamm, Спиридонов Сергей Вячеславович, Сергей Черевков, Кирилл Филатов, Чаплыгин Андрей, Виктор Николаевич Остроухов, Николай Богданов, Баев Ален, Плосков Александр, Сергей Матвеев a.k.a. stargrave, Илья, aykar, Олег Бартунов, micky_madfree, Алексей Курочкин aka kaa37, Николай Смолин, I, JDZab, Кравчик Роман, Дмитрий Мачнев, bergentroll, Иван А. Фролов, Александр Чащин, Муслимов Я. В., Sedar, Максим Садовников, Яковлев С. Д., Рустам Кадыров, Набиев Марат, Покровский Дмитрий Евгеньевич, Заворин Александр, Павлочев Сергей Юрьевич, Рустам Юсупов, Noko Anna, Андрей Воронов, Лисица Владимир, Алексей Ковура, Чайка Леонид Николаевич, Коробань Дмитрий, Алексей Вересов, suhorez, Ольга Сергеевна Цаун, Боборыкин Сергей Владимирович, Олохтонов Владимир, Александр Смирницкий, Максим Клочков, Анисимов Сергей, Вадим Вадимович Чемодуров, rumiantcev, babera, Артём Коротченко, Евгений Шевкунов, Александр Смирницкий, Артём Шутов, Засеев Заурбек, Konstantin Slobodnyuk, Yan Zaripov, Виталий Бодренков, Александр Сергиенко, Денис Кузаков, Пушистый Шмель, Сергей Спивак, suuuumcaa, Гагарин, Валерий Гайнуллин, Александр Махаев (mankms), VD, А. Б. Лихачёв, Col_Kurtz, Espanka, Дмитрий Сергеевич Х., Анатолий Камчатнов, Табакаев Евгений Владимирович, Александр Трошенков, Малюга Андрей, Сорокин Андрей, Буркин Иван, Александр Логунов, moya_bbf3, Vilnur_Sh, Кипнис Александр, Oleg G. Geier, Владимир Исаев (fBSD), Филимонов Сергей Владимирович, vsudakou, AniMath, Данилов Евгений, Воробьёв В. С., mochalov, Камчатская LUG, Логинов Сергей Владимирович, Чистяков Артём Андреевич, A&A Сулимовы, Денисов Денис Юрьевич, Сутупов Андрей Михайлович, kuddai, Озерицкий Алексей Владимирович, alexz, Владимир Цой, Владимир Бердовщиков, Сергей Дмитриченко, Иванцов Данил Игоревич, Замыслов Д. А., Владимир Халямин, Максим Карасев (begs), ErraticLunatic, А. Е. Артемьев, FriendlyElk, Алексей Спасюк, Андриевский Константин Андреевич, Сукачёв Владислав Вячеславович, Артём Абрамов, maxon86, Соколов Павел Андреевич, Алексей Н, Никита Гуляев, Бодюль Евгений, rebus_x, Рачинский Максим Юрьевич, Мезенцев Андрей Владимирович, Ижмяков Павел Андреевич, Денис К., Умнова Наталья Евгеньевна, Артём Жилеев, Легоцкий Николай Евгеньевич, Виталий Саган, Александр Д., Иванов Павел Николаевич, Роман Викторович Щербинин, Константин Хомутов, barsik, Чистяков Сергей, Дмитрий Нурисламов, Евгений lipklim Климов, Азаров Владимир, Иванов Виталий Сергеевич, Бичекуев Аслан Расулович, Владимир Бахтин, Иванцов Данил Игоревич, Смирнов Иван, lexizz19, Содномов Алексей Эрдэни-Баирович, yshawn, Авдеенков Василий, Андрей, Игорь, Денис Николаевич Ж., Aglaro, aoratos, Шарапов Кирилл Владимирович, Александр, Нестеренко Владислав Валерьевич, Рябчиков Дмитрий Александрович, Андрей Александрович Герман, pashak, Морозов Михаил Павлович, Тихонов Руслан Владимирович, Копкин Максим, ЛАГ-5891, Евгений Феликсович Набоков, Плотников Сергей, Худяев Глеб, Максим Шаталин, Пётр Захаров, Михаил, Тимофей Дубский, Андрей Анатольевич Бородин, gunpi, Зеленин Андрей, Плаксицкая Наталия, enoelia, Юрий Сидоров, Dizzy, Дмитрий Серегин, Михаил К., Павел Егоров, Даниил Шелепанов, Дьяконов Олег, Данилов Роман Викторович, le34, Константин Плахетка, Илья Денисов, Вадим Марченко, Ратников Сергей Валерьевич, Илья Алексеевич Белоусов, Кузьмин Ринат, savtog, Валеев Эдуард Накипович, paq1311, e_matveev, Ефремов Артур, братишка, Иван Михайлович Пузырёв, Jamezzz, Мелкумян Завен, Игорь Романович Шульга, Сергей Спивак, Василий Дзюбенко, sadsnake, Игорь Середов, Pavel Heiden, Богдан Артемьев, A6e3iana, В. М. Сотников, Ярослав Подымский, Андрей Барышев, Лиана Саргсян, Юрий Городничев, Дрожак Александр

Список донэйторов, поддержавших новое (электронное) издание (актуален на 30 апреля 2026 г.):

Даниил Киреев, Александр Шмигидер, posyx, okuznetc, Владислав Беин, Епишин Василий Ильич, user12, Eugene_Palkowski, Шиховцев Евгений, Василий Дзюбенко, Александр Хахмович, vostelmakh, Степан Клюкин, serereg, Пестриков Михаил, Владислав Юсов, Шестаков Егор, Анастасия Назарова, Nick Anoshin, Alex Kotovsky, Александр Морев, Валерий Акуленко, Александр Мукашев, Мария Теплинская, Максим Каланчекаев, Глеб Худяев, Hleb, Павел Зенов, Андрей Столяр, Idel A., Халитов Эльдар, Егор Гавриленко, Maxim Klochkov, Павел Владимирович Савкин (panchous), Арсений Шер, Оськин Александр Николаевич, Якушонок Сергей, Александр Колесниченко, Ansar, Daniil Lemenkov, Morf, Алексей Исмагилов, Дмитрий Макаревич, Имиль, Александр Мокин, Фатеев Дмитрий, Евгений (dovvakkin) К., Fouer, Artem Klimov, Баранов Виталий, Пе-тян, nevill, Андрей, greatEcho, Roman Andreeff, namtyda, А. В. Игнашев, nixan, Дмитрий Червов, Николай Костромитин, Вера Майер, timar07, Пузырёв Иван, Данил Кичай, memeexpert, Дмитрий Кронберг, andry0980, Timur Demyanov, adrianopol, Владимир К., Анатолий, Нецветаев Владимир, Максим Лавров, Ярослав Шальнов, Yury K., Маргарита Каранда, Максим Переведенцев, Дмитрий Володин, sib_coder, Андрей Т., Татьяна Белова, Аристов М.С., pupiz, andliar, Михаил, Юрий Широков, kezhich, Евгений Р., Додевски А.С, Костя, voothi, Евгений, Vasiliy Taranov, asterite, Пузырёв Иван Михайлович, Роман Филиппов, Федотов Максим Андреевич, Иван, Мита Таики, Белобородов И. Б., Миронов Владимир Вячеславович, Артемий Коваленко

Ещё раз спасибо всем тем, кто принял участие в финансировании проекта. Без донэйторов этой книги не было бы вообще ни в каком виде.

Часть 9

Парадигмы в мышлении программиста

9.1. Загадочное слово «парадигма»

С помощью первых двух томов нашей книги читатель, можно надеяться, освоил азы программирования на примере Паскаля, создание программ на уровне машинных команд (на языке ассемблера), узнал язык Си и, наконец, научился взаимодействовать с операционной системой на уровне системных вызовов. Настало время признать, что до сей поры мы рассматривали программирование лишь с одной стороны — «снизу».

Исторически «языками высокого уровня» назывались любые языки программирования, отличные от языка ассемблера, то есть не требующие работать на уровне машинных команд. Те времена давно прошли. Язык Си, появившийся, как мы знаем, в качестве альтернативы языкам ассемблера, позволяет контролировать компьютер практически столь же жёстко, как и явное задание последовательностей машинных команд, а для совсем специфических случаев вроде точек соприкосновения между ядром ОС и пользовательскими программами Си допускает компоновку программы с модулями, написанными «на ассемблере», и ассемблерные вставки непосредственно в текст на самом Си. Вне всякого сомнения язык Си по своей сущности намного ближе к ассемблерам, нежели ко всевозможным языкам, скрывающим от программиста устройство компьютера, таким как Хаскель или даже Джава, C# и Go; программирование на чистом Си следует считать низкоуровневым, как и работу на языках ассемблера — Си лишь снижает трудоёмкость, но не возводит никаких препятствий между программистом и машиной, так что он, как мы обсуждали, не просто годится для создания ядер операционных систем и прошивок микроконтроллеров (так называемого «программирования на голом железе»), но и остаётся в этой области единственной альтернативой ассемблерному программированию. Если же говорить о создании программ, предназначенных для работы под управлением ОС (а мы занимались и собираемся дальше заниматься только этим, написание новой операционки в наши планы не входит) — то здесь «в самом низу» оказывается уровень системных вызовов ядра, невозможно опуститься «ещё ниже». Следовательно, всё, чем мы занимались, осваивая во втором томе материал частей с пятой по седьмую, тоже относится к низкоуровневому программированию, ну а в восьмой части, посвящённой внутреннему устройству ядра, мы не программировали, но изучали при этом, как работает ядро — программа, ниже которой только сам компьютер. Получается, что по меньшей мере с третьей части нашей книги и до конца второго тома мы даже не пытались рассматривать программирование в другой его ипостаси — высокоуровневое.

Чуть сложнее ситуация с языком Паскаль, на примере которого мы проиллюстрировали базовые принципы создания программ; Паскаль всегда считался языком высокого уровня, но и при работе на этом языке мы точно знаем, например, что переменная есть не что иное, как область памяти, нам доступна прямая работа с указателями, что тоже подразумевает наличие памяти в фоннеймановском смысле, да и такая (ныне уже совсем привычная читателю) вещь, как присваивание, если приглядеться, обусловлена использованием машины фон Неймана как базового вычислителя.

Низкоуровневое программирование подразумевает, что в процессе создания программы мы не только помним, как устроен компьютер, но и активно используем это знание, чтобы сделать программу эффективнее или расширить её возможности. Если говорить о «системном программировании», то обычно под таковым подразумевают создание программ, которые будут обслуживать другие программы. Иначе говоря, системные программы напрямую не предназначены для решения проблем, имеющихся у конечного пользователя, то есть таких проблем, которые сами по себе не связаны с компьютером; предназначение системных программ скорее в том, чтобы упростить жизнь программистам и прочим компьютерным профессионалам — с их помощью *обслуживают другие программы и сами компьютеры*.

Здесь следует отметить два момента. Во-первых, программы, обслуживающие компьютер, в большинстве случаев должны быть написаны с учётом того, как этот компьютер устроен. Во-вторых, от того, насколько эффективно написана системная программа, может зависеть качество работы многих прикладных программ, а иногда и вообще *всех* — как в случае с ядром операционной системы, которое, конечно, тоже представляет собой системную программу; ну а добиться высокой эффективности, не учитывая устройство компьютера, практически невозможно. Поэтому чаще всего системное программирование подразумевает, что работа будет проходить на низком уровне.

В области ***прикладного программирования*** всё не так жёстко. При создании прикладных программ зачастую эффективность программы — скорость её работы и/или количество занимаемой ею памяти — оказывается фактором не столь важным, как трудозатраты на создание программы или, скажем, время, которое проходит от начала разработки до появления готового инструмента. Больше того, в ряде случаев эффективность по времени выполнения вообще никого не волнует: например, программа, ведущая диалог с пользователем, обычно не может «перетормозить» пользователя, даже если она написана как попало (впрочем, это совершенно не повод писать программы как попало).

Так или иначе, в прикладном программировании требования к эффективности программ в среднем не столь жёсткие, как в программировании системном, так что программист может позволить себе отвлечься от мыслей об устройстве компьютера, чтобы сосредоточиться на том, как проще, короче или понятнее воплотить в виде программы своё представление, что́ эта программа должна делать. Мышление вырывается на свободу из технологической клетки, построенной из особенностей машины фон Неймана, и вскоре после этого мы с удивлением обнаруживаем, что о программе можно *думать по-другому*.

Весь третий (и последний) том нашей книги будет посвящён так называемым ***парадигмам программирования***, начиная с гиперпопулярного нынче ***объектно-ориентированного программирования*** и заканчивая такими экзотическими моделями, в которых, например, привычные нам переменные не имеют ничего общего с областями памяти, а то и вовсе куда-то исчезают; мы увидим, что программировать можно без присваиваний, без циклов, а сама программа совершенно не обязана быть последовательностью приказов, к чему мы уже успели привыкнуть.

Было бы ошибкой заявить, что многообразие парадигм программирования может существовать лишь в прикладном программировании. Ранее мы активно использовали рекурсию, которая представляет собой хрестоматийный пример частной парадигмы. Изучая компьютерные сети, мы при создании TCP-сервера столкнулись с ***событийно-управляемым программированием*** и программированием в терминах явных состояний; это тоже парадигмы, но более общие, способные охватить всю программу, а не какой-то её отдельный фрагмент. Отдельную часть нашей книги мы посвятили ***параллельному программированию***; хотя мы и пытались убедить читателя отнестись к соответствующим инструментам с осторожностью и вообще по возможности не применять их, приходится признать, что, например, в ядре операционной системы параллельное программирование совершенно неизбежно — и, кстати, именно по этой причине мы рассматри-

вали его столь подробно, прежде чем перейти к описанию внутреннего устройства ядра. Между тем параллельное программирование — это тоже, несомненно, парадигма, ведь оно требует совершенно особого стиля мышления.

Как мы увидим чуть позже, даже такие похожие друг на друга языки программирования, как Паскаль и Си, на самом деле во многом резко отличаются друг от друга своим влиянием на мышление программиста. Тем не менее в системном программировании выбор парадигм всё же ограничен требованиями, предъявляемыми к программам. Прикладные задачи в этом плане намного мягче, а применение экзотических парадигм и языков программирования в некоторых случаях способно довольно заметно снизить трудозатраты.

Кроме того, изучение альтернативных стилей мышления способствует развитию мозга в нужном направлении и повышает общую гибкость интеллектуального аппарата, что ценно само по себе, даже если очередная освоенная парадигма не будет применяться на практике. Так или иначе, умение осмысливать одну и ту же проблему разными способами оказывается очень полезно и повышает общую ценность специалиста едва ли не в любой области, а в программировании, пожалуй, особенно.

9.1.1. Терминологические трудности

С термином «парадигма программирования» мы уже встречались. Впервые это словосочетание появилось ещё во вводной части (см. т. 1, §1.3.7); там же были упомянуты ***функциональное программирование***, ***декларативная семантика***, ***объектно-ориентированное программирование*** и, наконец, ***императивное программирование***, которым мы, собственно говоря, до сей поры и занимались.

Уже во втором томе, обсуждая в §5.1.5 философию Unix, мы упомянули «универсальную парадигму “всё есть поток байтов”»; впрочем, и Unix-way как целое, и отдельные его составляющие тоже вполне можно рассматривать как парадигмы, хотя и не всегда именно как парадигмы *программирования* — так, принцип «каждая программа должна решать одну задачу, и решать её хорошо» относится скорее к проектированию программных комплексов, нежели к процессу создания самих программ.

Из составляющих понятия Unix-way стоит особенно выделить принцип «всё есть текст». Свойства *текстового* представления произвольной информации и концептуальные отличия этого представления от всех остальных (*бинарных*) мы рассматривали ещё во вводной части (см. §1.4.6). Пожалуй, стремление представлять в виде текста любую информацию, за исключением такой, для которой это представление совершенно бессмысленно (поскольку всё равно не сделает формат пригодным для человека; это оцифрованная аналоговая информация вроде фотографий, аудио и видео, а также машинный код, сжатые данные и зашифрованная информация), тоже следует считать парадигмой, хотя и не совсем парадигмой программирования. Если так можно выразиться, текстовые данные — это особая в своём роде парадигма компьютерного представления информации.

Позднее, в §6.4.7, термин «парадигма» был применён к ***событийно-управляемому*** способу построения программы, при котором, как мы помним, в программе имеется ***главный цикл***, одна его итерация соответствует наступлению некоего события из определённого множества, а все действия программы построены как реакция на событие; сам главный цикл (точнее, его тело) делится на две стадии: ***выборка события*** и ***обработка события***.

Вся часть VII нашей книги была посвящена ***параллельному программированию***, хотя с его элементами мы столкнулись раньше — при изучении управления процессами. Параллельное программирование тоже обычно считается парадигмой, хотя во втором томе мы этого не сказали; в качестве отдельной парадигмы можно упомянуть и работу с ***разделяемыми данными*** — и это именно отдельная парадигма, хотя и связанная с параллельным программированием. Дело в том, что параллельное программирование как таковое может происходить и без разделяемых данных — например, если программа запустит несколько параллельных процессов, которые будут трудиться над одной задачей, но общаться при этом станут через сокеты или каналы; в языке Эрланг параллельное программирование считается основной парадигмой, но разделяемых данных там нет и быть не может, поскольку нет глобальных переменных. С другой стороны, если разделяемые данные находятся не в памяти, а в файлах на диске, и доступ к ним имеют не несколько процессов, относящихся к одной программе, а несколько независимых программ — то ни о какой парадигме параллельного программирования речи уже не идёт. Таким образом, эти две парадигмы — параллельное программирование и разделяемые данные — вполне способны существовать друг без друга.

Можно ещё добавить, что неоднократно использовавшаяся нами ***рекурсия*** — это тоже парадигма, хотя и несколько другого сорта; проблема в том, что всё это не слишком приближает нас к ответу на один простой вопрос: *а что такое, собственно говоря, «парадигма программирования»*? Из контекстов, в которых мы ранее упоминали этот загадочный термин, можно догадаться, что при наличии принципиально разных способов решения одной и той же задачи где-то рядом с этими способами как раз и располагаются парадигмы; больше того, существуют языки программирования, совсем не похожие на те, что мы рассматривали до сих пор, и они то ли стимулируют иные способы мышления, то ли требуют от программиста уметь мыслить иначе, и это тоже имеет отношение к парадигмам. Всё это так и есть, но коль скоро мы собираемся на некоторое время (до конца книги) сделать парадигмы программирования предметом изучения, стоит, по-видимому, всё-таки понять, чтó же это за предмет, и смутные ощущения из серии «где-то там рядом» нас не устроят.

Рассматривая термин «парадигма программирования» более внимательно, мы можем обнаружить, что даже среди авторов, посвятивших парадигмам программирования статьи и монографии, нет уверенного согласия в том, что считать парадигмами, а что не считать. Так, Роберт Флойд в своей лекции [2] называет в качестве примера парадигмы хорошо известное нам *структурное программирование* (см. т. 1, §2.4.1); при этом другие авторы (см., например, [4]) предпочитают считать, что структурное программирование к числу парадигм программирования не относится. Ясно, что о *строгом определении* такого термина и речи быть не может, хотя многие авторы такие определения давать пытались. Получилось, прямо скажем, так себе: определения, взятые из разных работ, противоречат друг другу, и вдобавок понять их (практически все) можно разве что уже зная, что такое парадигма.

Часто (и отнюдь не только в программировании) решить проблему помогает обсуждение, откуда она взялась и *зачем* её нужно решать, то есть что мы получим, решив проблему, или же, к примеру, что нам мешает оставить проблему без решения. Попробуем так поступить и мы: попытаемся для начала понять, по каким таким причинам мы не можем далее обойтись без загадочного слова «парадигма» и что оно нам даст.

9.1.2. Парадигмы вне программирования

Слово «парадигма» заимствовано из греческого языка; оригинальное греческое $\pi\alpha\rho\alpha\delta\epsilon\iota\gamma\mu\alpha$ можно перевести как *пример* или *модель*, но ни тот, ни другой перевод не передаёт настоящего смысла (что и понятно: иначе зачем нам потребовалось бы слово «парадигма» в русских текстах, а равно и в английских). Своим современным значением в научно-техническом контексте термин «парадигма» обязан, по-видимому, Томасу Куну и его книге «Структура научных революций» [5], где парадигмами назывались устоявшиеся системы научных взглядов, в рамках которых ведутся исследования. Согласно Куну, в процессе развития научной дисциплины может произойти замена одной парадигмы на другую — как, например, геоцентрическая небесная механика Птолемея сменилась гелиоцентрической системой Коперника, а теория флогистона уступила место современной неорганической химии, основанной на окислительных и восстановительных реакциях, — при этом старая парадигма продолжает ещё некоторое время существо-

вать и даже развиваться, поскольку её сторонники по тем или иным причинам не могут или не хотят перестраиваться для работы в другой парадигме.

Достаточно популярно слово «парадигма» и у гуманитариев, в особенности у тех, кто рассуждает о человеческом обществе и процессах в нём: читатель наверняка сможет припомнить, что словосочетание «смена парадигм» в применении к социально-политическим явлениям ему не раз попадалось. Отметим, что ораторы, заявляющие с разнообразных трибун о «назревшей необходимости смены парадигм», далеко не всегда сами могут объяснить, о чём, собственно говоря, идёт речь.

Внимательно рассмотрев всевозможные контексты, в которых авторы самых разных текстов употребляют слово «парадигма», мы можем заметить, что речь в большинстве случаев идёт не о каких-то объективных явлениях, а о *различных способах восприятия человеком (людьми) одного и того же явления*. Смена научных парадигм по Куну — это постепенный переход от одного подхода (объяснения объективно существующего явления) к другому подходу, объясняющему *то же самое явление*, но *лучше* (в том или ином смысле). Когда же мы слышим о «смене парадигм» от политиков, обществоведов и иже с ними, речь, как правило, идёт о том, что нужно как-то по-новому осмыслить происходящее в обществе, подвергнуть ревизии свои (политические или философские) взгляды на социально-экономические реалии, поскольку старый вариант мышления нас (или «их», или ещё кого-то) по той или другой причине устраивать перестал.

Как мы увидим позднее, понимание «различных парадигм» как *различных подходов к осмыслению (человеком) одного и того же явления* практически идеально подходит к случаю интересующих нас *парадигм программирования*, так что именно этот вариант объяснения смысла слова «парадигма» мы и зафиксируем в роли рабочего; но, прежде чем вернуться к нашему предмету — программированию — попытаемся для большей ясности привести несколько отвлечённых примеров парадигм, не имеющих к программированию никакого отношения.

Первый пример наверняка хорошо знаком всем читателям этой книги: в его роли выступит «школьная» (т. е. элементарная) геометрия. Известно, что точки, прямые и плоскости — первичные объекты традиционной евклидовой геометрии — представляют собой отвлечённые абстракции, не имеющие никакого отношения ко всевозможным рисункам, чертежам и прочим *визуализациям*, как и наоборот: как бы мы ни старались изобразить «точку» на бумаге, доске или другом носителе графической информации, получится у нас никакая не точка, а пятно, притом всегда можно подобрать достаточную степень увеличения, чтобы это пятно оказалось имеющим совершенно неправильную, грубую форму; с прямыми дело обстоит ещё хуже, ведь изобразить бесконечную прямую мы никак не можем, когда же дело доходит до плоскостей, любые иллюстрации становятся ещё более условными, полагаясь на воображение зрителя. Так или иначе, доказывать теоремы и решать задачи по геометрии можно, вообще говоря, не прибегая ни к каким рисункам, для этого достаточно иметь формулировки аксиом и уметь пользоваться правилами логики; рисунки нужны лишь для облегчения понимания происходящего. Встречаются, хотя и редко, люди, воспринимающие геометрию точно так же, как алгебру и многие другие разделы математики, в виде утверждений и формул — и без всяких рисунков. Поскольку речь на самом деле идёт об одном и том же предмете, можно смело заявить, что работа с объектами школьной геометрии без применения рисунков — это не что иное, как *парадигма*, как, впрочем, парадигмой следует считать и традиционный подход к геометрии, предполагающий использование иллюстративных чертежей и эскизов.

При более внимательном взгляде на всё ту же школьную геометрию можно обнаружить, что и с визуализацией её объектов всё не так просто, как кажется: точки, прямые и плоскости каждый человек на самом деле представляет для себя по-своему. Так, автору этих строк прямые всегда казались этакими подвешенными в пространстве «прутиками» *коричневого цвета*; причиной тому стал использовавшийся в школе в 1980-е годы учебник геометрии Погорелова, в котором иллюстрации были напечатаны коричневой краской. Получается, что каждый, кто имеет дело с элементарной геометрией, использует для этого, строго говоря, свою собственную парадигму. То же самое можно сказать про многие другие предметные области; взять хотя бы *указатели*, на которые мы потратили целую главу первого тома (см. т. 1, гл. §2.10).

Как правило, при изучении очередного нетривиального предмета ученикам приходится самостоятельно изобретать для него свою собственную парадигму. В большинстве случаев при этом ни сами ученики, ни их учителя толком не осознают, что речь здесь и сейчас идёт о формировании в мозгу обучаемого новой парадигмы, так что успех этого процесса для ученика оказывается в большой степени продуктом везения, учитель же уподобляется легендарному инструктору по плаванию из древней Спарты: берём ученика и кидаем в воду, выплывет — молодец, цель достигнута, ну а утонет — и чёрт с ним. Рискнём заявить, что хороший учитель в основном отличается от плохого именно своей способностью оказать ученику действенную помощь в формировании подходящей парадигмы.

Сделаем теперь неожиданный финт и обратимся к области совершенно иной, вовсе никак не связанной ни с программированием, ни с математикой, ни даже (на первый взгляд) вообще с наукой — катанию на горных лыжах; как мы сейчас увидим, парадигмы — реальность вездесущая, точнее, они есть везде, где есть реальное явление, которое люди вынуждены осмысливать.

Если внимательно посмотреть на людей, заполнивших склон на каком-нибудь горнолыжном курорте, можно достаточно легко выделить тех, кто хорошо умеет кататься — их стойка выглядит красиво и непринуждённо, колени расположены рядом, отсутствуют ненужные громоздкие движения всем корпусом, присущие, увы, большинству горнолыжников-любителей из-за недостатка технической подготовки. Оставив в покое спортсменов и просто опытных горнолыжников, мы можем обратить внимание, что среди всех остальных, катающихся «неправильно» и именуемых в просторечии «чайниками», резко выделяются те, кто не просто «неправильно катается», а совсем не умеет этого делать. Дело тут даже не в напряжённой («деревянной») позе и не в судорожных движениях, хотя всё это хорошо заметно. Существует один тривиальный признак, по которому можно отличить совершенно зелёного новичка от человека, который, хотя и не имеет серьёзной подготовки, всё же встал на лыжи далеко не в первый раз. Попробуем с этим разобраться.

При спуске с горы на горных лыжах едва ли не самое важное — сохранять контроль за скоростью. Если направить лыжи вниз по склону и не предпринимать никаких действий для торможения, даже на достаточно пологих горках лыжник станет очень быстро разгоняться и уже через несколько секунд достигнет такой скорости, при которой не сможет сохранить стабильность своего движения; первая же самая незначительная неровность приведёт к весьма зрелищному падению с фонтаном снега, разлетающимися в разные стороны лыжами и прочими спецэффектами.

Избежать неконтролируемого разгона можно, если спускаться по извилистой траектории, чередуя правые и левые виражи — это позволяет притормозить и сбросить излишек скорости. И вот тут мы как раз подбираемся к тому, ради чего затеяли всё обсуждение. Конечно, необходимость движения зигзагом становится ясна каждому, кто встал на горные лыжи, без этого спуститься с горы вообще невозможно. При этом отличить «зелёного новичка» от пусть и «чайника», но чуть более искушённого в происходящем, можно *по направлению взгляда*: новичок всегда смотрит туда же, куда направлены носки его лыж, тогда как горнолыжник, успешно преодолевший стадию перворазника, направляет взгляд туда, *куда хочет попасть* — обычно вниз по склону, а в общем случае туда, куда он движется *без учёта* виражей, нужных для подтормаживания. Несложно обнаружить, что здесь тоже имеет место *смена парадигмы*: перворазник считает, что движется «змейкой», тогда как более опытный лыжник воспринимает виражи вправо-влево как нечто вспомогательное, не влияющее на основную траекторию движения — прямую, ведущую к цели, или плавную кривую, огибающую препят-

ствия. Ясно, что объективное явление — спуск с горы — у двух рассматриваемых субъектов общее, и от смены направления взгляда *механически* вроде бы ничего не меняется; однако на самом деле лыжник, изменив направление своего взгляда, из всей имеющейся скорости начинает воспринимать в качестве «своей» лишь её проекцию на воображаемую траекторию — ведущую прямо к цели линию, вдоль которой как раз и направлен взгляд. Как ни странно, контролировать скорость при этом становится намного проще, а заодно вместо мельтешения фрагментов окружающего пейзажа (неизбежных при постоянной смене направления взгляда) лыжник видит более-менее постоянную картину со всеми препятствиями, которые вскоре предостоит объезжать, и это добавляет его действиям столь важной уверенности.

В целом горные лыжи могут служить прекрасным примером того, как небольшая сознательная модификация используемой парадигмы («я еду не зигзагом, а прямо вон туда, а виражи тут чисто вспомогательные») может облегчить жизнь и повысить эффективность наших действий. Заодно можно проиллюстрировать разницу между обычным инструктором и *хорошим* инструктором. Фразу «взгляд направлен в долину, плечи развёрнуты вниз по склону», разумеется, произнесёт любой инструктор, если он хоть что-то собой представляет; но редкий инструктор при этом ещё и объяснит, *зачем* это нужно — и если вам достался именно такой инструктор, можете считать, что вам очень крупно повезло.

Вернёмся к нашему предмету. Термин «парадигма программирования», судя по всему, первым ввёл Роберт Флойд в уже упоминавшейся выше лекции [2]. Ссылаясь на Томаса Куна, Флойд отмечает, что явление, подобное куновской смене научных парадигм, можно иногда наблюдать и в программировании, но в основном парадигмы программирования не исключают одна другую; напротив, они могут сочетаться, обогащая инструментарий программиста: «*Если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм,* — пишет Флойд, — *то совершенствование искусства отдельного программиста требует, чтобы он расширял свой репертуар парадигм*».

Можно считать, что наш третий том, который вы сейчас читаете, целиком посвящён именно этому *расширению репертуара*. Остаётся, впрочем, один вопрос, на который мы не сможем дать ответа: если набор парадигм, которыми владеет отдельно взятый программист, достаточно широк, то как выбрать наиболее подходящую парадигму для конкретной задачи? Объективного решения здесь нет, во многом использование той, а не другой парадигмы зависит от личных вкусов программиста. С уверенностью можно сказать только одно: чем лучше вы будете уметь переключать собственный мозг с одной модели на другую, чем бóльшей гибкостью будет обладать ваше мышление, тем выше ваши шансы выбрать оптимальный путь решения для любой вставшей перед вами технической проблемы.

9.1.3. Обзор основных стилей

Все языки программирования, с которыми мы имели дело до сей поры, относятся к большой группе языков, называемых ***императивными***. Создавая программу на таком языке, мы обычно представляем себе её выполнение в некой «среде», в которую включаем имеющиеся переменные с их значениями (иначе говоря, оперативную память, точнее — её часть, находящуюся в распоряжении нашей программы), а также внешние по отношению к программе сущности, такие как потоки ввода-вывода, другие выполняющиеся программы (процессы), да и вообще всё, на что наша программа может повлиять и что может повлиять на неё; так, для программы, ведущей диалог с пользователем, в состав «среды выполнения» попадает и сам пользователь, ведь выполняющаяся программа воздействует на него, выдавая сообщения, а пользователь, в свою очередь, оказывает влияние на выполнение программы, вводя информацию с клавиатуры, нажимая кнопки и совершая другие действия, которые наша программа может обнаружить. Когда программа управляет каким-то внешним оборудованием — да хоть бы даже и освещением в комнате — то частью среды выполнения становится это оборудование (взаимное влияние тут очевидно), и т. д.

Определяющим для императивного программирования оказывается не сама по себе среда выполнения (в конце концов, она есть *всегда*, в каком бы стиле мы ни писали программу), а скорее некоторые свойства этой среды. Прежде всего, *среда выполнения программы обладает* ***состоянием****, которое изменяется с течением времени.* Вот это вот изменение с течением времени может казаться нам сейчас (с высоты уже имеющегося опыта) фактором очевидным, тривиальным и даже не заслуживающим внимания, но, как мы вскоре увидим, это лишь сила привычки. Вспомним, что программа всегда реализует некий *алгоритм*, а алгоритм есть не что иное, как *преобразование информации* — или, говоря строже, каким-то конструктивным образом заданное отображение (в математическом смысле) из множества слов над неким избранным алфавитом в само это множество[1]. Где здесь какие-то «изменения с течением времени»? Программируя на Паскале, Си и языке ассемблера, мы часто забываем о математической сущности алгоритма, но, как может уже догадаться читатель, не все языки программирования императивны — просто мы пока что других не видели.

Возвращаясь к изменению состояния с течением времени, мы обнаруживаем, что основная часть интересующих нас при написании программы изменений происходит, что вполне ожидаемо, в результате *действий* самой программы — то есть программа, выполняясь, производит действия, которые что-то где-то (на самом деле — как раз в среде выполнения) *изменяют*. Ключевое слово тут — *действие*; если же вспомнить, что программа сама себя выполнять не умеет, её выполняет тот или иной *исполнитель* (в случае компилируемых языков — центральный процессор, в случае интерпретации — интерпретатор) — то вместо действий программы мы обнаруживаем действия исполнителя, а сама программа оказывается набором *указаний*, какие именно действия должны выполняться. Собственно говоря, именно концепция *программы как набора указаний, подлежащих исполнению*, даёт название всему стилю: слово «императив» можно считать синонимом слова «приказ».

«Приказы» эти, как мы знаем, бывают очень разными: ввод и вывод информации, всевозможные сигналы, установление и разрыв соединений — чего только работающие программы не делают с тем, что их окружает; но, пожалуй, самым простым, очевидным и при этом самым часто встречающимся действием в императивных программах остаётся ***присваивание***: есть некая переменная, в ней находится какое-то значение (возможно, неопределённое, если переменная только что появилась), но программа не обращает на это значение никакого внимания, а вместо этого *заносит в переменную новое значение*, попутно, разумеется, стирая (разрушая) старое. **Если задаться целью отыскать одну «самую главную» особенность императивного программирования, то, несомненно, этой особенностью окажется присваивание**.

Программисты, как начинающие, так и опытные, часто недооценивают присваивание, воспринимая его как нечто само собой разумеющееся; между тем именно это понятие оказывается первым (и даже, возможно, самым серьёзным) входным барьером на пути новичка, решившего научиться программировать. Большинство программистов пребывает в искреннем недоумении, как это возможно — не понять, что такое переменная и что такое присваивание; автор этих строк на основании своего преподавательского опыта готов засвидетельствовать, что это *действительно* очень трудно понять человеку, не имеющему склонности к программированию. Можно даже сделать более сильное утверждение: отдельно взятый новичок либо понимает переменные и присваивания сразу же, без каких-либо объяснений, либо не понимает их вовсе, вплоть до полной невозможности ему что-либо на эту тему объяснить. Промежуточный вариант вида «сначала не понял, потом объяснили,

[1] Это мы обсуждали в т. 1, §1.3.5; строго говоря, алгоритм сам по себе не есть такое преобразование, правильнее будет сказать, что *алгоритм реализует преобразование* в том смысле, что одно и то же преобразование (как математическая функция) может быть реализовано разными алгоритмами, и таких алгоритмов — счётная бесконечность для каждого конструктивного преобразования; в то же самое время существует (математически) множество преобразований, имеющее мощность континуум, для которых алгоритма не существует — хотя бы потому, что таких преобразований континуум, тогда как множество всех алгоритмов — не более чем счётное.

и я понял» встречается настолько редко, что впору усомниться в его возможности: немногочисленные случаи такого рода можно списать на то, что либо ученик отвлёкся на что-то постороннее, когда ему рассказывали про присваивание, либо учитель (или кто там вместо него) был совсем не в форме — короче говоря, на то, что понимание с первого объяснения не было достигнуто чисто случайно, и это не было обусловлено особенностями мышления обучаемого.

Как правило, когда возникают сложности с пониманием присваивания (даже если в итоге объяснить его всё-таки удаётся), дальше такому ученику оказывается невозможно объяснить циклы, особенно вложенные. Судя по всему, таким ученикам попросту не хватает воображения, чтобы как-то внутренне визуализировать для себя *выполнение программы с течением времени*. Путь в программирование этим людям, увы, заказан.

Вспомним теперь, что переменная — это не что иное, как область памяти, во всяком случае в Паскале и Си (и, добавим, в других императивных языках тоже), ну а память в том виде, в котором мы её знаем — с ячейками и адресами — одна из основных особенностей концепции, известной как «машина фон Неймана». Императивное программирование часто поэтому называют ***фоннеймановским***; первым (или, во всяком случае, одним из первых) такую терминологию применил Джон Бэкус [1].

Строго говоря, классический фоннеймановский стиль достаточно сильно отличается от современного. Так, большинство ранних вычислительных машин не поддерживало косвенную адресацию (см. т. 1, §3.2.5); обращение к элементам массива на таких машинах реализовывалось способом, крайне непривычно выглядящим с современной точки зрения: программа *изменяла адрес операнда* в машинном коде одной из своих собственных команд, после чего передавала на эту команду управление. Поэтому программа, которая прямо во время исполнения изменяет сама себя, на заре компьютерной эры была вполне обычным явлением, тогда как сейчас мы бы ничего подобного сделать не смогли, даже если бы хотели: при работе под управлением мультизадачных операционных систем машинный код процесса хранится в области памяти, которую процесс может читать, но не изменять. Можно отыскать и другие серьёзные концептуальные отличия, и в этом нет ничего удивительного: программирование как практическая дисциплина существует уже больше восьмидесяти лет, с тридцатых годов прошлого века. Неизменным в фоннеймановском программировании остаётся основа: переменные и присваивание.

Довольно часто можно встретить упоминания ***процедурного программирования***, причём некоторые авторы утверждают, что императивное и процедурное программирование — это попросту одно и то же. Такое утверждение легко объяснить: императивные языки, не предполагающие подпрограмм, благополучно вымерли, так что сегодня нам сложно представить, как это так — императивное программирование без возможности обособления отдельных частей программы. Тем не менее такое часто встречалось в те времена, «когда компьютеры были большими, а программы — маленькими».

Так, в своё время (вплоть до начала 1990-х) был очень популярен язык Бейсик, а под ним в те времена понималось довольно странное сооружение, в котором в начале *каждой* строки программы ставили её номер. Работа с интерпретатором Бейсика не требовала и не предполагала использование внешнего редактора текстов; пользователь вводил строку, а интерпретатор эту строку тут же выполнял, но при этом «выполнение» строки, начинающейся с числа (номера) состояло в том, что эта строка вставлялась в текст программы. Если в программе уже была строка с таким номером, она заменялась; для удаления строки в некоторых интерпретаторах можно было ввести её номер без дальнейшего содержимого, в других это использовалось, чтобы вызвать строку на редактирование, а удаление делалось специальной командой. Сейчас сложно представить, зачем такое могло потребоваться; но на протяжении 1980-х годов и даже в начале 1990-х были достаточно популярны домашние компьютеры, использовавшие в качестве единственного внешнего запоминающего устройства *кассетный магнитофон*. В памяти компьютера могла находиться только одна программа, а чтобы запустить другую, на нахождение нужного места на кассете и считывание программы в память уходило несколько минут, причём перематывать кассету в поисках начала записи нужной программы приходилось, естественно, вручную. В таких условиях привычный нам рабочий цикл «редактор, транслятор, запуск» организовать, мягко говоря, затруднительно.

Кроме управления вводом, номера строк использовались ещё и в роли меток для операторов `GOTO`, которые в Бейсике встречались в изобилии — например, чтобы построить привычное нам ветвление по схеме if-then-else, приходилось задействовать *два* оператора `GOTO` подобно тому, как, работая на языке ассемблера, мы использовали для той же цели два перехода — сначала по противоположному условию, чтобы прыгнуть на начало ветки else, а потом безусловный, чтобы *перепрыгнуть* через ветку else после выполнения ветки then. Обычно в Бейсике строки программы при вводе нумеровались «через десятку», чтобы можно было вставить новую строку между уже имеющимися; когда очередную строку вставить оказывалось некуда, давали команду «перенумерования» (`RESEQ`), и интерпретатор снова расставлял номера строк «через десятку», автоматически заменяя старые номера на соответствующие новые в операторах `GOTO`.

Строка в этом варианте Бейсика была единственной структурной единицей программы. Некие зачатки подпрограмм всё же присутствовали в виде операторов `GOSUB` и `RETURN`: первый делал переход на заданную строку с запоминанием номера строки для возврата, второй возвращал управление обратно; рассматривать это как полноценные подпрограммы было невозможно, поскольку язык не предусматривал ни локальных переменных, ни передачи параметров (их приходилось передавать через переменные, а все переменные были глобальными). Собственно говоря, язык вообще не предусматривал ничего для *обособления* фрагмента программы, оставляя эту работу воображению программиста.

В основе процедурного программирования лежит сам принцип выделения подзадач в обособленные части программы, называемые ***процедурами*** (или подпрограммами, или функциями, как в Си) — так, что вызывающему нет никакого дела до внутреннего устройства вызываемого, а вызываемому — до устройства, целей и задач вызывающего. Такое обособление, называемое ***декомпозицией***, позволяет, как мы знаем, эффективно бороться с нарастающей сложностью программы: мы воспринимаем сколь угодно сложное решение некоторой подзадачи как один оператор вызова процедуры, полностью выкинув из головы то, *как* эта процедура делает своё дело; и, напротив, решая (в виде процедуры) частную подзадачу, мы можем временно забыть, зачем эта подзадача нам потребовалась и как наша подпрограмма будет использована в вызывающей программе.

Полноценное обособление немыслимо без локальных переменных и передачи параметров — в противном случае нам придётся всё время помнить, какие из имеющихся переменных задействованы в каждой конкретной подпрограмме. После появления локальных переменных следующий естественный шаг — знакомая нам реализация локальных переменных через стековые фреймы, делающая подпрограммы ***повторновходными***, или ***реентерабельными*** (от английского *reenterability*) и позволяющая использование рекурсии. Считать ли рекурсию неотъемлемой частью процедурного программирования или же его развитием — дело вкуса, хотя автору приходилось видеть компиляторы Паскаля, по умолчанию размещавшие локальные переменные подпрограмм в сегменте данных (примерно так, как компилятор Си размещает локальные переменные, описанные со словом `static`); подпрограммы, предполагающие использование рекурсии, в этих версиях Паскаля требовалось пометить особым словом.

Вспомнив язык ассемблера, мы можем заметить, что там процедуры тоже существуют лишь в нашем воображении. Использовать стек для размещения локальных переменных мы вроде бы научились, но при этом прекрасно помним, что никакой поддержки со стороны ассемблера для этого не предусмотрено: даже дать локальным переменным имена мы могли бы разве что с помощью макропроцессора. Можно сказать, что ассемблер по своей сути императивен, поскольку всё императивное программирование обусловлено фоннеймановскими принципами работы процессора; при этом процедурным программированием на языке ассемблера заниматься *возможно*, но не в силу свойств языка, а только благодаря квалификации программиста.

Так или иначе, процедурное программирование — это, несомненно, разновидность программирования императивного, но эти два понятия далеко не тождественны.

Обсуждая императивное программирование, нельзя не упомянуть ещё одну его разновидность — программирование ***командно-скриптовое***, с которым мы уже встречались в первом томе при обсуждении скриптов на языке командного интерпретатора (см. §1.2.15). Этой парадигме мы посвятим несколько глав последней части нашей книги, пока же отметим несколько ключевых особенностей командно-скриптовых языков. Прежде всего следует сказать, что эти языки *чисто интерпретируемые*. Вторая их особенность — *текстовая строка как универсальное представление как программы, так и данных*. В частности, переменные в командно-скриптовых языках идентифицируются их именами — и более никак, то есть нет никаких указателей, адресов и т. п., но, с другой стороны, обычно возможно хранить *имя* од-

ной переменной в другой переменной и по имени обращаться к значению; вряд ли кому-нибудь придёт в голову заявить, что переменные в командно-скриптовых языках — это «области памяти», ведь никаких возможностей, обусловленных фоннеймановским понятием памяти, в этих языках нет. Можно сказать, что командно-скриптовое программирование является императивным, но не является фоннеймановским.

Полной противоположностью императивного подхода можно считать ***функциональное программирование***, в котором, если рассматривать его «чистым», присваивания нет вообще. Более того, чистое функциональное программирование в принципе не предполагает никаких «изменений с течением времени» и, как следствие, никаких «модифицирующих действий». В случаях, когда в функциональный язык программирования вынужденно включаются посторонние для него модифицирующие возможности, для них даже используется другой термин: ***разрушающие действия***. Термин вполне оправдан: в самом деле, если в переменную занести новое значение (то есть выполнить привычное нам присваивание), старое значение при этом будет утрачено — *разрушено*. Вообще говоря, термины «модифицирующее действие» и «разрушающее действие» эквивалентны, но второй при этом имеет очевидную отрицательную коннотацию, что соответствует общему отношению сторонников функционального программирования к любым модификациям среды выполнения.

Переменные в функциональном программировании обычно всё же используются (хотя существуют и языки без переменных, во всяком случае, в привычном нам смысле), но своё значение переменная получает при её создании и не меняет его в течение всего срока своей жизни. Чаще всего переменные используются в роли формальных параметров функций: при вызове функции такие переменные получают значения, соответствующие фактическим параметрам, и, что вполне ожидаемо, своих значений не меняют, пока работа функции не завершится — а после этого исчезают, как и все локальные сущности. Кроме того, при программировании на функциональных языках часто применяются локальные переменные, вводимые больше для удобства, чтобы поименовать значение того или иного вычисленного выражения (например, если это значение нужно использовать несколько раз). Уместно будет напомнить, что ***инициализация*** переменной — это совершенно не то же самое, что *присваивание*, и с этим принципиальным различием мы уже встречались (см., например, т. 2, §4.3.17). Дело в том, что, инициализируя переменную, мы задаём ей *начальное значение*, то есть создаём значение там, где раньше никакого значения не было — ведь переменная-то как раз сейчас создаётся. Следовательно, **инициализация, в отличие от присваивания, разрушающей операцией не является** — ей нечего разрушать.

Впрочем, отсутствие разрушающих действий — это далеко не главная особенность функционального программирования. В основе этой парадигмы лежат, как можно догадаться из названия, ***функции***: программа состоит из функций, её выполнение заключается в ***вычислении*** этих функций, а сами функции представляют собой, как это обычно заявляется в литературе, «объекты первого класса» в том смысле, что функция сама по себе может быть значением переменной, её можно передавать в другие функции как аргумент, возвращать из функций в качестве значения, плюс к тому большинство функциональных языков программирования позволяет создавать функции (если угодно, функциональные объекты) во время выполнения.

Можно заметить, что всё перечисленное (за исключением разве что создания функций «на лету») для нас не новость: в хорошо уже знакомом нам языке Си программы тоже состоят из функций, выполнение программы есть фактически вычисление функции `main`, а указатели на функции позволяют манипулировать ими как объектами. Больше того, если привлечь средства динамического связывания, предоставляемые операционными системами[2], или даже просто знакомый нам вызов `mmap` со сравнительно несложной «обвеской», то станет возможно во время работы подгрузить из внешних файлов функции, которых в нашей программе не было на момент её написания. Если в дополнение к этому предположить, что в системе установлен компилятор Си, то (во всяком случае, теоретически) появится возможность программного создания новых функций с последующим их исполнением в той же программе. Но всё это не делает язык Си функциональным: логика его построения фактически принуждает программиста к использованию модифицирующих действий, а функции Си в большинстве случаев представляют собой обыкновенные подпрограммы, состоящие из императивных инструкций, задающих модифицирующие действия; в функциональном же программировании понятие функции приближается к своему математическому аналогу: отображению из области определения в область значений. Функция, *чистая* в смысле функционального программирования, всегда вернёт одно и то же значение, если её применить к одному и тому же набору значений аргументов, а ничего иного, кроме вычисления своего значения, чистая функция не делает; например, применение такой функции к кортежу из констант можно попросту *заменить* соответствующим значением, и ничего не изменится (кроме разве что времени выполнения программы, которое, очевидно, сократится). Ясно, что функции, с которыми мы имеем дело в программах на Си, в большинстве своём даже близко не похожи на этот «функциональный идеал».

Интересно, что, хотя Си и не является функциональным языком, на этом языке вполне можно при большом желании заниматься функциональным программированием. Для этого придётся запретить себе использование всех его модифицирующих операций — присваиваний и примкнувших к ним инкрементов и декрементов, а также всех библиотечных функций, которые делают что-то кроме вычисления своего значения (т. е. имеют ***побочный эффект***). С теоретической точки зрения мы здесь не должны ничего потерять: язык Си при этом останется *алгоритмически полным* в том смысле, что на нём можно будет реализовать машину Тьюринга (или любой другой из формализмов, заменяющих неуловимый объект под названием «алгоритм» в математических теориях, связанных с конструктивной вычислимостью; см. т. 1, §1.3.5) — и, как следствие, формально можно реализовать вообще любой алгоритм из числа известных человечеству на настоящий момент, ведь тезис Тьюринга пока что никто не опроверг. На практике, как водится, всё не так просто, и читатель наверняка уже успел почувствовать здесь какой-то подвох. Посмотрим, чего мы на самом деле лишимся, убрав из Си модифицирующие действия.

Для начала совершенно очевидно, что глобальные переменные становятся бессмысленными, но эта потеря, прямо скажем, не слишком велика: ранее мы неоднократно упоминали, что глобальные переменные — зло и их лучше не использовать. Второй момент может показаться не столь очевидным: в отсутствие модифицирующих действий становятся бессмысленными *циклы* — все сразу, так сказать, оптом. В самом деле, если при выполнении цикла ничего не меняется от итерации к итерации (а в отсутствие модифицирующих операций ничего и не может поменяться), то выполнять тело такого цикла раз за разом очевидным образом бессмысленно, нам с тем же успехом хватило бы и одного раза. Но и это, как ни странно, совсем не фатально: вместо циклов всегда можно воспользоваться рекурсией. Пусть нам, к примеру, нужно найти в строке заданный символ и вернуть его адрес. Вместо функции с циклом, например, такой:

```
const char *strchar(const char *s, char c)
{
    const char *p;
    for(p = s; *p; p++)
        if(*p == c)
            return p;
    return NULL;
}
```

можно написать рекурсивную функцию — вот так:

```
const char *strchar(const char *s, char c)
{
```

[2]Мы не стали рассматривать динамические библиотеки и связанную с ними механику, поскольку от них обычно существенно больше вреда, чем пользы; но это не значит, что их нет.

```
    if(!*s)
        return NULL;
    if(*s == c)
        return s;
    return strchar(s+1, c);
}
```

или даже вот так:

```
const char *strchar(const char *s, char c)
{
    return !*s ? NULL : *s == c ? s : strchar(s+1, c);
}
```

Функций в программе, конечно, станет намного больше, ведь цикл мы выносить в отдельную функцию не обязаны, тогда как для рекурсии своя функция требуется обязательно; но нас вроде бы никто не ограничивает в количестве написанных функций, даже наоборот — мы знаем, что декомпозиция программы на бóльшее число подпрограмм снижает её сложность.

Вообще-то, конечно, утверждение, что вместо цикла всегда подойдёт рекурсия, само по себе на грани фола, ведь циклы бывают долгими и даже бесконечными, а стек не резиновый. «Настоящие» языки функционального программирования обычно прибегают к так называемой ***оптимизации остаточной рекурсии***[3], благодаря которой стек расходуется существенно меньше, а в некоторых случаях может не расходоваться вовсе. Если результирующим значением текущей функции должно стать значение, вычисленное какой-то другой функцией — например, вот в такой ситуации:

```
int f(int x)
{
    int y;
    /* ... некие вычисления ... */
    return g(y);
}
```

— то, очевидно, ***стековый фрейм***, созданный для работы функции `f`, после передачи управления функции `g` уже не будет содержать ничего полезного, и его можно ликвидировать *до* вызова `g`, а не после. Именно так «настоящие» функциональные языки и поступают, что позволяет даже бесконечные циклы заменять рекурсией без риска исчерпать стек. Иной вопрос, что компиляторы Си такого обычно не умеют. Кроме того, остаётся вопрос эффективности: циклы часто работают заметно быстрее, чем рекурсия. Трансляторы функциональных языков частично справляются и с этой проблемой, поскольку специально расчитаны на массированное использование рекурсии, но от Си такого ждать не приходится.

Впрочем, когда речь идёт о математическом понятии алгоритма, нам в любом случае приходится как-то преодолевать разницу между конечным компьютером и бесконечной лентой машины Тьюринга, и чаще всего мы для этого попросту пренебрегаем ограничениями, заявляя что-то вроде «допустим, память нашего компьютера бесконечна». В принципе, ничто не мешает с тем же успехом заявить, что в нашем распоряжении имеется бесконечный стек.

Итак, без циклов и тем более без глобальных переменных мы как-нибудь перебьёмся, но вот как быть с вводом-выводом? В самом деле, любое действие по выводу информации за пределы исполняющейся программы, как и любое действие по вводу информации извне, очевидным образом *изменяет среду выполнения программы*: где-то там, за пределами её памяти, исчезает информация из буферов ввода, а в буферах вывода, напротив, появляется, причём и то, и другое может повлечь за собой последствия, о которых наша программа даже не догадывается.

Вопрос с вводом-выводом остаётся головной болью для сторонников «неразрушающих» парадигм программирования, в том числе и функциональщиков. В большинстве случаев приходится идти на компромисс и допускать существование функций с побочным эффектом — как минимум для ввода-вывода, но часто дело этим не ограничивается, по принципу «увяз коготок — всей птичке пропасть»: как мы увидим позднее, в таких (якобы функциональных) языках, как Лисп и Scheme[4], присутствуют и присваивания, и глобальные переменные, и даже циклы. Справиться с вводом-выводом, не пожертвовав чистотой концепции, удаётся лишь в сравнительно редких языках программирования, основанных на принципе ***ленивых вычислений***, при которых часть вычислений функций (преобразований из выражения, содержащего вызов функции, в выражение, соответствующее вычисленному значению) не происходит до тех пор, пока значение не потребуется в других вычислениях; как мы увидим позднее, это позволяет работать с условно бесконечными структурами данных: например, от построенного «бесконечного» списка в памяти можно хранить лишь те элементы, которые уже кто-то затребовал, а остаток (который, собственно говоря, и заключает в себе «бесконечность») представлять в виде невычисленного (пока ещё не вычисленного!) выражения. Потоки ввода при этом можно представить в виде таких вот «бесконечных» последовательностей; результатом чтения из потока становится пара из прочитанного значения и самого потока (аналогично тому, как результатом декомпозиции бесконечного списка становится пара, состоящая из извлечённого из списка элемента и самого списка, точнее, строящей его вычислительной процедуры). С выводом ситуация сложнее: например, в языке Хоуп (Hope) можно построить главную функцию программы как постепенно вычисляющую элементы некоего списка (возможно, бесконечного, но не обязательно) и с помощью специально предназначенной для этого директивы указать, что элементы списка следует по мере их вычисления направлять в стандартный поток вывода. Оформленные таким способом программы будут совершенно чисты в плане побочных эффектов, вот только поток вывода здесь может быть ровно один.

Язык Хаскель (*Haskell*) — пожалуй, самый популярный сегодня среди «ленивых» языков программирования — использует более сложный подход, основанный на так называемых ***монадах***, а точнее — на монаде `IO`. Взаимодействие программы на Хаскеле с потоками ввода-вывода можно примерно описать так: главная функция, вычисляясь, порождает в качестве своего значения монаду, которая заключает в себе инструкции, когда что откуда прочитать и когда что куда записать. На самом деле благодаря «ленивости» всё выполнение программы и есть порождение этой монады, она строится функциями программы по мере использования.

К этому моменту у читателя наверняка возник вполне законный вопрос: а что такое, собственно говоря, «монада»? Автору этих строк один знакомый хаскелист как-то раз (причём давно, в самом начале 2000-х) сказал, что если он придумает, как объяснять новичкам, что такое монада, он уж точно перевернёт мир. Судя по тому, что мир с тех пор так и не перевернулся, удовлетворительного способа объяснить, что такое монада, не появилось. Интересно, что монада имеет строгое математическое определение, из которого совершенно невозможно понять, на кой чёрт ЭТО нужно и как им пользоваться.

Сам ваш покорный слуга в течение без малого двадцати лет руководил спецсеминаром «Парадигмы программирования», куда достаточно часто приходили студенты, знающие Хаскель и умеющие на нём писать. Одно время на семинаре едва ли не каждый год очередной энтузиаст из числа студентов-хаскелистов делал доклад на тему «что такое монады»; никакие два доклада из примерно полудюжины заслушанных не были похожи друг на друга, и, что вполне естественно, ни один из этих докладов не показал способа, как же всё-таки правильно этот странный предмет разъяснять.

Автор книги предпочтёт не браться за задачу, которая ему не по силам, и воздержится от попытки объяснить, что же такое монады в Хаскеле и как с ними бороться — как, впрочем, и от рассказа о самом Хаскеле.

Возвращаясь к языку Си, попробуем понять, как же всё-таки написать на нём программу в чистом функциональном стиле. Для начала заметим, что операции ввода нам не так уж и нужны: всё, что требуется для работы программы, можно (по крайней мере, теоретически — если считать, что мы должны реализовать некий алгоритм, то есть конструктивное преобразование из входной информации в выходную) передать через аргументы командной строки. С выводом всё несколько сложнее — если отказаться от традиционных операций вывода, то всё, что сможет сделать «чисто функциональная» программа — это вернуть операционной системе число от 0 до 255 — хорошо знакомый нам код завершения процесса; очевидно, этого никак не может быть достаточно для реализации всей полноты возможных алгоритмов.

Можно пойти по пути большинства функциональных языков и сделать исключение из общего правила: разрешить использование

[3] Оригинальный английский термин здесь — *tail recursion optimization*; прямой перевод этого термина «оптимизация хвостовой рекурсии» или просто «хвостовая рекурсия» довольно часто попадается в русскоязычных текстах.

[4] Произносится «*Ским*»; до сей поры названия языков программирования мы обычно писали по-русски, но здесь придётся отступить от нашей традиции — словоформа «Ским» смотрится слишком экзотично, а довольно распространённое среди русскоязычных программистов наименование «Схема» — откровенно жаргонное.

каких-нибудь (конечно, не всех) функций ввода-вывода, но при этом помнить, что больше никакие функции с побочными эффектами использовать нельзя. Если при этом не ограничиваться только выводом, то есть разрешить и операции ввода, то в таком виде можно будет переписать вообще практически любую программу на Си; разве что в некоторых случаях, учитывая отсутствие оптимизации остаточной рекурсии, придётся воспользоваться конструкцией бесконечного цикла вроде `for(;;)`, но эта конструкция сама по себе чистоты функционального стиля не нарушает — в отличие, увы, от операций ввода-вывода.

Впрочем, нашим иллюстративным целям лучше соответствует другой подход: можно снабдить программу некоторой «основной» функцией (не `main`), которая на вход в качестве аргументов получит командную строку прямо в том виде, в котором она передана функции `main`, а в качестве своего значения вернёт список строк, которые нужно выдать в стандартный вывод; за сам вывод будет отвечать функция `main`. Поскольку привычная нам библиотечная функция `malloc` тоже никак не может считаться чистой, создание нового элемента списка лучше тоже вынести в отдельную функцию, которая хотя бы снаружи будет выглядеть похожей на чистую. Получится что-то вроде следующего:

```
#include <stdio.h>
#include <stdlib.h>

struct str_item {
    char *str;
    struct str_item *next;
};

struct str_item *mkitem(const char *s, struct str_item *tail)
{
    struct str_item *p = malloc(sizeof(*p));
    p->str = malloc(strlen(s)+1);
    strcpy(p->str, s);
    p->next = tail;
    return p;
}

struct str_item *main_func(int argc, char **argv);

int main(int argc, char **argv)
{
    struct str_item *p, *tmp;
    p = main_func(argc, argv);
    for(tmp = p; tmp; tmp = tmp->next)
        fputs(tmp->str, stdout);
    return 0;
}
```

Здесь предполагается, что вся реализация того алгоритма, ради которого мы пишем программу, помещена в функцию `main_func` и другие функции, вызываемые из неё. В программе, построенной таким способом, функции `main` и `mkitem` могут (конечно, при соблюдении должной аккуратности) остаться единственными, где применяются разрушающие действия, а вся остальная программа будет полностью соответствовать принципам чистого функционального программирования. Конечно, *любую программу* мы так не перепишем: нам не удастся оформить в таком виде ни интерактивную программу, ни программу, работающую с файлами, ни сетевой сервер или клиент. Но если задача предполагает *преобразование одной информации в другую*, то в таком виде получится написать любое преобразование, если только оно алгоритмически разрешимо.

Следует отметить, что **с формальной точки зрения любая программа производит преобразование одной информации в другую** и больше ничего не делает. Всевозможные управляющие воздействия вроде хорошо знакомых нам системных вызовов вполне можно рассматривать как часть выходного (результирующего) слова, придумав соответствующую систему кодирования; любые внешние события точно так же можно считать частью входного слова. **Единственное, что не удаётся «загнать» в модель теории алгоритмов — это выполнение с учётом течения времени;** собственно говоря, течением времени можно было бы пренебречь, если бы окружение, в котором выполняется программа, не обладало способностью, во-первых, изменяться с течением времени само по себе (например, под действием пользователя, других программ, каких-то физических процессов и т. п.), и, во-вторых, реагировать на действия программы.

Если попытаться сформулировать происходящее в терминах традиционной теории алгоритмов, у нас получится, что на момент старта программы в общем случае существует *не всё* входное слово, а только какая-то его часть (сколько-то первых символов, возможно, даже ни одного), остальную же часть генерирует некий внешний по отношению к алгоритму «чёрный ящик» (т. е. нечто такое, внутреннее устройство чего не может быть изучено), причём этот чёрный ящик сам по себе не обязан быть исполнителем какого-либо алгоритма, он вполне может быть, в частности, недетерминированным, а главное — он работает с течением времени в том, например, смысле, что очередную порцию входного слова он может «зажать и долго не отдавать». Но самое, пожалуй, занятное свойство нашего источника входного слова состоит в том, что вот это вот «долго» может зависеть от собственных действий нашего алгоритма: в простейшем случае нам могут не отдавать очередную порцию входного слова до тех пор, пока мы сами не сгенерируем определённый фрагмент слова выходного. Именно этот аспект отличает те же интерактивные программы от алгоритмов в классическом смысле и не позволяет (опять-таки в общем случае) пренебрегать течением времени при рассмотрении выполнения программ. Между прочим, из-за чувствительности среды к действиям нашей программы может получиться знакомый нам ***тупик***: если наша программа будет ждать очередной порции входного слова, а её окружение окажется устроено так, что для генерации этой порции ему будет нужно получить что-то от нашей программы, то система «программа+окружение» благополучно зависнет; теория алгоритмов таких вещей не предполагает.

Следующая парадигма из числа основных, обычно упоминаемых в литературных источниках при попытке перечислить парадигмы — это так называемое ***логическое программирование***. Программа здесь формулируется в виде набора логических правил, описывающих некую предметную область, а исполнение программы состоит в попытке доказать утверждение (или, если уж совсем точно, в попытке подобрать контрпримеры к *отрицанию* утверждения).

Наиболее известным языком логического программирования можно считать язык Пролог; надо признать, что если о Прологе все, кто имеет отношение к программированию, хотя бы слышали, то о существовании других логических языков большинство программистов, судя по всему, даже не подозревает. Впрочем, при близком знакомстве с этим языком (которое у нас впереди) можно с удивлением обнаружить, что остатки его «логичности» куда-то испарились, а пролог-решатель — программный механизм, делающий выводы на основе заданных фактов и правил — внезапно превратился в машинку для тупого (или, возможно, совсем даже и не тупого) перебора вариантов. Дело тут, как водится, в недостаточном умении думать по-разному об одном и том же явлении — людям, привыкшим к императивным программам, попросту *легче* думать о выполнении программы на Прологе в терминах последовательности действий, выполняемых решателем. Впрочем, даже в роли машинки для перебора Пролог, несомненно, стоит времени, потраченного на его изучение: когда на практике встречается задача, связанная с перебором вариантов, лучше всего с ней справляются (притом независимо от того, какой применяется язык программирования) программисты, имеющие хотя бы небольшой опыт работы с Прологом.

Если функциональным программированием мы можем при желании заниматься на языках, не предназначенных для этого — хотя бы на том же Си — то с логическим программированием этот номер не проходит, поскольку для выполнения логических программ необходим решатель, устройство которого слишком сложно, чтобы просто взять и реализовать его. Можно сказать, что в основе функционального программирования лежит *отказ* от использования части возможностей языка, тогда как логическое программирование, напротив, требует инструментов, которых в большинстве языков не предусмотрено. Рассуждая о функциональном программировании, мы смогли составить о нём первое впечатление, пользуясь для этого только рассмотренными ранее языками. С логическим программированием так не получится, поэтому приведём небольшой фрагмент программы на Прологе.

Факты в пролог-программах (и вообще логическом программировании) выглядят примерно так:

```
parent(mary, ann).
parent(mary, john).
parent(george, john).
male(george).
```

```
male(john).
female(mary).
female(ann).
```

Здесь мы сообщили пролог-решателю, что Джордж и Джон — мужчины, Мэри и Энн — женщины, при этом Мэри приходится родителем Энн и Джону, а Джордж приходится родителем Джону (кто второй родитель Энн — не уточняется). На основе этого можно сформулировать решателю простейшие вопросы вроде «кто является родителями Энн» (`parent(X, ann)`) или «перечисли (известных тебе) мужчин» (`male(X)`), но, конечно, суть логического программирования не в этом. Кроме фактов, в программе могут также присутствовать ***правила вывода***:

```
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).
brother(X, Y) :- parent(T, X), parent(T, Y), male(X).
```

Теперь можно попросить решатель найти брата Энн (`brother(B, ann)`); решатель ответит, что таковым является Джон, поскольку можно подобрать такие значения переменных, что все условия в правой части соответствующего правила окажутся выполнены (`X=john, Y=ann, T=mary`). Отметим, что заданные в программе отношения между объектами — неважно, заданы ли они в виде фактов или правил — называют ***предикатами***; в нашем примере фигурируют шесть предикатов: `parent`, `male`, `female`, `father`, `mother` и `brother`.

Добавив в программу факты, касающиеся возраста и других анкетных данных рассматриваемых людей, мы сможем заставить решатель справляться с более сложными запросами вроде «найди нам мужчину, у которого есть младший брат и старшая сестра, а сам он при этом владеет французским языком, по образованию химик, но работает программистом». С помощью правил вывода можно будет описать новые предикаты, например, для поиска близких родственников в заданной возрастной группе и т. п.

Если воспринимать пролог-программу *правильно*, то есть в парадигме логического программирования, может возникнуть ощущение (заметим, очень полезное, ради него всё и делается), что мы задаём свойства нужного нам решения, а решатель «как-то там» находит подходящий объект, и нам при этом совершенно всё равно, как именно он это делает. Такую точку зрения на имеющийся инструмент, когда мы задаём (*декларируем*), *что* хотим получить, но не объясняем, *как*, называют ***декларативным программированием***.

Декларативное программирование не всегда связано с логическим. В нашей книге мы не рассматривали системы управления базами данных, поскольку невозможно объять необъятное; но для читателей, понимающих, о чём идёт речь, отметим, что языки запросов к СУБД, в том числе небезызвестный SQL, как раз являются декларативными, при этом тот же SQL никоим образом не относится к языкам логического программирования.

Коль скоро речь зашла о базах данных, стоит заметить, что *предикат* — это, собственно, то же самое, что в теории баз данных называют ***отношением*** (англ. *relation*), а на практике — попросту ***таблицей***; аналогичным образом одну и ту же сущность описывают термины «факт», «кортеж» и «строка таблицы». Весь набор имеющихся в пролог-программе фактов так и называют базой данных, и больше того, Пролог позволяет во время работы программы добавлять в неё новые факты и изымать имеющиеся, то есть базу данных можно динамически изменять (с правилами так поступать нельзя).

Идея применения логического программирования для работы с базами данных вообще выглядит очень естественной, но, к сожалению, SQL «успел первым», заняв нишу якобы «стандартного» языка запросов. В Интернете можно найти довольно много проектов, применяющих логическое программирование в области управления базами данных, часть из этих проектов основана на Прологе, часть — на строго логическом языке Дэйталог (*Datalog*), но чтобы эти проекты обнаружить, нужно точно знать, что именно вы хотите найти.

В Прологе допускаются предикаты, вычисление которых имеет ***побочный эффект***. Встроенные предикаты такого рода применяются для ввода-вывода, для динамического добавления и удаления фактов, а ещё — для явного управления процессом вычислений; всё это отнюдь не способствует мышлению в терминах логической парадигмы. Как мы увидим позже в главе, посвящённой Прологу, побочными эффектами дело не ограничивается, логическую семантику убивает такая безобидная вроде бы вещь, как арифметика, плюс к тому неожиданно «нелогичным» оказывается простое отрицание, и так далее. Так или иначе, приступая к программированию на Прологе, следует быть готовым к тому, что одним из основных инструментов построения программы оказывается хорошо уже знакомая нам рекурсия, а переменные в Прологе служат исключительно для обозначения неизвестных величин в правилах, они всегда локальны в отдельном предложении программы; аналога присваиванию в Прологе нет (хотя было бы ошибкой заявить, что переменная не может меняться со временем, но то, *как* переменные в Прологе меняют своё значение, ничего общего с императивным подходом не имеет).

При работе на Прологе мы в ряде случаев всё же вынуждены думать в терминах последовательности действий, выполняемых решателем (так называемой ***процедурной семантики***); некоторые другие языки логического программирования, в том числе упоминавшийся выше Дэйталог, этого недостатка лишены и имеют семантику, полностью отвечающую правилам математической логики.

Наш обзор «больших» парадигм остался бы неполным без ***объектно-ориентированного программирования*** (или, для краткости, ООП), которое, пожалуй, можно назвать самой популярной парадигмой в мире, если считать по упоминаниям — в литературе, в Интернете, просто в разговорах; конечно, императивным программированием люди по-прежнему пользуются чаще, и это вряд ли изменится, но не все, кто пишет императивные программы, помнят сам термин «императивное программирование», тогда как каждый, кому пришло в голову воспользоваться классами и объектами, гордо заявляет, что занимается ООП. При этом на вопрос, что́ же такое, собственно говоря, это ваше ООП, обычно можно услышать произносимые с невообразимым уровнем пафоса слова «инкапсуляция», «наследование», «полиморфизм», но объяснять эти слова люди обычно отказываются. Попробуем разобраться, что здесь к чему.

Постоянно растущие размеры программ всегда заставляли программистов искать пути для борьбы со сложностью. Первым шагом в этой борьбе стала хорошо известная нам *раздельная трансляция модулей*; имена, скрытые в отдельных модулях, позволяют ничего о них не знать при работе над другими модулями, что резко снижает общее количество информации, которую программист в каждый момент времени вынужден принимать во внимание. Этот простой факт показывает нам, что настоящим предметом борьбы здесь оказывается даже не столько сложность программ как их собственное свойство, сколько сложность их восприятия человеком — уже как свойство человеческого мышления.

Сокрытие деталей реализации той или иной подсистемы программы от других её подсистем — это и есть, собственно говоря, ***инкапсуляция***. Примерами инкапсуляции могут служить локальные переменные в подпрограммах, а также имена (подпрограмм, типов, глобальных переменных), видимые только внутри модулей, где они введены, и недоступные из других модулей.

Очевидно, что инкапсуляция бывает не только в объектно-ориентированном программировании: мы встречали её и в Паскале, и в Си, и даже при работе на языке ассемблера. С другой стороны, можно сказать, что объектно-ориентированное программирование выводит инкапсуляцию на качественно новый уровень. Осмысливая свою программу в соответствии с объектно-ориентированной парадигмой, мы прежде всего представляем данные в виде некоторых ***объектов*** — «чёрных ящиков». Внутреннее устройство объекта извне недоступно, а в ряде случаев может быть просто неизвестно — например, если данный объект реализован другим программистом. Всё, что можно сделать с объектом — это послать ему сообщение и получить ответ. Так, операция «$2 + 3$» в терминах объектно-ориентированного программирования выглядит как «мы посылаем двойке сообщение “прибавь к себе тройку”, а она отвечает нам, что получилось 5». Вполне возможно, что, получив сообщение, объект произведёт какие-то действия, сменит своё внутреннее состояние или пошлёт сообщение другому объекту, но всё это остаётся на его усмотрение; посылая объекту сообщение, мы не должны делать никаких предположений о том, как именно устроена (реализована) реакция объекта на данное сообщение.

Недоступность деталей реализации объекта за пределами его описания как раз и объясняет, почему инкапсуляцию столь настойчиво называют одним из «трёх китов» ООП. Конечно, инкапсуляция есть не только в языках, поддерживающих ООП, но если в других языках основной единицей инкапсуляции обычно является *модуль*, то в ООП мы используем инкапсуляцию на уровне отдельных объектов, что в ряде случаев позволяет резко снизить общую видимую сложность самих модулей.

Объекты, внутреннее устройство которых одинаково, образуют ***классы***. В прагматическом смысле класс — это описание внутреннего устройства объекта; при создании нового объекта указывается, к какому классу он принадлежит (объектом какого класса он будет). Имея описание класса, можно создать произвольное количество[5] объектов этого класса. Более того, можно взять имеющийся класс и на его основе создать новый класс, в чём-то похожий на старый и проявляющий его свойства, но при этом всё-таки отличающийся от него, как правило, в сторону усложнения (хотя и не всегда). Эта возможность называется ***наследованием***.

Часто бывает так, что несколько различных классов способны обрабатывать некоторый общий для них всех набор сообщений. В таком случае говорят, что эти классы поддерживают общий ***интерфейс***; во многих объектно-ориентированных языках программирования интерфейсы описываются в виде классов специального вида.

Остаётся последний из «трёх китов» — ***полиморфизм***. Строго говоря, полиморфизм тоже никоим образом не новость: это способность одинаковых конструкций языка (чаще всего выражений, хотя и не обязательно) обозначать различные действия в зависимости от *типов* задействованных переменных и значений. Например, выражение «`a+b`» в Паскале может означать сложение двух целых чисел, сложение двух чисел с плавающей точкой (изучив язык ассемблера, мы точно знаем, что это совсем другая операция), а может и вовсе обозначать конкатенацию двух строк. Изучая Си, мы столкнулись с тем, что выражение вида «`t[i][j]`» может вычисляться совершенно по-разному: одно дело, если `t` представляет собой указатель на указатель, и совсем другое — если `t` является именем двумерного массива (либо, что фактически то же самое, значением мозгодробительного типа «указатель на массив»). Надо сказать, что к ООП это всё не имеет совершенно никакого отношения.

В языке Си++, изучению которого мы посвятим всю следующую часть нашей книги, программист может сам вводить для своих типов операции, выглядящие так же, как встроенные операции над встроенными типами — то есть, по сути, управлять полиморфностью операций, встроенных в язык; но и это, как ни странно, не имеет отношения к ООП. Когда слово «полиморфизм» употребляется в составе пафосной триады — вместе со словами «инкапсуляция» и «наследование» — имеются в виду такие виды полиморфизма, которые возникают при наследовании классов: с объектом любого класса-наследника можно обращаться так же, как с объектом класса-предка, игнорируя тот факт, что объект вообще-то имеет другой тип; при этом сам объект на такое обращение вполне может отреагировать не так, как реагирует его предок. К этому вопросу мы вернёмся при изучении соответствующих возможностей Си++, пока же отметим, что из «трёх китов» лишь один — наследование — оказывается фирменной особенностью объектно-ориентированного программирования. Кстати, встречаются языки программирования, поддерживающие работу с классами и объектами, но не поддерживающие при этом наследование; иногда такой вариант «объектной надстройки» называется *object-based programming* (в отличие от *object-oriented*).

С объектно-ориентированным программированием часто (и безосновательно) смешивают другую парадигму программирования — ***абстрактные типы данных***. *Абстрактным* называют такой тип данных, для которого неизвестна его внутренняя организация, а известен лишь некий набор базовых операций; именно так мы воспринимаем, например, тип `FILE`, вводимый стандартной библиотекой Си для высокоуровневой работы с файлами — мы знаем, что можно описать переменную типа `FILE*` (т. е. указатель на `FILE`), что с ней можно работать, используя функции `fopen`, `fclose`, `fputc`, `fgetc` и т. п., но при этом нас не интересует, что́ в действительности представляет собой тип `FILE`. Если подумать, можно обнаружить, что к абстрактным относятся встроенные типы языка Си для обработки чисел с плавающей точкой — `float`, `double` и `long double`, поскольку в языке Си нет никаких средств, зависящих от конкретного представления «плавучих» чисел. В то же время целочисленные типы отнести к абстрактным не получится, ведь для них язык предоставляет побитовые операции, то есть внутреннее устройство целых чисел на уровне языка зафиксировано и доступно.

Путанице между ООП и АТД, несомненно, способствует наличие в обеих парадигмах неких требований «закрытости внутреннего устройства», но сходство на этом заканчивается. При работе с абстрактными типами данных не идёт никакой речи ни об «обмене сообщениями», ни о «внутреннем состоянии», ни тем более о наследовании или загадочном «полиморфизме». В большинстве случаев применение АТД не приводит к серьёзным изменениям стиля мышления, господствующая парадигма остаётся традиционной — императивной, тогда как правильный подход к ООП заставляет программиста полностью поменять образ мыслей.

Ключевое различие между объектами в смысле ООП и абстрактными данными можно проиллюстрировать тем фактом, что абстрактные типы данных можно присваивать[6] и это никак не противоречит избранной модели мышления, тогда как в «чистом» ООП присваивание оказывается чем-то чужеродным. В самом деле, обычных переменных «чистое ООП» не предусматривает, там существуют только объекты; присваивание (безотносительно конкретного языка программирования) есть не что иное, как указание сделать один объект таким же, как некий другой, то есть, присвоив один объект другому, мы будем точно знать, что теперь внутреннее состояние этих двух объектов одинаково, но ведь ООП запрещает какие-либо предположения о внутреннем состоянии объектов!

Между прочим, вот эта вот чужеродность присваивания для ООП позволяет опровергнуть довольно часто встречающееся утверждение, что якобы ООП — это то ли разновидность императивного программирования, то ли новый этап его развития, то ли ещё каким-то образом с императивным программированием связано. В действительности, конечно же, ООП — это полностью самостоятельная парадигма, а её восприятие «в связке» с императивным обусловлено лишь совокупной популярностью императивных языков программирования с объектно-ориентированными надстройками, таких как Си++, Джава, C#, Go и тому подобных; но частое появление ООП в одном языке с императивным программированием было бы неправильно рассматривать как основание для далеко идущих выводов о якобы «исходно императивной» природе ООП, причина здесь скорее иная: императивное программирование наиболее популярно из-за своей близости к машине фон Неймана; как следствие, именно оно лежит в основе большинства индустриальных языков программирования, ну а появление в этих языках объектной надстройки отвечает на вполне объективную потребность практического программирования в соответствующих возможностях: использование ООП даже за счёт одной только инкапсуляции способно изрядно снизить трудоёмкость при разработке крупных программ.

Надо признать, впрочем, что даже в таких объектно-ориентированных до мозга костей языках, как Smalltalk и Eiffel, присваивание всё же присутствует; кроме того, у императивного программирования и ООП есть одна весьма фундаментальная общая особенность: *состояние, которое изменяется с течением времени*, хотя работа с состоянием организована в этих двух парадигмах совершенно по-разному.

Список парадигм программирования, конечно же, не ограничивается перечисленными основными стилями. Напомним, что парадиг-

[5] Строго говоря, иногда можно встретить класс, объект которого может существовать только в единственном экземпляре, и даже такие классы, для которых вообще нельзя создавать объекты, но это скорее исключение из правил.

[6] Из этого правила тоже есть исключения. Например, файловые переменные языка Паскаль не присваиваются; впрочем, остаётся открытым вопрос, можно ли считать эту сущность абстрактным типом данных.

ма — это то, как человек предпочитает (здесь и сейчас) думать о каком-то явлении; достаточно очевидно, что и при написании императивной программы, и в рамках функционального, логического или объектно-ориентированного стиля всё ещё остаётся достаточно свободы манёвра для разнообразных интеллектуальных упражнений, связанных с различными способами осмысления одного и того же предмета. Заявив, что мы-де занимаемся функциональным (или каким-нибудь ещё) программированием, мы, разумеется, не сможем зафиксировать разом весь образ мыслей программиста; это вообще вряд ли возможно, особенно если учесть, что мышление каждого человека уникально.

Какова бы ни была «основная» парадигма, в рамках которой мы решили работать, всегда останется место для многообразия способов мышления — попросту говоря, в рамках любой парадигмы можно найти другие парадигмы, которые могут меняться. Функции и процедуры, присваивания и связывания, циклы разных видов, рекурсия — всё это оказывает влияние на восприятие. В дальнейшем тексте мы постараемся показать многообразие возможных парадигм на примерах. Уместно будет сразу же отметить, что нашей целью при этом будет отнюдь не освоение тех или иных приёмов *программирования* (хотя это тоже полезно); главное, чего мы здесь хотим добиться — это умения *думать по-разному*, переключать собственный мозг, если так можно выразиться, из одного режима в другой.

9.2. Рекурсия как пример парадигмы

В зависимости от того, на каком языке программирования мы работаем, наша потребность в применении рекурсии может сильно меняться: в программе на Прологе или Хоупе без неё не обойтись, в программе на Лиспе обойтись без рекурсии вроде бы можно, но результат будет выглядеть исключительно коряво; в программах на Паскале или Си рекурсия остаётся на усмотрение программиста, решение о её применении принимается исходя из личных предпочтений; в программе на языке ассемблера применять рекурсию — удел сильных духом; в скрипте, написанном на Bourne Shell, работающая рекурсия вызвала бы удивление и — почти наверняка — дружеский совет больше так не делать; наконец, если нам удастся найти работающий интерпретатор старого Бейсика с нумерованными строками или экзотического языка Фокал, то там применить рекурсию у нас бы вообще не получилось ввиду отсутствия областей видимости, локальных переменных и передачи параметров.

В этой главе мы попытаемся показать, что рекурсия представляет собой самостоятельную парадигму программирования, а с некоторых точек зрения — даже целый комплекс парадигм; заодно мы попытаемся — наряду с освоением рекурсии как «хитрого приёмчика» — осознать, что существует и такое явление, как *рекурсивное мышление*, и понять, в чём оно заключается.

9.2.1. Пример с обходом дерева

Для разминки рассмотрим хорошо нам известный пример — обход двоичного дерева поиска. Эту задачу мы уже решали и на Паскале (см. т. 1, §2.11.5), и на Си (т. 2, §4.5.4), причём на Си мы рассмотрели как рекурсивное решение, так и решение без рекурсии. Напомним, что рекурсивное решение очень простое. Если узел дерева описывается структурой

```
struct node {
    int val;
    struct node *left, *right;
};
```

и нам нужно напечатать все числа, хранящиеся в узлах дерева, то сделать это можно так:

```
void int_bin_tree_print_rec(const struct node *r)
{
    if(!r)
        return;
    int_bin_tree_print_rec(r->left);
    printf("%d ", r->val);
    int_bin_tree_print_rec(r->right);
}
```

Словесное описание принципа решения тоже легко понять: если дерево пустое, то делать ничего не надо (как полагается, базис рекурсии выбираем самый тривиальный из всех возможных), если же в дереве есть хотя бы один (текущий) элемент, то сначала печатаем левое поддерево, потом текущий элемент, потом правое поддерево.

Итеративный вариант оказывается намного сложнее. Вот он:

```
void int_bin_tree_print_loop(const struct node *r)     /* bintree.c */
{
    enum state { start, left_visited, completed };
    struct backpath {
        const struct node *p;
        enum state st;
        struct backpath *next;
    };
    struct backpath *bp, *t;
    bp = malloc(sizeof(*bp));
    bp->p = r;
    bp->st = start;
    bp->next = NULL;
    while(bp) {
        switch(bp->st) {
        case start:
            bp->st = left_visited;
            if(bp->p->left) {
                t = malloc(sizeof(*t));
                t->p = bp->p->left;
                t->st = start;
                t->next = bp;
                bp = t;
                continue;
            }
            /* no break here */
        case left_visited:
            printf("%d ", bp->p->val);
            bp->st = completed;
            if(bp->p->right) {
                t = malloc(sizeof(*t));
                t->p = bp->p->right;
                t->st = start;
                t->next = bp;
                bp = t;
                continue;
            }
            /* no break here */
        case completed:
            t = bp;
            bp = bp->next;
            free(t);
        }
    }
}
```

В этот раз мы действуем (во всяком случае, на первый взгляд) совсем иначе. Прежде всего, нам нужно помнить, каким путём возвращаться назад, к узлам-предкам, когда мы завершили обход поддерева. Для этого мы создаём односвязный список узлов, являющихся предками текущего узла. Сам текущий узел (точнее, указатель на него) тоже включается в этот список, чтобы при работе в поддеревьях, исходящих из этого узла, мы знали, как в него вернуться.

Кроме адреса, для каждого из узлов от текущего до корневого нам нужно ещё помнить, на чём мы с этим узлом остановились. Здесь возможны три варианта:

- `start`: мы только что узнали о существовании этого узла, внесли его в список (это произошло при рассмотрении его узла-предка), но пока что ничего с ним не делали, то есть нам остаётся ещё рассмотреть его левое поддерево, напечатать число из самого этого узла, потом рассмотреть правое поддерево;
- `left_visited`: мы уже рассмотрели левое поддерево и вернулись к текущему узлу, так что надо только напечатать число из него и рассмотреть правое поддерево;

- `completed`: мы вернулись к текущему узлу после рассмотрения его правого поддерева, так что нужно завершить его рассмотрение, т. е. убрать его из списка, чтобы текущим снова стал его предок.

В начале работы мы формируем список из одного узла — корневого, отметив, что для него ещё ничего не сделано. Дальше мы запускаем цикл, на каждой итерации которого смотрим, как обстоят дела с текущим узлом дерева. Если с ним ещё ничего не происходило, то у нас есть два варианта: если левое поддерево не пусто, добавляем в список узел, являющийся левым потомком текущего узла (он в результате будет текущим узлом на следующей итерации цикла); если же левое поддерево пусто, то рассматривать его не нужно и список мы не изменяем. В обоих случаях, прежде чем продолжить выполнение цикла, мы помечаем, что для текущего узла рассмотрение его левого поддерева состоялось.

Если для текущего узла его левое поддерево (неважно, пустое или нет) уже обработано, мы печатаем число из текущего узла, а дальше, как и для левого поддерева, либо делаем правого потомка текущим узлом, либо (если его нет) — не делаем, но в любом случае запоминаем, что с текущим узлом делать больше ничего не надо.

Наконец, если для текущего узла больше ничего делать не требуется, мы исключаем его из списка, так что текущим становится его предок. Если при этом текущим был корневой узел дерева, список опустеет; это обстоятельство мы используем в качестве условия цикла — продолжаем работу, пока список не пуст.

Читатель может заметить в тексте нашего примера небольшую оптимизацию: если левое или правое поддерево пусто, то спускаться в него нужды нет, как нет нужды и в лишней итерации цикла: можно сразу перейти к следующей стадии обработки того же самого текущего узла, что мы и делаем, не поставив `break` в конце соответствующей ветви оператора выбора. Если его там всё же поставить, работать функция будет точно так же, просто чуть-чуть медленнее — именно чуть-чуть, мы здесь не сможем заметить разницу в быстродействии. Концептуально решение без этой оптимизации, конечно, чище; автор вынужден признать, что во время работы над вторым томом не смог от этого хака удержаться: когда очевидно, что будет делаться переход на следующий оператор в теле `case`, делать этот переход через возврат в объемлющий цикл и повторное выполнение выбора ветки `case` кажется варварством, пусть даже это *почти* ничего не стоит.

Столь подробное описание решения с помощью цикла мы привели, чтобы подчеркнуть его отличие от короткого и ясного рекурсивного решения. Казалось бы, они не имеют между собой ничего общего; но не будем торопиться. Прежде всего заметим, что наш список узлов, выполняющий роль нити Ариадны при возвращении из глубин дерева, работает по принципу стека: узлы изымаются из него в порядке, обратном их занесению. Активные действия мы производим только с узлом, адрес которого находится в первом элементе списка — на вершине стека; при этом в каждом элементе стека мы храним две вещи: адрес текущего узла и то, на какой стадии находится его обработка.

Вернёмся теперь к рекурсивному решению. Очевидно, что узлы дерева мы здесь обходим строго в том же порядке — слева направо. Когда наша функция вызвана для определённого узла, этот узел следует считать текущим, его адрес хранится в параметре функции — в переменной `r`. Вспомнив материал первого тома (см. §3.3.6), мы можем заметить, что эта информация располагается *в стеке*, просто на этот раз стек используется аппаратный, тот, где хранятся адреса возврата и локальные переменные. Но что со второй частью информации, использовавшейся в итеративном алгоритме — с тем, на какой стадии находится обработка данного узла? В рекурсивном решении эта информация представлена *текущей позицией исполнения* тела функции. Иначе говоря, пока функция занята обработкой данного узла, стадия его обработки задаётся значением регистра счётчика команд (`EIP` в архитектуре, которую мы изучали в первом томе), когда же обработка узла откладывается, чтобы обойти его левое или правое поддерево, та же информация о стадии оказывается представлена *адресом возврата*, который, разумеется, *тоже заносится в стек*!

Итак, при рекурсивном обходе дерева у нас имеется стек из записей, каждая запись соответствует узлу дерева где-то на пути от корневого узла до текущего, запись в стеке создаётся, когда мы начинаем обход непустого поддерева, и ликвидируется, когда мы этот обход завершаем, в каждой записи хранятся адрес соответствующего узла дерева и информация о стадии обработки этого узла. Про итеративное решение можно сказать, очевидно, *всё то же самое*; совершенно неожиданно мы обнаруживаем, что рекурсивное и итеративное решения делают одно и то же.

Здесь, очевидно, можно возразить, что аппаратный стек — это совершенно не то же самое, что стек на базе односвязного списка. В самом деле, наше итеративное решение будет работать гораздо медленнее рекурсивного из-за операций с динамической памятью, но с этой проблемой можно частично справиться, реализовав стек не в виде списка, а в виде массива, который при необходимости увеличивается в размерах; после этого разница в эффективности между итеративным и рекурсивным обходом дерева станет не столь значительна. Впрочем, можно отметить и ещё один момент. При работе на Си или Паскале вызовы подпрограмм — фактически единственный интерфейс к аппаратному стеку, но ведь программировать можно и на языке ассемблера, и там нам никто не помешает воспользоваться аппаратным стеком для хранения обратного пути по узлам дерева при реализации итеративного алгоритма. При таком подходе работа рекурсивного решения будет отличаться от работы его итеративного «близнеца» лишь форматом представления информации о стадии обработки отдельного узла дерева.

Вернёмся чуть назад и посмотрим на словесные описания этих двух алгоритмов (одна фраза для рекурсивного, целая страница для итеративного) и текст их реализации (пять строк против сорока). Коль скоро в действительности речь идёт об одних и тех же действиях машины, в чем же состоит отличие, из-за которого и описания, и реализации по своей сложности различаются в несколько раз? Возьмём на себя смелость утверждать, что основное отличие тут в *восприятии происходящего человеком*, в том, если угодно, под каким углом зрения программист рассматривает решение поставленной задачи. **Парадигма программирования — не в компьютере, она в голове программиста**.

9.2.2. Виды рекурсии

В большинстве учебников в качестве простого примера рекурсии рассматривают вычисление факториала. Этот пример трудно назвать удачным, поскольку в действительности факториалы никогда не вычисляются во время выполнения программ. В самом деле, число 21! не помещается в 64-битное целое, то есть в большинстве языков программирования и сред выполнения не может быть представлено, так что и вычислений с участием этого числа производить нельзя; что касается значений факториала для чисел от 1 до 20, то их, очевидно, проще будет вычислить заранее и расположить в константном массиве, нежели считать каждый раз заново.

Если вам потребовалось численное значение факториала для числа, превышающего 10, то вы, как правило, делаете что-то не так. Например, в прикладных задачах сравнительно часто могут потребоваться биномиальные коэффициенты, формулу которых ($C_n^k = \frac{n!}{k!(n-k)!}$) мы выводили во вводной части первого тома (см. §1.3.1). Вычислять их «в лоб» по этой формуле — попросту глупо и в большинстве случаев для этого не хватит разрядности в ходе промежуточных вычислений; описанный в той же главе первого тома *треугольник Паскаля*, основанный на соотношении $C_{n+1}^k \equiv C_n^{k-1} + C_n^k$, позволяет обойтись для этого одними только сложениями, при этом промежуточные результаты никогда не превысят вычисляемого коэффициента. Например, с помощью треугольника Паскаля не составляет проблем вычислить $C_{30}^{12} = 86493225$, тогда как вычисление того же коэффициента по формуле с факториалами потребовало бы так называемой «длинной арифметики».

Так или иначе, численные значения факториалов — не то, что может реально потребоваться вычислять в прикладной задаче.

Отказавшись рекурсивно считать факториал, остановимся на другом простом примере: попробуем рекурсивно реализовать подсчёт суммы элементов целочисленного массива. В качестве базиса рекурсии выберем самый простой случай, когда вычисления вообще не нужны — массив, не содержащий ни одного элемента; сумма элементов такого массива, очевидно, равна нулю. Если же массив не пуст, то его сумма есть сумма массива, составленного из всех элементов исходного, кроме самого первого, к которой прибавлен первый элемент; сумму массива

можно вычислить с помощью самой функции, которая сейчас пишется. Получается примерно так:

```
int array_sum(const int arr[], int len)
{
    if(len <= 0)
        return 0;
    return arr[0] + array_sum(&(arr[1]), len - 1);
}
```

Конечно, «настоящие сишники» напишут то же самое иначе, примерно так:

```
int array_sum(const int *arr, int len)
{
    return len <= 0 ? 0 : *arr + array_sum(arr+1, len-1);
}
```

Впрочем, машинный код из этих двух вариантов получится, скорее всего, один и тот же; разница здесь, как водится, в восприятии. Основных отличий можно отметить два: использование разыменования вместо индексирования и условное выражение вместо условного оператора; если первое — это своего рода визитная карточка языка Си с его адресной арифметикой, то второе, как ни странно, превращает реализацию из императивной в функциональную.

Так или иначе, здесь мы имеем дело с ***простой рекурсией***: функция вызывает сама себя ровно один раз. Схематически это можно показать так:

```
f() { ... f() ...  }
```

Существуют и другие виды рекурсии, которые мы сейчас попытаемся рассмотреть. Если функция вызывает другую функцию, та, возможно, третью и так далее, но в какой-то момент снова вызывается первая, говорят о ***взаимной рекурсии***. С таким вариантом (правда, всего для двух функций) мы встречались в т. 2, когда решали на Си задачу о сопоставлении строки с образцом (см. §4.3.22). Схематически общий вид взаимной рекурсии выглядит так:

```
f1() { ... f2() ... }
f2() { ... f3() ... }
...
fn() { ... f1() ... }
```

а для простейшего случая из двух функций — так:

```
f() { ... g() ...  }
g() { ... f() ...  }
```

В ситуации, когда функция за одно выполнение может вызвать сама себя больше одного раза, но все эти вызовы выполняются так, что ни один из них не зависит от результатов работы другого, мы имеем дело с ***параллельной рекурсией***. Название оправдывается тем, что при наличии более чем одного процессора отдельные рекурсивные вызовы могут быть отработаны одновременно (параллельно). Схематически параллельная рекурсия выглядит так:

```
f() { ... g(... f(), ... f(), ... ) ... }
```

Здесь g — это не обязательно функция, это может быть, например, арифметическое выражение, для вычисления которого нужны все его операнды, но которому при этом всё равно, в каком порядке эти операнды вычислены. В частности, если бы нам нужно было обойти двоичное дерево не для распечатки его элементов, как мы это делали в предыдущем параграфе, а для вычисления его суммы, у нас получилось бы что-то вроде следующего:

```
int treesum(const struct node *r)
{
    return r ? treesum(r->left) + r->val + treesum(r->right) : 0;
}
```

В роли функции g здесь выступает обычное сложение.

Как отдельный вид рекурсии иногда рассматривают ***рекурсию высшего порядка***[7], когда функция может вызвать сама себя больше одного раза, причём один вызов зависит от результата другого вызова:

```
f() { ... f(... f() ...) ... }
```

В качестве примера такой рекурсии обычно приводят функцию Аккермана, которую мы уже упоминали в первом томе (см. §1.3.6). Определение её таково:

$$A(m,n) = \begin{cases} n+1, & m = 0; \\ A(m-1,1), & m > 0, n = 0; \\ A(m-1,A(m,n-1)), & m > 0, n > 0. \end{cases}$$

В теории вычислимости эта функция играет чрезвычайно важную роль — она служит простейшим примером частично-рекурсивной функции, которая при этом не является примитивно-рекурсивной; но, коль скоро мы занимаемся практическим программированием, никакой пользы от функции Аккермана мы, скорее всего, не получим. Даже если бы она нам вдруг понадобилась, то мы бы всё равно не стали её вычислять с помощью рекурсии, повторяющей определение. Дело в том, что для $m \in \{0,...,3\}$ существуют достаточно простые нерекурсивные формулы вычисления $A(m,n)$, а при $m > 3$ функция Аккермана растёт столь стремительно, что её значения всё равно никуда не поместятся, причём нас толком не спасёт даже длинная арифметика: для хранения числа $A(4,3) = 2^{2^{65536}} - 3$, если бы мы попытались честно представить его в виде битовой строки, не хватило бы физической памяти всех компьютеров мира, ну а $A(4,4) = 2^{2^{2^{65536}}} - 3$ содержит больше двоичных цифр, чем Вселенная — атомов. При $m > 4$ для записи значений функции Аккермана приходится прибегать к нетрадиционным математическим обозначениям вроде «стрелочной нотации Кнута», но это уже к практическому программированию никакого отношения не имеет. Автору книги не удалось придумать сколько-нибудь практичный, короткий и при этом убедительный пример рекурсии высших порядков.

9.2.3. Редукция (свёртка) последовательностей

Термины ***редукция*** и ***свёртка*** — если, конечно, рассматривать предметную область, связанную с обработкой последовательностей — представляют собой синонимы; надо сказать, что и в английской литературе для обозначения этого приёма используются два разных термина — *reduce* и *fold*. Суть редукции в том, что одна и та же функция (от двух аргументов) последовательно применяется к первому элементу последовательности, затем к полученному результату и второму элементу, затем к полученному результату и третьему элементу и так далее. На самом первом шаге, то есть когда у нас нет результата предыдущего шага, вместо него используется некое значение, называемое ***затравкой***. Например, если у нас есть последовательность $\{a_1, a_2, ..., a_n\}$, затравка v и функция двух аргументов f, то формально можно рассматривать редукцию как последовательность вычислений

$$\begin{aligned} r_1 &= f(a_1, v), \\ r_2 &= f(a_2, r_1), \\ r_3 &= f(a_3, r_2), \\ &... \\ r_n &= f(a_n, r_{n-1}), \end{aligned}$$

или, если положить $r_0 = v$, то:

$$r_n = \begin{cases} v, & n = 0; \\ f(a_n, r_{n-1}), & n > 0. \end{cases}$$

Последний полученный результат (в данном случае r_n) рассматривается как результат всей редукции. Например, если в роли f будет выступать простое сложение ($f(x,y) = x+y$), а в роли затравки — 0, редукция даст нам сумму последовательности; если же в качестве f взять умножение

[7] Англ. *higher-order recursion*. Как водится, здесь есть трудности с переводом: формально *higher* — это сравнительная степень, а не превосходная, так что прямой перевод звучал бы как «*рекурсия более высокого порядка*», но в русском языке применение сравнительной степени немедленно влечёт вопрос вроде «более высокого, чем что?», и отсутствие прямого ответа на этот вопрос в ближайшем контексте вызывает ощущение семантической ошибки. Перевод словом «высший» здесь представляет собой довольно неприятный компромисс, поскольку, разумеется, «порядок», если так можно выразиться, может быть «ещё выше».

($f(x,y) = xy$), а затравкой сделать единицу, мы получим произведение. Определив f как максимум своих аргументов ($f(x,y) = max(x,y)$), можно найти наибольший элемент последовательности, но придётся в качестве затравки взять её первый элемент.

Редукцию можно проводить не с начала, а с конца заданной последовательности:

$$\begin{aligned} r_n &= f(a_n, v), \\ r_{n-1} &= f(a_{n-1}, r_n), \\ &\dots \\ r_2 &= f(a_2, r_3), \\ r_1 &= f(a_1, r_2). \end{aligned}$$

Результатом здесь будет r_1. Первый вариант называют ***левой редукцией***, второй — ***правой редукцией***; название оправдывается тем, *с какого конца* последовательности её начинают обрабатывать. Вышеприведённые соотношения можно записать без использования промежуточных переменных, и обычно так и поступают, меняя для наглядности местами аргументы функции для случая левой редукции:

$$\begin{aligned} r_{\text{left}} &= f(f(\dots f(f(v, a_1), a_2) \dots, a_{n-1}), a_n) \\ r_{\text{right}} &= f(a_1, f(a_2, \dots, f(a_{n-1}, f(a_n, v)) \dots)) \end{aligned}$$

В языках функционального программирования обычно есть библиотечная функция, реализующая оба варианта редукции; функция f при этом передаётся как один из параметров. Примечательно, что в нетипизированных языках вроде Лиспа это может быть действительно *функция*, тогда как в языках типизированных для редукции используется нечто *обобщённое* в том смысле, что применять это «нечто» можно к разным типам элементов последовательности, разным типам возвращаемого значения для функции f и разным видам последовательностей (массив, список или даже файл). Между прочим, в общем случае тип элемента последовательности не совпадает с типом значения функции. Например, с помощью редукции можно удалить из списка элементы, удовлетворяющие определённому условию; в этом случае, каков бы ни был тип элемента списка, функция f должна будет возвращать *список*, а в роли затравки будет, очевидно, пустой список.

Интерес для нас представляет рекурсивная реализация редукции — собственно говоря, так её всегда и реализуют. Левую редукцию при желании можно реализовать простым циклом, но правая допускает это лишь для последовательностей, по которым можно двигаться справа налево — например, для массивов или двусвязных списков. Если нам придёт в голову реализовать циклом правую редукцию для односвязного списка, придётся завести вспомогательный список — фактически получится, что мы сначала построим «перевёрнутый» список элементов, а потом уже будем применять f к его элементам. Так или иначе, итеративная реализация редукции — как правой, так и левой — выглядит несколько *противоестественно*.

К сожалению, язык Си слишком беден, чтобы на нём можно было показать редукцию последовательностей во всём её великолепии; такую демонстрацию мы отложим до появления в нашем багаже более подходящих для этого инструментов. Сейчас же ограничимся рассмотрением списков целых чисел, состоящих из звеньев такого типа[8]:

```
struct item {
    int val;
    struct item *next;
};
```

и функций, производящих арифметические действия, то есть принимающих на вход два параметра типа `int` и возвращающих тоже `int`. Опишем тип для указателя на такую функцию:

```
typedef int (*intfunptr)(int, int);
```

Функция, производящая левую редукцию списка, будет выглядеть так:

```
int intlist_reduce_l(intfunptr f, int i, struct item *ls)
{
    return ls ? intlist_reduce_l(f, f(i, ls->val), ls->next) : i;
}
```

Правая редукция делается так:

```
int intlist_reduce_r(intfunptr f, int i, struct item *ls)
{
    return ls ? f(ls->val, intlist_reduce_r(f, i, ls->next)) : i;
}
```

Несмотря на схожесть решаемых задач, очевидное подобие решений и даже одинаковый базис рекурсии (при пустом списке результатом объявляется затравка), эти две функции работают совершенно по-разному. При левой редукции мы сначала обращаемся к функции f, получаем результат, и задача сводится к предыдущей, только в роли затравки теперь выступает этот только что полученный результат, а от списка у нас осталось на один элемент меньше; остаётся только обратиться к той функции, которую мы пишем, чтобы, как полагается при использовании рекурсии, решить ту же задачу для чуть-чуть более простого случая.

С правой редукцией всё несколько сложнее. Обратиться сразу же к функции f мы не можем, поскольку знаем только один из параметров для этого обращения — первый элемент списка; но вторым её параметром должен стать результат редукции всего остального списка, каковой наша реализация и вычисляет, рекурсивно вызвав сама себя, и лишь после этого вызывает f. Если вы запутались, попробуйте разобрать следующую запись тех же самых решений:

```
int intlist_reduce_l(intfunptr f, int i, struct item *ls)
{
    int new_i;
    if(!ls)
        return i;
    new_i = f(i, ls->val);
    return intlist_reduce_l(f, new_i, ls->next);
}

int intlist_reduce_r(intfunptr f, int i, struct item *ls)
{
    int rest_res;
    if(!ls)
        return i;
    rest_res = intlist_reduce_r(f, i, ls->next);
    return f(ls->val, rest_res);
}
```

Забегая вперёд, отметим, что `intlist_reduce_l` поддаётся оптимизации остаточной рекурсии, а `intlist_reduce_r` — нет, поскольку после рекурсивного вызова делает что-то ещё, в данном случае — вызывает функцию `f`.

Чтобы попробовать нашу редукцию в деле, нам потребуются функции, которые можно использовать в роли f. Напишем несколько таких функций:

```
int int_plus(int x, int y) { return x + y; }
int int_mul(int x, int y) { return x * y; }
int int_max(int x, int y) { return x > y ? x : y; }
int int_zcnt_left(int n, int x) { return x==0 ? n+1 : n; }
int int_zcnt_right(int x, int n) { return x==0 ? n+1 : n; }
```

Теперь, если у нас есть список, на который указывает указатель `list`, то посчитать сумму и произведение его элементов мы можем так:

```
sum = intlist_reduce_l(int_plus, 0, list);
prod = intlist_reduce_l(int_mul, 1, list);
```

Найти максимальный элемент в списке можно так:

```
max = intlist_reduce_l(int_max, list->val, list->next);
```

[8] Приведённые здесь и далее в этом параграфе фрагменты кода вы найдёте в архиве примеров в файле `reduce_ls.c`.

Для суммы и произведения можно с тем же успехом использовать `intlist_reduce_r`, для поиска максимума, в принципе, тоже, хотя поиск при этом будет выполнен в несколько странной последовательности: сначала первый элемент будет принят за максимальный, а потом список будет просмотрен с конца до второго элемента.

Функции `int_zcnt_left` и `int_zcnt_right` позволяют посчитать в списке количество элементов, равных нулю, причём первая предназначена для левой редукции, вторая — для правой. Параметр `x` у этих функций — очередное число, которое нужно сравнить с нулём, а параметр `n` — количество ранее обнаруженных нулей; если `x` равен нулю, функция возвращает на единицу больше, чем `n`, в противном случае — просто `n`. Две разные функции потребовались из-за того, что `intlist_reduce_l` передаёт элементы списка в функцию f *вторым* параметром, а `intlist_reduce_r` — *первым*. Соответствующие вызовы будут выглядеть так:

```
zero_count = intlist_reduce_l(int_zcnt_left, 0, list);
zero_count = intlist_reduce_r(int_zcnt_right, 0, list);
```

Отметим ещё один момент. Все примеры этого параграфа носят скорее иллюстративный, нежели практический характер. В действительности вряд ли кому-то придёт в голову применять редукцию в настоящих программах на Си. Например, та же сумма списка, реализованная через редукцию, будет работать раз в десять медленнее, чем простое суммирование элементов циклом, ну а такие случаи редукции, когда она реально полезна, на Си не сделать. Мы обязательно вернёмся к редукции позднее, при рассмотрении таких языков программирования, где она выглядит намного естественнее.

9.2.4. Остаточная рекурсия

Рассмотрим ещё один простой пример. Пусть у нас имеется список целых чисел, такой же, какой мы использовали ранее (звенья с полями `val` и `next`), и нам нужно найти первый элемент с заданным значением. Если элемент найден, следует вернуть его адрес, а если элемента с таким значением нет — вернуть `NULL`. Вполне естественным будет рекурсивное решение: если список пуст, вернуть `NULL`, если он не пуст и первый элемент содержит нужное нам значение — вернуть адрес этого элемента, в остальных случаях попытаться поискать элемент в хвосте списка. Функция будет выглядеть так:

```
const struct item *find_value(const struct item *lst, int val)
{
    if(!lst)
        return NULL;
    if(lst->val == val)
        return lst;
    return find_value(lst->next, val);
}
```

Здесь примечателен тот факт, что после завершения рекурсивного вызова значение, возвращённое вызванной («вложенной») ипостасью нашей функции, немедленно становится значением, возвращаемым из текущего вызова функции, а её выполнение на этом завершается. Иначе говоря, рекурсивный вызов — если, конечно, выполнение пошло именно по этой ветке — становится *последним* действием, выполняемым функцией (на данном уровне вложенности), больше ничего делать не надо. Как следствие, стековый фрейм, соответствующий текущему выполнению функции, становится *не нужен* в тот момент, когда управление передано вложенному экземпляру. Очевидная оптимизация состоит в том, чтобы стековый фрейм для рекурсивно вызванного прогона функции создавался не так, как это обычно делается — сверху в стеке над текущим фреймом — а *поверх* текущего фрейма, затирая его; рекурсивная ипостась нашей функции в этом случае должна вернуть управление сразу туда, откуда изначально была вызвана наша функция.

К объяснению можно подойти иначе. При рекурсивном вызове управление, очевидно, передаётся на начало той же самой нашей функции. Если вызовы реализованы «честно», то есть без оптимизации, то при такой передаче управления будет создан новый стековый фрейм, содержащий в качестве адреса возврата конец функции, и когда рекурсия дойдёт до одного из базовых случаев, за этим последует цепочка возвратов (без каких-либо ещё действий), так что управление в итоге вернётся к тому, кто нас вызвал — но не сразу, сначала все созданные стековые фреймы будут ликвидированы по одному. Очевидно, тех же результатов можно достичь, не создавая новые фреймы и не добавляя в стек новые адреса возврата: достаточно переписать содержимое существующего стекового фрейма, оставив адрес возврата как есть. Проиллюстрируем сказанное, переписав нашу функцию с применением запрещённого приёма — `goto` назад:

```
const struct item *find_value(const struct item *lst, int val)
{
begin:
    if(!lst)
        return NULL;
    if(lst->val == val)
        return lst;
    lst = lst->next;
    goto begin;
}
```

На всякий случай подчеркнём: **писать так не надо!**

Естественно, то же самое можно сделать и без `goto`, просто чуть менее наглядно — здесь уже труднее заметить, что решение получено путём оптимизации рекурсивного варианта:

```
const struct item *find_value(const struct item *lst, int val)
{
    for(;;) {
        if(!lst)
            return NULL;
        if(lst->val == val)
            return lst;
        lst = lst->next;
    }
}
```

Такой вариант оптимизации с переходом на начало вместо рекурсивного вызова возможен только для случая простой рекурсии, когда функция непосредственно вызывает сама себя; однако ситуация с холостым расходованием стека возможна и в более сложных случаях — например, при взаимной рекурсии.

Проиллюстрируем это уже знакомым нам примером сопоставления строки с образцом, в котором символ «?» соответствует любому символу строки, символ «*» сопоставляется с произвольной *цепочкой* (в том числе пустой), а остальные символы обозначают сами себя и только с самими собой сопоставляются. За основу возьмём решение на Си, состоящее из двух функций (см. т. 2, §4.3.22), и слегка модифицируем его; в частности, избавимся от цикла, заменив его рекурсией. Получится у нас вот что:

```
/* match_rec.c */
int starmatch(const char *str, const char *pat);
int match(const char *str, const char *pat)
{
    switch(*pat) {
        case 0:
            return *str == 0;
        case '*':
            return starmatch(str, pat+1);
        default:
            if(!*str || (*str != *pat && *pat != '?'))
                return 0;
            return match(str+1, pat+1);
    }
}
int starmatch(const char *str, const char *pat)
{
    if(match(str, pat))
        return 1;
    if(!*str)
        return 0;
    return starmatch(str+1, pat);
}
```

Основная рекурсия здесь (в функции `match`) строится по убыванию образца — от него по мере обработки отщепляются символы слева. Базисом служит тривиальный случай пустого образца: такой образец успешно сопоставляется только с пустой строкой, если же строка не пуста — фиксируется неуспех, но в обоих случаях более никаких действий не требуется.

Когда остаток образца не пуст и его первый символ — не звёздочка, следует сначала проверить, не станет ли этот первый символ причиной неудачи сопоставления: это произойдёт, если остаток сопоставляемой строки пуст, а также если символ образца не совпадает с символом строки и при этом не является знаком вопроса. Если неудачи на текущем шаге не произошло, остаётся, отбросив первые (только что обработанные) символы, сопоставить остаток строки с остатком образца, а сделать это можно с помощью той же функции, которую мы пишем (`match`), т. е. выполнив рекурсивный вызов.

Наиболее интересен случай, когда первым символом остатка образца оказалась звёздочка. Этот случай обрабатывается функцией `starmatch`, которой передаётся вся оставшаяся для сопоставления строка, а также остаток образца — но уже без текущей звёздочки.

Со звёздочкой можно сопоставить подстроку произвольной длины, в том числе и пустую; именно с этого случая начинает рассмотрение ситуации функция `starmatch` — в предположении, что текущая звёздочка соответствует пустой подстроке, ей остаётся проверить, сопоставится ли при этом остаток строки с остатком образца, а эта задача, естественно, решается с помощью вызова функции `match`; поскольку ранее сама функция `starmatch` была вызвана из `match`, здесь имеет место косвенная рекурсия.

Если сопоставление остатков прошло успешно, можно фиксировать успех всего сопоставления целиком; но если сопоставление не удалось, это само по себе означает лишь, что текущей звёздочке не получилось поставить в соответствие пустую строку. Но коль скоро в сопоставляемой строке есть ещё символы, можно попробовать текущей звёздочке поставить в соответствие на один символ больше и посмотреть, не пройдёт ли сопоставление в таком виде — и продолжать это, пока в строке не исчерпаются символы. Функция `starmatch` организует этот процесс проб и ошибок рекурсивно, вызывая сама себя с тем же остатком образца и с укороченной на первый символ строкой; перед тем, как укорачивать строку, нужно удостовериться, что её есть куда укорачивать, и случай, когда строка опустела и укорачивать её дальше уже нельзя, служит базисом рекурсии: в этом случае приходится признать сопоставление неуспешным.

Сделаем очень важное замечание. Стековые фреймы в нашем решении служат для хранения развилок, образующихся при рассмотрении звёздочек в образце: здесь нужно попробовать сопоставить остатки, как если бы звёздочка соответствовала пустой строке, если же это не получилось — вернуться к развилке, отбросить первый символ строки и попробовать снова. Если говорить совсем строго, стековый фрейм функции `starmatch` нужен, чтобы помнить состояние рассмотрения очередной звёздочки. Но во всех остальных ситуациях после того, как любая из двух наших функций вызывает сама себя или вторую функцию, её стековый фрейм становится не нужен, и это видно по их коду: все вызовы, кроме одного, производятся непосредственно из оператора `return`, причём возвращается именно то значение, которое вернула вызываемая функция.

Интересно, что фрейм, хранящий развилку, становится не нужен, когда начато рассмотрение последнего возможного для данной развилки случая — а именно, когда укорачивать строку уже некуда, поскольку она опустела. Вышеприведённое решение этого не учитывает, но это можно исправить, слегка переписав функцию `starmatch`:

```
int starmatch(const char *str, const char *pat)
{
    if(!*str)
        return match("", pat);
    if(match(str, pat))
        return 1;
    return starmatch(str+1, pat);
}
```

Вызовов `match` здесь стало два, причём первый, соответствующий базисному случаю пустой строки, теперь будет ***остаточным*** — то есть таким, при котором стековый фрейм вызывающей функции больше ничего полезного в себе не содержит.

Компиляторы Си обычно не могут оптимизировать остаточную рекурсию, либо как максимум ограничиваются тривиальным случаем вызова функцией самой себя с новыми значениями параметров, который эквивалентен переходу на начало тела функции (см. стр. 21). Взаимная рекурсия, такая, как в нашем случае, компиляторам Си не по зубам. Однако если бы компиляторы Си оптимизировали каждый остаточный вызов, в буквальном смысле ликвидируя стековый фрейм текущей функции перед передачей управления вызываемой (что технически вполне возможно), то наше сопоставление использовало бы столько стековых фреймов, сколько в каждый момент времени есть оставшихся позади звёздочек, для которых ещё есть нерассмотренные возможности. В частности, сопоставление с образцом, не содержащим ни одной звёздочки, проходило бы вообще без расходования стека.

На всякий случай предостережём читателя от чрезмерного оптимизма по поводу оптимизации остаточной рекурсии и вообще остаточных вызовов. Компиляторы Си не делают такой оптимизации не потому, что их создатели слишком ленивы или глупы; напротив, современные оптимизаторы кода, включаемые в состав компиляторов Си, представляют собой одну из самых интеллектоёмких областей программирования.

Реальная причина тут в том, что оптимизация остаточных вызовов снизила бы быстродействие получаемого машинного кода. В самом деле, значения параметров для вызываемой функции пришлось бы *в текущем фрейме* выстроить так, как этого ожидает вызываемая функция, и это при том, что значения этих (новых) параметров могут зависеть от значений, хранящихся в локальных переменных, т. е. в самом фрейме. Выстроить последовательность вычислений новых параметров так, чтобы нигде не была затёрта локальная переменная, которая всё ещё нужна — задача нетривиальная, особенно если пытаться решать её в общем виде, ну а тривиальное решение — сначала всё вычислить, расположить как надо (в свободной области фрейма), и уже потом скопировать в начало фрейма поверх существующих переменных, которые теперь-то уж точно не нужны — очевидно приводит к лишнему копированию.

Создатели оптимизаторов обычно ставят перед собой задачу противоположную: повысить быстродействие даже ценой увеличения расхода памяти. Кроме того, Си допускает использование рекурсии, но к числу основных инструментов она здесь не относится; в этой части книги мы используем Си для иллюстраций, просто чтобы не рассказывать для этих целей какой-то ещё язык.

Важно понимать, что вызов функции должен быть именно *самым последним, что нужно сделать в теле функции*, иначе он уже не сможет быть остаточным. Начинающие часто путаются с этим условием, поэтому уместно будет привести для иллюстрации пример, когда оптимизация остаточной рекурсии невозможна. Напишем функцию, определяющую длину списка:

```
int list_length(const struct item *lst)
{
    if(!lst)
        return 0;
    return list_length(lst->next) + 1;
}
```

или, что то же самое,

```
int list_length(const struct item *lst)
{
    return lst ? list_length(lst->next) + 1 : 0;
}
```

Здесь после рекурсивного вызова, прежде чем вернуть значение, функция выполняет ещё одно действие: прибавляет единицу к тому, что ей вернула её «рекурсивная ипостась». Это прибавление единицы происходит, естественно, в стековом фрейме вызывающего экземпляра, то есть фрейм ещё потребуется после возврата из рекурсии — и, как следствие, его нельзя уничтожить раньше, чем рекурсивный вызов завершит свою работу. Можно посмотреть на это и с другой точки зрения: результатом текущего прогона функции будет не то число, которое вернёт рекурсивный вызов, так что произвести возврат управления сразу нашему

вызывающему нельзя — сначала мы должны завершить свою работу, прибавив единицу.

Интересно, что во многих подобных случаях можно преобразовать рекурсию в остаточную, применив ***накапливающий параметр***[9]. Перепишем нашу функцию, снабдив её вторым параметром, который будет хранить *длину, накопленную к настоящему моменту*; в таком виде функция, делая всю фактическую работу, будет играть вспомогательную роль, поэтому мы переименуем её в `list_length_do`, а функция `list_length` будет её вызывать:

```
int list_length_do(const struct item *lst, int count)
{
    if(!lst)
        return count;
    return list_length_do(lst->next, count+1);
}
int list_length(const struct item *lst)
{
    return list_length_do(lst, 0);
}
```

Аналогичным образом можно переписать функцию, подсчитывающую сумму элементов массива (см. стр. 19):

```
int array_sum_do(int *arr, int len, int sum)
{
    if(len <= 0)
        return sum;
    return array_sum_do(arr+1, len-1, *arr + sum);
}
int array_sum(int *arr, int len)
{
    return array_sum_do(arr, len, 0);
}
```

Область применения техники накапливающего параметра не ограничивается приведением рекурсии к остаточному виду. Пусть, например, нам нужно развернуть односвязный список, расположив его элементы в обратном порядке. С помощью накапливающего параметра эта задача решается совершенно элементарно — достаточно на каждом уровне рекурсии отщеплять от исходного списка по одному элементу и добавлять отщеплённые элементы в начало списка, пошагово формирующегося во втором параметре. Создаваемый список, естественно, изначально следует сделать пустым:

```
struct item *reverse_list_do(struct item *lst, struct item *res)
{
    struct item *list_rest;
    if(!lst)
        return res;
    list_rest = lst->next;
    lst->next = res;
    return reverse_list_do(list_rest, lst);
}
struct item *reverse_list(struct item *lst)
{
    return reverse_list_do(lst, NULL);
}
```

Попытаться решить эту задачу без накапливающего параметра предложим читателю в качестве пищи для размышления; учтите только, что «очевидное» решение — «перевернуть» хвост списка рекурсивным вызовом, а первый элемент прибавить в конец — никуда не годится, поскольку потребует количества действий, пропорционального *квадрату* длины списка (ведь добавление элемента в конец потребует просмотреть весь список, и так каждый раз), тогда как вышеприведённое решение работает за линейное время.

С другой стороны, конечно же, накапливающие параметры — не панацея; отнюдь не любую рекурсию можно привести к остаточному виду, и особенно наглядно это видно на примере параллельной рекурсии. Применяя накапливающий параметр при уже знакомом нам вычислении суммы элементов двоичного дерева, мы получим что-то вроде следующего:

```
int tree_sum_do(const struct node *r, int sum)
{
    if(!r)
        return sum;
    sum = tree_sum_do(r->left, sum);
    return tree_sum_do(r->right, sum + r->val);
}
```

Формально говоря, здесь даже имеет место остаточная рекурсия — но только для одного из двух рекурсивных вызовов. Если задаться целью довести это преобразование до классической остаточной рекурсии *любой ценой*, то в итоге мы, конечно, сможем это проделать — по тем же причинам, по которым любую рекурсию можно «развернуть» в цикл; но для этого нам потребуется дополнительный параметр, который будет помнить все узлы дерева, откуда мы ещё не пробовали ходить вправо, и для их хранения потребуется в простейшем случае список, в более сложном (например, ради повышения эффективности) — массив, снабжённый информацией о его размере и количестве задействованных элементов. Следует обратить внимание, что при попытке развернуть обход дерева в цикл мы тоже встречались с чем-то похожим, только там был список узлов для возвращения к корню. Похожи друг на друга и причины возникновения такой вспомогательной структуры данных: в простом рекурсивном решении соответствующая информация размещается в стековых фреймах, но если мы применяем цикл, то никаких фреймов создавать не можем, и точно так же, если мы хотим сделать рекурсию «чисто остаточной», мы вынуждены позаботиться, чтобы во фреймах не оставалось ничего полезного.

9.2.5. Рекурсивное мышление

> The determined Real Programmer can write FORTRAN programs in any language.
>
> *Ed Post*

> Можно писать на Фортране, главное — не думать на Фортране.
>
> *неизвестный автор*

Поскольку сейчас мы рассматриваем рекурсию в контексте разговора о парадигмах, уместно будет отметить, что рекурсию как приём программирования можно использовать, продолжая при этом *думать* так же, как и без рекурсии — в традиционных терминах действий и их последовательностей; между тем рекурсия представляет собой самостоятельную парадигму программирования, то есть предполагает свой собственный стиль мышления, и этим стилем чрезвычайно полезно владеть. Иначе говоря, мало освоить рекурсию как приём программирования — нужно научиться думать в соответствии с ней.

Для иллюстрации этого (возможно, на первый взгляд странного) утверждения рассмотрим ещё раз нашу простенькую функцию подсчёта длины списка:

```
int list_length(const struct item *lst)
{
    if(!lst)
        return 0;
    return list_length(lst->next) + 1;
}
```

Работу этой функции можно описать так:

> *Если список пуст, считаем его длину равной нулю. В противном случае сначала вычислим длину хвоста списка, применив для этого рекурсивный вызов, а затем прибавим единицу, и дело сделано.*

[9] В англоязычной литературе применяется термин *accumulator*, но, в отличие от некоторых других случаев, русским словом «аккумулятор» это обычно не переводят.

Здесь, собственно говоря, мы и наблюдаем императивное мышление в применении к рекурсии. Попробуем сравнить это описание с другим:

> *Длина пустого списка равна нулю, а длина любого другого списка на единицу больше, чем длина его хвоста.*

Если вам кажется, что из второго нужно обязательно сначала сделать первое, превратив некий «принцип» в «алгоритм», а потом уже то, что получится, реализовывать — значит, рекурсивное мышление вам пока что не покорилось. В действительности второе описание — это тоже описание алгоритма, причём *конструктивное*, то есть непосредственно пригодное для реализации, и никакого предварительного перевода в другую форму для него не требуется.

Ещё более интересным примером рекурсивного мышления в противовес императивному осмыслению рекурсии могут служить формулы, которыми мы поясняли свёртку списков в §9.2.3. Приведённые там соотношения

$$r_{\text{left}} = f(f(...f(f(v, a_1), a_2)..., a_{n-1}), a_n)$$
$$r_{\text{right}} = f(a_1, f(a_2, ..., f(a_{n-1}, f(a_n, v))...))$$

соответствуют рекурсивному мышлению и, более того, в действительности именно они демонстрируют подлинную суть операции свёртки. Приведённые чуть раньше формулы для последовательного вычисления r_1, r_2 и т. д. (см. стр. 19) отражают императивное восприятие свёртки; большинству читателей они могут показаться яснее (поэтому мы их и привели), но причина этой «ясности» на самом деле лишь в уже успевших сформироваться императивных привычках.

К сожалению, нам сразу же придётся признать, что рекурсивное мышление при работе с рекурсией годится не всегда (увы, это так, несмотря на всю кажущуюся нелепость такого утверждения). Так, накапливающие параметры, рассмотренные в предыдущем параграфе, в рекурсивное мышление не укладываются — любые попытки дать «рекурсивное» описание решения с накопителем будут выглядеть вымученными и непонятными.

Ещё хуже обстоят дела с ***прямым и обратным ходом рекурсии***. Впервые столкнувшись с рекурсией ещё в первом томе (см. §2.3.7), мы применяли прямой и обратный ход, чтобы расставить в нужном порядке десятичные цифры заданного числа. Приведём ещё один пример — реализуем рекурсивно создание копии заданной строки в динамической памяти:

```
/* strdup_rec.c */
char *strdup_rec_do(const char *str, int depth)
{
    char *res;
    if(*str)
        res = strdup_rec_do(str+1, depth+1);
    else
        res = malloc(depth+1);
    res[depth] = *str;
    return res;
}
char *strdup_rec(const char *str)
{
    return strdup_rec_do(str, 0);
}
```

Вспомогательную функцию можно реализовать и короче:

```
char *strdup_rec_do(const char *str, int depth)
{
    char *res =
        *str ? strdup_rec_do(str+1, depth+1) : malloc(depth+1);
    res[depth] = *str;
    return res;
}
```

Работает всё это так. Сначала мы, погружаясь во вложенные вызовы, проходим строку слева направо, чтобы определить её длину; при этом каждая «ипостась» функции `strdup_rec_do` помнит свою позицию в строке — на эту позицию указывает параметр `str`. Кроме того, каждый вложенный вызов функции знает и позицию своего символа в строке, которая станет копией — эта позиция равна глубине вложенности вызовов, хранящейся в параметре `depth`. Когда конец строки достигнут, срабатывает ветка, служащая базисом рекурсии: под копию строки выделяется нужное количество памяти. Адрес выделенной памяти (то есть, в итоге, адрес создаваемой копии, ради которой всё и затевалось) возвращается в качестве значения функции; он так и будет возвращаться из каждого рекурсивного вызова, пока они не кончатся, но прежде чем вернуть управление, наша функция на каждом из своих уровней вложенности заносит в свою позицию копии строки соответствующий (свой) символ из оригинальной строки.

Как видим, описание получилось насквозь императивным: мы подробно перечислили, что, как и в каком порядке делается в ходе работы. Проблема в том, что иначе и не получится: *последовательность действий здесь важна* — во всяком случае, копирование символов обязано происходить *после* выделения памяти. В каком порядке будут скопированы сами символы, в принципе, неважно, но это лишь особенность данной конкретной задачи; бывают и такие задачи, в которых некая последовательность действий должна произойти строго зеркально по отношению к другой последовательности, и это достигается использованием обратного хода рекурсии (вышеупомянутый пример из первого тома прекрасно этот момент иллюстрирует).

Следует понимать, что это не хорошо и не плохо — это просто реальность: при применении рекурсии как программистского приёма рекурсивное мышление в одних случаях банально не работает, но в других оказывается крайне полезно, так что его желательно освоить, а заодно научиться различать ситуации, когда оно нужно и когда — не очень.

Сказанное позволяет отделить сущность, обозначаемую словом «парадигма», от всевозможных «приёмов», «методов», «техник», «подходов» и т. п. Несомненно, рекурсия — это прежде всего технический приём или, скорее, целый ряд таких приёмов, объединённых тем, что подпрограмма должна (прямо или косвенно) вызвать сама себя. **Рекурсия становится *парадигмой* лишь в случае, если мы применяем рекурсивное мышление**, что, впрочем, никоим образом не обесценивает рекурсию как сугубо технический навык — в этом проявлении она тоже нужна и полезна.

9.3. Парадигмы и языки программирования

9.3.1. О роли языка программирования

Многие авторы отмечают ключевую роль языка программирования в формировании мышления программиста. Так, Дейв Баррон в своём «Введении в языки программирования» [6] утверждает:

> ... Язык программирования — нечто большее, чем просто средство описания алгоритмов: он несёт в себе систему понятий, на основе которых человек может обдумывать свои задачи, и нотацию, с помощью которой он может выразить свои соображения по поводу решения задачи.

Эдсгер Дейкстра в своей лекции лауреата премии Тьюринга, озаглавленной «Смиренный программист» [3], делает ещё более категоричное заявление:

> Инструменты, которые мы пытаемся использовать, а также язык или обозначения, применяемые нами для выражения или записи наших мыслей, являются главными факторами, определяющими нашу способность хоть что-то думать и выражать!

Определяющую роль выбора языка программирования прекрасно иллюстрирует пример, приведённый Тимоти Баддом в книге [7]. В этом примере два программиста решают задачу, связанную с анализом цепочек ДНК: в достаточно большом массиве[10] целых чисел нужно выделить повторяющиеся последовательности, имеющие длину не менее заданной (небольшой, пять или шесть элементов). Первый программист (в примере Бадда) решает задачу на Фортране, известном своей высокой эффективностью, но делает это «в лоб», с помощью трёх вложенных циклов: первый цикл просматривает массив от начала до конца, второй просматривает тот же массив от текущей позиции первого цикла до конца массива, третий последовательно сравнивает элементы массива, начиная от текущих позиций первого и второго циклов. Фортран мы не проходили, а на Си то же самое выглядело бы примерно так (предполагается, что `arr` — массив, `len` — длина массива, `min` — наименьшая длина последовательности, способная нас заинтересовать):

```
int i, j, k, found;
for(i=0; i<len-min-1; i++) {
    for(j=i+1; j<len-min; j++) {
        found = 1;
        for(k=0; k<min; k++) {
            if(arr[i+k] != arr[j+k]) {
                found = 0;
                break;
            }
        }
        if(found) {
            /* !!! здесь фиксируется совпадение
               цепочек, начиная с позиций i и j */
        }
    }
}
```

Естественно, при достаточно больших величинах `len` такая программа за приемлемое время ничего не сделает.

Второй программист в примере Бадда работает на языке APL, который, в отличие от Фортрана, предполагает интерпретируемое исполнение и, как следствие, заведомо менее эффективен. Однако в этом языке есть одна важная возможность: встроенные в язык средства позволяют *отсортировать* объекты произвольной природы по произвольному критерию. Согласно Бадду, сам факт наличия этой возможности наводит APL-программиста на правильную мысль[11]. Для решения задачи достаточно создать матрицу (двумерный массив) размерности `(len-min+1)`×`(min+1)` (`len-min+1` строк, `min+1` элементов в каждой строке), в строку с номером `0` занести элементы исходного массива с индексами с `0` по `min-1`, в следующую строку — элементы с `1` по `min`, и так далее, т. е. каждая строка с индексом `k` будет содержать элементы исходного массива, имеющие индексы от `k` до `k+min-1`; при этом задействуются все столбцы матрицы, кроме последнего. В последний столбец каждой строки заносится номер (индекс) самой этой строки. После этого матрица сортируется по строкам, что, как известно, может быть сделано за $c \cdot N \cdot \log N$ действий, где N — длина массива, а c — некий постоянный коэффициент. Далее достаточно *за один проход* выяснить, есть ли в полученной матрице строки, в которых все элементы, кроме последнего (хранящего исходную позицию каждой строки), совпадают. Если такие строки обнаружены, то из их последних элементов можно взять позиции исходного массива, с которых начинаются интересующие нас цепочки.

Несмотря на то, что APL является языком интерпретируемым, такое решение будет работать *на несколько порядков быстрее* и завершится за приемлемое время даже для исходного массива, содержащего десятки миллионов элементов. Бадд заостряет внимание читателя на том, как наличие частной встроенной возможности (в данном случае — сортировки массивов произвольной природы) может натолкнуть программиста на более изящное решение проблемы. Важно понимать, что устроить сортировку матрицы по строкам за время $O(N \log N)$ можно, разумеется, и на Фортране, но в этом языке нет *встроенной* возможности для такой сортировки, и поэтому первый программист правильное решение проглядел.

С выводами Бадда можно не согласиться (предложим читателю отыскать контраргументы самостоятельно), но в целом влияние используемого языка программирования на мышление программиста несомненно. Больше того, каждый активно используемый язык со временем обрастает определёнными традициями и создаёт специфическую субкультуру, а в некоторых случаях — и не одну. Если парадигма императивного программирования развилась под влиянием архитектуры фон Неймана, то все остальные существующие парадигмы своим возникновением обязаны языкам программирования.

Одно время в Википедии в статьях, посвящённых языкам программирования, пытались указывать «парадигму», к которой данный язык якобы относится. Как водится, живое явление, каким являются языки программирования, не пожелало вписываться в рамки упрощённой классификации: постепенно более «продвинутые» авторы статей начали отмечать, что тот или иной язык позволяет использовать больше одной парадигмы, такие парадигмы начали пытаться перечислить, называя при этом сам язык «мультипарадигмальным». В итоге «мультипарадигмальными» оказались практически все языки. В действительности вопрос, к какой парадигме относится тот или иной язык, лишён смысла: языки не *относятся* к парадигмам, языки *формируют* парадигмы, причём каждый язык может способствовать становлению целого ряда парадигм, а едва ли не каждая парадигма обязана своим существованием нескольким (иногда очень разным) языкам.

Более интересным оказывается вопрос о том, *в каком соотношении* может находиться конкретный язык программирования с конкретной парадигмой. Очевидно, этот вопрос не праздный: даже если рассматривать всё ту же рекурсию, можно найти языки, в которых без неё не обойтись, и языки, в которых применение рекурсии невозможно (к счастью, таких языков почти не осталось — но игнорировать их существование нельзя). Мы попытаемся выделить семь возможных «уровней совместимости» между языком и парадигмой. Итак, язык программирования может:

- **навязывать** применение парадигмы, делая программирование без этой парадигмы невозможным;
- **понуждать** к применению парадигмы — об этом следует говорить, если программирование на данном языке без применения данной парадигмы возможно, но крайне неудобно;
- **поощрять** применение парадигмы: программирование без неё возможно и даже удобно, но если всё-таки начать её применять, это приведёт к ощутимому выигрышу (в возможностях, в трудоёмкости, в эффективности и т. п.);
- **поддерживать** парадигму; в язык могут быть включены специальные средства, поддерживающие работу с использованием конкретной парадигмы для тех, кто желает её использовать, но при этом может не возникать никакой ощутимой выгоды от её использования, а в некоторых случаях даже могут появляться лишние трудности;
- **допускать** парадигму в том смысле, что никакой специальной поддержки для неё в язык не включено, но если программист точно знает, чего хочет, то он всё-таки сможет эту парадигму применить, не испытывая существенных неудобств;
- **препятствовать** применению парадигмы; в этом случае программист, привыкший к конкретной парадигме, возможно, и сможет её применить в работе на данном языке, но это будет сопряжено с неудобствами, которые, скорее всего, заставят рано или поздно от данной парадигмы отказаться;
- **запрещать** парадигму, делая её использование полностью невозможным.

[10] В те времена, к которым, по-видимому, относится рассказанная Баддом история, под «достаточно большим массивом» в этом примере можно было подразумевать массив, содержащий несколько десятков тысяч строк. Современные компьютеры работают намного быстрее, но если массив окажется длиной в миллион элементов, «лобовое» решение будет неприемлемо медленным — дождаться результата можно будет разве что если его запустить на суперкомпьютере, предварительно распараллелив.

[11] Здесь мы немного отступим от оригинального решения, описанного в книге Бадда, поскольку оно не совсем верно.

Частично забегая вперёд, а в некоторых случаях ссылаясь на материал, который в нашу книгу вообще не планируется включать, мы сможем привести конкретные примеры для всех семи перечисленных уровней. Так, SmallTalk очевидным образом навязывает объектно-ориентированное программирование, а, скажем, Фортран навязывает присваивания; Лисп и его родственники понуждают к использованию рекурсии, а Паскаль — к использованию присваивания (да, на Паскале можно писать без присваиваний, но большинству людей такое и в голову не придёт).

Язык Си++, который мы рассмотрим в следующей части, поощряет использование всех возможностей, которые отличают его от чистого Си, а это и абстрактные типы данных, и объектно-ориентированное программирование, и обобщённое программирование (шаблоны), и обработка исключений. Каждая из перечисленных возможностей требует особого стиля мышления и, как следствие, является парадигмой, и без любой из них на Си++ можно работать — даже без всех сразу (это практически превратит Си++ в чистый Си, но ведь на Си *можно* программировать). При этом освоение любой такой возможности и присущего ей стиля мышления ощутимо повышает комфорт и снижает трудоёмкость работы. На эту тему можно привести пример и из совсем другой области — функционального программирования. Если рассматривать функции высоких порядков в сочетании с безымянными функциями («лямбдами») как отдельную парадигму (что в целом правильно), то, например, Лисп эту парадигму прекрасно поддерживает и её использование делает жизнь программиста приятнее, но начинающие обычно подолгу «созревают», прежде чем начать использовать тот же `mapcar`.

Лисп включает средства для организации циклов, Си++ включает поддержку для макросов, чистый Си поддерживает рекурсию (в качестве такой поддержки следует рассматривать локализацию переменных в функциях и ***повторновходимость*** (***реентерабельность***) функций, достигаемую без усилий со стороны программиста). Во всех трёх случаях, несмотря на наличие поддержки, другие особенности языка способствуют тому, чтобы данная парадигма не использовалась; особенно это заметно для случая циклов в Лиспе.

Практически любой язык, в котором есть функции и локальные переменные, допускает использование парадигмы функционального программирования; выше мы обсуждали, как можно было бы написать функциональную программу на Си, и вполне очевидно, что на Паскале это тоже возможно, но требуется точно знать, чего мы хотим, и приложить к этому определённые усилия. Впрочем, этот пример не слишком жизненный; но для варианта «язык допускает использование парадигмы» можно привести пример более чем практический. Как известно, ядро ОС Linux написано на чистом Си, в котором нет и не предвидится поддержки для объектно-ориентированного программирования. Несмотря на это, многие подсистемы ядра используют объектно-ориентированную архитектуру; особенно явно объектно-ориентированные свойства проявляет виртуальная файловая система (VFS). В роли объектов выступают структуры, в роли методов — функции, получающие указатель на такую структуру своим первым параметром; вместо наследования общая структура (предок) включается в частную в качестве поля; таблицы виртуальных функций создаются в виде структур, все поля которых представляют собой указатели на функции. Интересно, что эти поля даже в официальной документации называют методами[12].

Примером, когда язык препятствует применению парадигмы, можно считать попытки применения обобщённого программирования на чистом Си. Несомненно, можно написать шаблон для функции, используя многострочные макросы, и *инстанциировать* этот шаблон, применив макровызов, но каждый, кто хотя бы раз пробовал так поступить, знает, насколько это на самом деле неудобно и противно. Второй пример на эту тему тоже будет связан с чистым Си. Используя библиотечные функции `setjmp` и `longjmp`[13], можно при некотором старании организовать *обработку исключений*. Если вы знаете, о чём идёт речь, и при этом вам кажется, что это просто — попробуйте.

Наконец, простейшим примером полного запрета на парадигму могут служить языки, в которых нет подпрограмм с локальными переменными — всё те же Фокал и старый Бейсик. Очевидно, что заниматься функциональным программированием на этих языках не получится; более того, не выйдет даже организовать рекурсию. Автор этих строк привык считать, что другим примером полной несовместимости языка и парадигмы выступает пара из ООП и Bourne Shell (язык традиционного интерпретатора командной строки Unix), но как-то раз встретил людей, утверждающих, что при большом желании это возможно. К сожалению, автор так и не понял, как именно они собирались это делать, и продолжает подозревать, что ничего не получится.

9.3.2. Концептуальное отличие Си от Паскаля

Ранее мы неоднократно подчёркивали, что между Паскалем и Си больше сходства, чем различий, и во многом это действительно так. В обоих языках есть переменные и типы, присваивания, составной оператор, операторы ветвления и циклов, причём в обоих языках циклов три — с предусловием, с постусловием и `for` (правда, в Паскале это честный арифметический цикл, тогда как в Си — универсальная конструкция, об исходном предназначении которой часто забывают); в обоих языках есть подпрограммы, в обоих языках есть указатели и т. д. Конечно, есть и различия, но здесь следует понимать, что достойно внимания, а что — нет. Так, замена ключевых слов `begin` и `end` фигурными скобками, знака `:=`, обозначающего присваивание, обычным знаком равенства, а слова `and` символом `&&` по существу не меняет *вообще ничего* — составной оператор остаётся составным оператором, операторные скобки — операторными скобками, присваивание — присваиванием, операция логического «и» имеет тот же смысл — и не всё ли равно, какими закорючками всё это обозначается! К смене обозначений человек полностью адаптируется часа за полтора — и перестаёт их замечать.

Можно назвать целый ряд более серьёзных отличий Си от Паскаля: отсутствие в Си понятия «главной программы» и, как следствие, равноправие единиц трансляции, возможность описывать локальные переменные в любом составном операторе, а не только в отдельных подпрограммах, отсутствие различия между символом и его кодом и прочее в таком духе. Отсутствие в Си иных способов передачи параметров, нежели передача по значению, может оказать довольно чувствительное влияние на восприятие. Впрочем, и в Си есть немало такого, чего нет в Паскале — взять хотя бы адресную арифметику. Пусть даже взамен приходится пожертвовать массивами как отдельными сущностями, зато в Си мы можем рассматривать любую часть любого массива как самостоятельный массив. Чтобы понять, насколько это может быть важно, достаточно сравнить приводившиеся в первых томах решения задачи о сопоставлении строки с образцом (см. §2.11.3 и §4.3.22); громоздкое паскалевское решение с его дополнительными параметрами для передачи границ и вспомогательной подпрограммой, которая эти параметры устанавливает, явно проигрывает ясному и изящному коду, написанному на Си.

Ещё одно важное и, несомненно, позитивное отличие Си от Паскаля состоит в чётком разграничении возможностей самого языка (того, что знает о языке компилятор) и возможностей библиотеки. Как мы имели возможность убедиться, успех Си во многом обусловлен именно этим.

Так или иначе, все перечисленные различия носят скорее технический, нежели концептуальный характер. Даже паскалевская нечёткость границы между языком и библиотекой может быть легко скорректирована, если нам придёт в голову написать свой компилятор Паскаля. В самом деле, Паскаль не перестанет быть Паскалем, если о всевозможных `writeln`, `abs` и прочем в таком духе компилятор будет узнавать из библиотек, вместо того чтобы пользоваться своими собственными знаниями об этих (ныне встроенных) процедурах и функциях. Даже наличие особого синтаксиса для `write`, `writeln` и `str` и вариадичность многих встроенных процедур вполне можно преодолеть, например, предусмотрев изначально отсутствующие в Паскале макросы (их можно сделать вариадическими без изменения конвенции передачи параметров в подпрограммы) и какие-нибудь «дополняющие» или «уточняющие» параметры, записываемые при вызове через двоеточие от основного. Что касается адресной арифметики, то, как показывает пример Delphi, её в Паскаль уже удалось затащить, притом без особых проблем.

При обсуждении парадигм программирования оказывается существенно интереснее другое различие между этими двумя языками, которое неизменно упускают из виду авторы всевозможных обзоров и сравнений. Оно и понятно, ведь в условиях многолетнего господства «си-подобных» языков программистская общественность начинает забывать смысл термина, ключевого для обнаружения этого различия, да и сам термин потихоньку забывается тоже. Но в нашей книге нужный термин упоминался и в предисловиях, и в части, посвящённой Паскалю, и, конечно же, в части, посвящённой Си[14]; читатель, несомненно, догадался, что речь идёт о словосочетании «***побочный эффект***».

Как мы неоднократно подчёркивали, среди тех, кто всё-таки помнит о существовании этого термина, крайне популярен миф о том, что «побочным эффектом» следует называть вообще любое действие, изменяющее хоть что-нибудь в среде выполнения программы — любой ввод-вывод (и вообще любое влияние на внешний мир, такое как отправка сигнала), присваивание, а в некоторых языках и более сложные

[12] См. файл `Documentation/filesystems/vfs.txt` в дереве исходных текстов ядра.

[13] С этими функциями мы встречались во втором томе, см. §8.2.3.

[14] В первом томе см. «предисловие второе, методическое», стр. 14, и §2.3.6; во втором томе — §4.8.3.

и хитрые действия. **На самом деле «побочный эффект» и «модифицирующее действие» — это совершенно не одно и то же**; строго говоря, побочный эффект есть частный случай модифицирующего действия, но не более того; ну а совпадать эти понятия начинают, во-первых, в функциональном программировании, и, во-вторых, в «си-подобных» языках, а точнее — в таких языках, где нет процедур, а есть только функции, и где присваивание может входить в выражения.

По-настоящему побочный эффект может возникнуть только при вычислении выражения, и это очень важно: побочный эффект — это произвольное изменение, которое происходит при вычислении выражения и может быть позже каким-то образом обнаружено. Если единственное, что происходит при вычислении выражения — это получение его результата (значения), говорят, что такое выражение (точнее, его вычисление) побочных эффектов не имеет.

В частности, присваивание в языке Си — это *операция*, которая может входить в более сложные выражения; выражение «`x = 7`» имеет *значение* `7`, а то, что при его *вычислении* меняется текущее значение переменной `x` — это как раз и есть побочный эффект. Как мы обсуждали при изучении языка Си, если нам нужно именно изменить переменную и больше ничего, мы можем превратить выражение присваивания в *оператор* (*statement*), добавив к нему точку с запятой:

```
x = 7;
```

Это так называемый *оператор вычисления выражения ради побочного эффекта*. Значение выражения здесь, как мы помним, игнорируется.

Привыкнув к такому положению вещей, программисты, работающие на Си, часто полагают, что присваивание вообще всегда есть побочный эффект, и в Си это действительно так — но это совершенно не так для Паскаля. **В Паскале присваивание — это оператор, а не операция; поскольку паскалевское присваивание (целиком) не представляет собой арифметического выражения, ни о каком побочном эффекте речи здесь идти не может**, ведь у операторов вообще — по определению — нет никаких побочных эффектов, они же не являются выражениями.

Конечно, в состав паскалевского оператора присваивания может входить арифметическое выражение произвольной сложности, причём как справа от знака присваивания, так и слева (например, в индексе массива), и это выражение вполне может производить побочные эффекты. Например, мы можем написать на Паскале функцию от целочисленного аргумента, которая будет возвращать квадрат своего аргумента, но при этом будет значение своего аргумента выводить в поток стандартного вывода («на экран»):

```
function f(x: integer) : integer;
begin
    writeln(x);
    f := x * x
end;
```

Функции, как мы знаем, вызываются из арифметических выражений, так что любое выражение, содержащее вызов этой функции `f`, будет обладать побочным эффектом — печатать значение аргумента функции. Поэтому, например, побочный эффект возникнет при выполнении следующих операторов присваивания:

```
a := f(15) + 2;
m[f(5) - 3] := 7;
```

но — и это следует чётко понимать — побочным эффектом тут обладают не сами операторы, а входящие в их состав выражения: в первом случае это `f(15) + 2`, во втором — `f(5) - 3`. Заявлять, что побочным эффектом тут будет обладать *оператор присваивания*, было бы несколько странно: мы ведь не говорим, что у того же оператора `while` может быть побочный эффект, даже если в его заголовке (не говоря уже про тело) окажется выражение с побочным эффектом.

Поскольку сам по себе вызов функции — это тоже арифметическое выражение, вполне правомерно будет сказать, что *функция `f` имеет побочный эффект*, и так действительно говорят. Но вот **у процедуры в том же Паскале никакого побочного эффекта быть не может**, что́ бы она ни делала, ведь её вызов — это никоим образом не выражение, это отдельный оператор, который так и называется «оператор обращения к процедуре» или «оператор вызова процедуры». Здесь уместно будет отметить, что в изначальном варианте Паскаля — в том его виде, в котором этот язык был предложен Никлаусом Виртом — вызвать функцию ради побочного эффекта (проигнорировав её возвращаемое значение) было нельзя. Это «неудобное» ограничение было устранено создателями Turbo Pascal; увы, эти люди, очевидно, привыкли программировать на Си, а о какой-то там «логике построения языка» или «чистоте семантики» попросту не задумывались.

Если вернуться теперь к языку Си, можно вспомнить, что в нём нет не только оператора присваивания, но и никакого оператора вызова процедуры, да и самих процедур тоже нет. На этом месте истинные почитатели Си обычно начинают возмущаться и напоминать про функции, имеющие `void` в качестве типа возвращаемого значения. Действительно, *технически* (на уровне машинного кода) паскалевская процедура и void-функция из Си — это одно и то же, но дело ведь не в том, какой из нашей программы получится машинный код; когда программист пишет программу на «совсем высокоуровневых» языках, он обычно вовсе не задумывается, как компилятор будет управляться с переводом его программы на язык процессора. Коль скоро мы обсуждаем парадигмы, нам важнее не то, что́ из нашей программы получится после её обработки транслятором, а то, как мы сами воспринимаем программу, когда пишем её.

Формально семантика функции типа `void` описывается фразой вроде «это функция, всегда вызываемая ради побочного эффекта» или «функция, возвращаемое значение которой всегда игнорируется». Функция при этом остаётся именно функцией, то есть её вызов есть по-прежнему частный случай арифметического выражения, а вызывается она, как и любая функция в Си, с помощью всё того же *оператора вычисления выражения ради побочного эффекта*. Хуже того — как мы уже обсуждали, в Си вообще любое действие оказывается побочным эффектом, а поскольку язык Си изначально императивный (фоннеймановский), то есть выполнение программы, написанной на этом языке, состоит в последовательном выполнении определённых действий, мы получаем своего рода квинтэссенцию «сишного» мышления: *выполнение программы (в терминах языка Си) состоит из побочных эффектов* — то есть в программе на Си вообще нет ничего другого, одни побочные эффекты. В таких условиях совершенно невозможно сохранить адекватное восприятие побочного эффекта как чего-то *нежелательного* — эдак, чего доброго, нежелательной станет вся программа. Иначе говоря, семантика языка Си неизбежно приводит к восприятию побочного эффекта как чего-то не просто «допустимого», а, лучше будет сказать, само собой разумеющегося.

Пример с копированием строки циклом, записанным в одну строчку, нам уже мог изрядно поднадоесть, этот пример приведён и в предисловиях, и в части, посвящённой Си, но всё же рискнём повторить его — ведь, пожалуй, это самая одиозная из всех строчек кода, когда-либо написанных на Си:

```
while((*dest++ = *src++));
```


Предложение написать вместо этого, например, так:

```
while(*src) {
    *dest = *src;
    src++;
    dest++;
}
*dest = 0;
```

зачастую вызывает жёсткий отпор — мол, так же длиннее, а чем оно при этом лучше? Правильный ответ на вопрос о недостатках первого варианта состоит в том, что там в условном выражении целых три побочных эффекта, что, несомненно, превращает код в ребус. Вот только

этот аргумент может совсем не повлиять на человека, привыкшего к реалиям Си: он нам ответит, что *в новом варианте кода побочных эффектов ровно столько же* — и с формальной точки зрения он будет прав. Аналогичный цикл, записанный на Паскале[15], не содержал бы ни одного побочного эффекта; совсем другое дело — Си.

Ситуация осложняется ещё и тем, что *в некоторых случаях* побочный эффект в условном выражении оказывается явлением скорее положительным, чем наоборот. Например, если нам потребуется прочитать последовательность чисел из текстового файла или другого потока до наступления ситуации «конец файла», на Си мы, скорее всего, применим следующий код:

```
while(1 == fscanf(stream, "%d", &n)) {
    /* здесь обрабатывается число n */
}
```

Разумеется, это можно написать иначе:

```
res = fscanf(stream, "%d", &n);
while(res == 1) {
    /* здесь обрабатывается число n */
    res = fscanf(stream, "%d", &n);
}
```

Побочного эффекта в условном выражении здесь уже нет, зато появилось дублирование кода. Что лучше — побочный эффект в условном выражении или дублирование кода? Можно сколько угодно ломать копья в пользу того или иного ответа на этот вопрос, но правильнее будет признать довольно очевидный факт: *на самом деле и то, и другое — плохо*. Источник проблемы мы подробно обсуждали во втором томе — сначала в части, посвящённой Си (см. т. 2, стр. 69), мы отметили сам факт возникновения потребности в построении цикла по схеме «выборка–проверка–обработка», а позже, добравшись до ***событийно-управляемого программирования***, узнали, что за этой схемой, если по ней построить ***главный цикл*** программы, стоит целая парадигма (см. т. 2, §6.4.3).

Такое построение циклов не укладывается в каноны структурного программирования, допускающего лишь циклы с предусловием и с постусловием; но **в реальной жизни в программах очень часто требуется именно такой цикл «с условием посередине»**, при этом существующие языки программирования такой конструкции не предоставляют. Здесь есть некое противоречие между практическими потребностями программистов с одной стороны и свойствами доступных инструментов — с другой; отсюда же вытекает и откровенная «кривизна» обоих рассмотренных нами только что решений — то ли дублировать код *выборки*, то ли загнать *выборку* в заголовок цикла в качестве побочного эффекта условного выражения. Часто встречается также и третий вариант — сделать цикл «бесконечным», а выходить из него оператором `break` или его аналогом, примерно так:

```
for(;;) {
    res = fscanf(stream, "%d", &n);
    if(res != 1)
        break;
    /* здесь обрабатывается число n */
}
```

Заметим, для главного цикла в событийно-управляемых программах наиболее типичен именно этот подход. Автору, впрочем, доводилось видеть программы на Си, в которых «`for(;;)`» выступал вообще единственным вариантом заголовка цикла; не делайте так, читать такое крайне тяжело. Но если такими «якобы-бесконечными» циклами не злоупотреблять, а применять их только для реализации схемы «выборка–проверка–обработка», в целом ничего страшного не случится. Конечно, это решение имеет свой собственный недостаток: обычно от конструкции бесконечного цикла читатель программы ожидает, что этот цикл окажется действительно бесконечным, то есть будет работать до тех пор, пока программу не завершат принудительно внешним воздействием, либо по крайней мере не случится какой-нибудь форс-мажор вроде фатальной ошибки. К тому же условие выхода, притом выхода *штатного* — такого, который применяется в обычных условиях, а не в каких-то редких особых случаях — оказывается спрятано где-то в недрах тела цикла, где его ещё нужно суметь заметить. Расфокусированным взглядом такое не поймать, `if`/`break` сольётся с окружающим кодом, и в итоге читателю будет труднее сообразить, для чего вообще этот цикл написан и что он делает.

Какое из трёх решений выбрать — во многом вопрос личных пристрастий, важно другое: мы здесь имеем внятно сформулированную проблему (как правильно оформить цикл, если по смыслу он должен состоять из выборки, проверки и обработки), однако не можем указать ни одного решения, полностью свободного от недостатков. Следовательно, в этой ситуации побочный эффект в условном выражении используется *по делу* и никоим образом не может свидетельствовать о безалаберности программиста — только (как максимум) о его вкусах, и нельзя сказать однозначно, что такие вкусы чем-то плохи, ведь два других решения тоже не слишком хороши.

Важно помнить, что цикл с проверкой посередине — это специфический частный случай, который никак не может оправдать *произвольное* применение побочных эффектов, особенно в управляющих условных выражениях. Ни в `if`'е, ни в цикле с постусловием (`do-while`) применение побочного эффекта в условном выражении не может рассматриваться как хорошая практика, да и в циклах с предусловием — `while` и `for` — нужно для начала убедиться, что имеет место именно построение цикла по схеме «выборка–проверка–обработка», и лишь после этого, обречённо вздохнув, вставлять в условие побочный эффект. Например, в программах, управляющих процессами[16] в системах семейства Unix, можно встретить следующую конструкцию:

```
while(wait(NULL) != -1)
    {}
```

Между прочим, «особо продвинутые» товарищи ухитряются написать вот так: «`while(~wait(NULL));`». Этот ребус (который, как ни странно, делает ровно то же) предложим читателю разгадать самостоятельно.

На первый взгляд может показаться, что тут (ну, по крайней мере в первом варианте) всё нормально: пока у нас есть процессы-потомки, `wait` по мере их завершения, превращения в «зомби» и окончательного удаления будет возвращать их `pid`'ы, а когда потомков не останется — вернёт минус-единицу, после чего цикл завершится; почему бы и не сделать так, если зачем-то нужно дождаться завершения всех наших потомков? Но торопиться не стоит: пытаясь натянуть на этот цикл схему «выборка–проверка–обработка», мы не найдём одного из звеньев — третьего, то есть вообще-то остаются только «выборка» (то есть некое действие) и «проверка» (результата этого действия), а это, очевидно, цикл с постусловием как он есть, и написать, разумеется, следовало иначе:

```
do {
    pid = wait(NULL);
} while(pid != -1);
```

Заметим, что если тело исходного цикла вдруг станет непустым — например, потребуется выдавать какие-нибудь сообщения, когда очередной процесс-потомок завершился — можно (и даже нужно) будет вернуться к схеме с побочным эффектом в заголовке цикла; но сначала тело должно перестать быть пустым.

Всевозможным конструкциям вроде «`a[i++]=b`», предполагающим больше одного побочного эффекта в одном выражении, тоже невозможно подобрать оправдание. Когда-то давно с их помощью можно было повысить быстродействие, но со времени появления оптимизирующих компиляторов подобный код окончательно перешёл в категорию неуместных трюков. Всё, что они делают — это *затрудняют* вос-

[15] Здесь приходится предположить, что либо это какой-то диалект Паскаля с адресной арифметикой, либо вместо указателей используются индексы.

[16] Если вы не помните, что делает вызов `wait`, перечитайте во втором томе §5.3.7.

приятие текста программы, пусть и незначительно; но наша цель при написании текста должна быть прямо противоположной.

Отдельного внимания заслуживают функции, которые на самом деле (по смыслу) должны быть процедурами, то есть написаны ради побочного эффекта, но при этом *что-то возвращают*. Когда «процедуре», написанной на Си, есть что сообщить вызывающему — хотя бы просто отчитаться об успешности или неуспешности своей работы — это практически всегда делается через возвращаемое значение функции. Семантическая нелепость этого подхода становится ясна, как только из подпрограммы требуется получить больше одного значения; во втором томе мы посвятили этой проблеме целый параграф (см. § 4.8.4). На Паскале обычно процедуры оформляют в виде «функций» ради возможности вернуть значение только программисты, привыкшие к си-подобным языкам[17], а люди, не имеющие таких сложностей, скорее снабдят процедуру параметром-переменной, чем переделают её в функцию; хорошим примером таких процедур выступают встроенные `BlockRead` и `BlockWrite` (см. §2.9.4), но есть и плохой пример — «функция» `ReadKey` из модуля `crt` (см. §2.8.3). Если попытаться перенести это правило на Си, предполагая, что аналогом процедур можно считать void-функции, то придётся переписать всю стандартную библиотеку, включая обёртки системных вызовов; то есть в этом плане ситуация с Си, скажем честно, совершенно безнадёжна, отделить процедуры от функций не получается никак.

Итак, побочные эффекты — это вроде бы нехорошо, но полностью запрещать их никак нельзя, цикл с выборкой, проверкой и обработкой может нам потребоваться вне зависимости от того, на каком языке мы пишем, в том числе и на Паскале, ну а если мы к тому же пишем на Си, то не запрещать же весь этот язык целиком — адекватной замены ему, как мы знаем, нет и не предвидится. Следовательно, нужна какая-то инструкция, по возможности лаконичная и ясная, о том, когда и как побочные эффекты применять можно, а когда — не стоит. В терминах Паскаля такая инструкция выглядит очень просто: *побочные эффекты применять не следует, за исключением случая, когда в виде побочного эффекта функции оформляется выборка при построении цикла по схеме «выборка-проверка-обработка»*.

В части, посвящённой языку Си, мы попытались сформулировать аналогичную инструкцию для этого языка и обнаружили, что семантика Си и традиции построения его стандартной библиотеки оказывают нам весьма достойное сопротивление. Для начала перед нами встаёт во весь рост проблема присваивания; но здесь хотя бы можно потребовать, чтобы присваивание всегда делалось ради побочного эффекта, то есть запретить использовать его значение — а значит, запретить вообще включать присваивание в более сложные выражения. С функциями хуже. Очевидный вариант — если по смыслу той или иной функции это на самом деле процедура, то есть её побочный эффект — это то, ради чего она написана, потребовать вызывать её как процедуру, не используя возвращаемое значение; но мы уже признали, что таких радикальных изменений требовать бесполезно.

Текст инструкции, который мы во втором томе всё же смогли кое-как состряпать (см. т. 2, стр. 69), вместе с примерами (а без них там вообще никак) занял больше страницы, но это не самое страшное. Хуже то, что наша инструкция, скорее всего, будет понятна лишь людям, знающим Паскаль или какой-то другой язык, в котором есть паскалеподобные процедуры; если же человек из императивных языков видел только Си, с хорошей вероятностью он элементарно *не поймёт, о чём* в этой инструкции *идёт речь* и откуда взялись такие требования. Мы просто не сможем объяснить такому человеку, где граница между допустимыми и недопустимыми побочными эффектами, нам для этого не хватит таких слов, которые были бы ему понятны. Если же мы продолжим настаивать на соблюдении требований, так и не объяснив причину их установления, то в ответ получим, скорее всего, только поток негатива, как это бывает всегда, когда человек вынужден исполнять непонятные для него инструкции.

Превратив присваивание из оператора в операцию и отказавшись от процедур как самостоятельной сущности, создатели Си тем самым выписали многим поколениям программистов универсальную индульгенцию на ничем не ограниченное и ничем не оправданное применение побочных эффектов к месту и не к месту. Язык Си *провоцирует* программиста на применение побочных эффектов, максимально такое применение упрощая и при этом затрудняя, насколько возможно, любые объяснения, почему так делать не стоит. В этом и состоит концептуальное или, если угодно, парадигматическое отличие Си от Паскаля, а заодно причина совершенно катастрофического влияния Си на неокрепшие мозги начинающих.

Автор вынужден признать, что ещё совсем недавно не осознавал этого — хотя всегда утверждал, что язык Си калечит мышление, но тем не менее не мог дать краткого и точного ответа на вопрос, как и чем именно калечит. Теперь такой ответ есть, и, кстати, именно его появлением обусловлена большая часть отличий текста второго издания этой книги, вышедшего в 2021 г., от первого, публикация которого началась на пять лет раньше (и именно тогда, в 2016 г., вышли первые два тома, содержавшие и часть о Паскале, и часть о Си).

Из нашего рассуждения можно сделать достаточно важный вывод. Влияние языка программирования на мышление часто недооценивается, и недооценивается совершенно напрасно. Даже два столь похожих друг на друга языка, как Си и Паскаль, оказываются при внимательном рассмотрении совершенно разными в плане парадигм, причём Си, если не принять серьёзных мер предосторожности, способен мышление программиста необратимо искалечить, а если его изучать в качестве первого в жизни языка, то никакие меры не помогут. По-видимому, к выбору языков программирования, причём не только в начальном обучении, но и в практической работе, следует подходить с куда как бóльшей тщательностью, нежели это в наши дни принято.

Коль скоро мы обсуждаем парадигмы программирования, стоит признать, что **побочный эффект (функции, выражения и т. п.) — это, несомненно, тоже парадигма**, т. е. применение побочных эффектов требует особого подхода к осмыслению происходящего. Проиллюстрировать этот момент довольно просто. Ещё сравнительно недавно на факультете ВМК МГУ, где автор этих строк проработал больше двадцати лет, зачётные комиссии по программированию (мероприятие для студентов, зачёта не имеющих, т. е., грубо говоря, для двоечников) проводились в форме письменной работы. При проверке таких работ неоднократно попадалось, например, следующее:

```
fork();
if(fork() == 0) {
    /* ... */
```

Умственные способности студентов, пишущих подобное, обсуждать не будем; гораздо интереснее попытаться понять, что же творится у автора такой работы в голове — и по какой причине. Отметим для начала очевидное: студент, написавший подобный фрагмент кода, концепцией побочного эффекта не владеет, в противном случае он бы понимал, что `fork` у него отработает дважды (а точнее — в общей сложности трижды, но это уже особенность самого `fork`). Несмотря на это, `fork` в заголовке `if`'а у него почему-то есть. Ответ на это «почему» на самом деле тривиален: *он видел, что так пишут преподаватели*. При этом, с одной стороны, концепция побочного эффекта у этого студента в голове не укладывается, то есть он не может понять, что *действие* (в данном случае порождение процесса) может произойти в ходе вычисления условного выражения — ведь там же *просто сравнение двух чисел!* С другой стороны, он точно знает, что порождение процесса, требующееся в задаче, делается системным вызовом `fork` (этот факт, увы, просто зазубрен). Итак, в мозгу несчастного обучаемого сталкиваются три постулата:

- надо породить процесс, а на лекциях говорили, что для этого нужно вызвать `fork`;
- надо обязательно написать «`if(fork() == 0) {`», потому что преподаватели всегда так писали, и, значит, это точно правильно;

[17] Во время подготовки первого издания книги автор, к его огромному сожалению, сам входил в их число, и преодолеть этот росток сишности мозга ему стоило определённого труда, в том числе сугубо технического: ко второму изданию в паскалевской части книги пришлось переделывать едва ли не половину примеров.

- условное выражение — это просто сравнение, оно, *разумеется*, не может ничего сделать, в том числе породить процесс (эту мысль обучаемый не облекает в слова, ограничение здесь сугубо подсознательное).

На выходе имеем то, что имеем. Виноват ли в этом сам студент? Несомненно! Если на занятиях что-то непонятно, надо разобраться, задать вопросы преподавателю или в крайнем случае другим студентам, хоть тушкой, хоть чучелом — но добиться понимания происходящего, а не дожидаться зачётной комиссии по принципу «авось прорвёмся». Но, с другой стороны, кто мешает *преподавателям* написать

```
pid = fork();
if(pid == 0) { /* child */
    /* ... */
```

— тем самым резко повысив шансы отстающих студентов понять, о чём идёт речь и как это всё работает?

Между прочим, вопрос не столь риторический, как может показаться — особенно в контексте обсуждения парадигм. Для преподавателей, разумеется, концепция побочного эффекта — барьер давно пройденный и забытый, притом для большинства из них — пройденный незамеченным. До когнитивных проблем, возникающих у студентов, преподаватели не снисходят, поскольку *не понимают, что эти проблемы вообще могут существовать*: ну в самом деле, *да что тут может быть непонятного?* Касается это, впрочем, отнюдь не только преподавателей программирования; преподаватели других предметов тоже в большинстве своём не считают нужным разобраться, что происходит в головах студентов, которые не поняли объяснение с первого раза.

Что же касается программирования, то тут автор с прискорбием вынужден констатировать крайне неприятный факт, ещё больше усугубляющий проблему: многие преподаватели сами страдают пресловутой «сишностью головного мозга», так что их ученики либо ничего не понимают, либо в результате обучения пополняют ряды «истинных сишников». Как исправить сложившуюся ситуацию и возможно ли вообще её исправить — вопрос открытый.

9.3.3. Два подхода к сравнению языков

Количество языков программирования, придуманных за время существования компьютеров, исчисляется десятками тысяч, но «лучшего» среди них так и не появилось; больше того, при пессимистичном взгляде на происходящее можно заявить — и не без оснований — что в мире вообще нет *хороших* языков, все имеющиеся языки отвратительны, просто некоторые отвратительны чуть меньше других.

Так или иначе, языки программирования приходится *сравнивать* между собой, и в том, как именно люди их сравнивают, внезапно обнаруживаются два взаимоисключающих подхода, которые, в принципе, тоже можно считать парадигмами, поскольку эти подходы в действительности вытекают из субъективного восприятия языка программирования как явления.

Язык программирования можно воспринимать как инструмент, позволяющий объяснить компьютеру — или даже шире, *вычислительной системе*, включающей компьютер, операционку и, возможно, часть системного и прикладного программного обеспечения — что и как должно быть сделано, чтобы поставленная (перед программистом) задача оказалась решена. Обычно при этом язык рассматривается как единое целое с его реализацией — транслятором и всевозможными дополнительными программами — в качестве этакого средства доставки нужной функциональности на компьютер (в систему) конечного пользователя, покрывающего все этапы такой доставки, от написания программы, её отладки и подготовки к передаче конечному пользователю — до инсталляции на машине конечного пользователя, эксплуатации и даже получения обратной связи вроде сообщений о произошедших у пользователя ошибках; но ключевым тут остаётся восприятие языка и его реализации как чего-то такого, что позволяет *заставить компьютер делать то, что нужно*, или, скажем, *описать, как возможности компьютера должны быть задействованы для решения поставленной задачи.*

Но можно подойти к языку программирования совсем по-другому. Многие программисты склонны рассматривать язык исключительно как знаковую систему, позволяющую сформулировать (если угодно, представить в объективной форме) их знание или понимание, в чём состоит решение поставленной задачи; от предыдущего варианта этот отличается тем, что всякие там компьютеры, операционные системы, внешние программы и прочее в таком духе из рассмотрения исключается как нечто несущественное. Как будет выглядеть их программа в системе пользователя (не то, как выглядит её пользовательский интерфейс, а то, как выглядит она сама — например, когда её установили, но ещё не запустили), как именно она будет запускаться и выполняться — всё это неважно и вообще не относится к деятельности программиста, пусть об этом позаботится кто-нибудь ещё.

Первый шаг к такому восприятию языка программирования делается одновременно с заявлением, что высокоуровневые языки позволяют абстрагироваться от устройства компьютера, перестать о нём думать и сосредоточиться на решении задачи, и это правильно, поскольку решение задачи само по себе достаточно трудно. Следующий «логичный» шаг оказывается почти незаметным, и оттого практически неизбежен: принцип «для решения задачи не обязательно думать о том, как устроен компьютер» подменяется сначала на «не обязательно *знать*, как устроен компьютер», чуть позже — на «не требуется понимать, как работает компьютер». Полученный принцип — опять же практически неизбежно — переносится на процесс обучения новых программистов, в результате чего образуется очередной конвейер по производству горе-специалистов, не понимающих, что делают; таких «спецов» в просторечии часто называют «макаками».

О восприятии языка программирования исключительно как системы обозначений свидетельствует, например, описание достоинств того или иного языка фразами вида «смотрите, как на этом языке легко написать то-то и то-то»; логичное продолжение вроде «а на вашем языке то же самое приходится писать намного дольше и сложнее» не всегда произносится, но практически всегда подразумевается. Недоумённое «минуточку, да ведь твой язык — интерпретируемый, а мой — компилируемый» расшибается о чугунное «ну и что»; в лучшем случае нам ещё заявят, что, мол, по эффективности наш замечательный язык почти не проигрывает этим вашим компилируемым, хотя про эффективность вроде бы никто не спрашивал. Автору доводилось видеть людей, уверенных, что операционную систему можно написать на чём угодно — хоть на Лиспе, хоть на Питоне; вопрос, откуда на «голом железе» возьмётся интерпретатор Питона, этих людей не беспокоил.

Различие между двумя описанными подходами можно описать ещё проще, если сформулировать своего рода характеристическое утверждение: **язык общего назначения не должен *навязывать* ничего такого, что будет видно конечному пользователю**. Выбор языка (и, говоря шире, *инструментов*) — это до определённой степени личное дело программиста, и если выбранный язык что-то навязывает *самому программисту* — то никого другого это не касается; но это верно лишь до тех пор, пока сие никак не влияет на конечный результат — программу, передаваемую пользователю. Если же, выбрав инструмент, уже сам программист начинает что-то навязывать пользователю — например, «всего лишь» установку интерпретатора на компьютер — то язык, по-видимому, выбран неправильно и на роль языка общего назначения не годится. Так вот, приверженцы «инструментального» восприятия языка с этим утверждением, скорее всего, согласятся, или как минимум будут готовы его обсуждать, прикидывать выгоды и издержки и т. п. Напротив, индивиды, для которых язык лишь система знаков, от утверждений подобного рода могут только отмахиваться — ведь их восприятие предполагает, что особенностями целевой системы можно *пренебречь*, примерно как сопротивлением воздуха в школьных (читай — бессмысленных) задачах по физике.

К рассмотрению интерпретации и компиляции в качестве особых парадигм программирования мы вернёмся в последней части нашей

книги, а пока просто учтите, что восприятие языка программирования как знаковой системы в отрыве от реализации иногда, как ни странно, вполне допустимо — но отнюдь не всегда.

9.4. Примеры частных парадигм

Случается, что мы имеем дело с частной парадигмой, совершенно об этом не задумываясь. Например, в коде начинающих программистов можно встретить что-то вроде следующего:

```
if(a == b)
    return 1;
else
    return 0;
```

Автору этих строк неоднократно встречались студенты, из чьего кода подобная конструкция никак не может исчезнуть даже после того, как им неоднократно было сказано, показано и объяснено, что вместо этих странных четырёх строк следует, разумеется, написать один оператор:

```
return a == b;
```

Причина этого упорства, судя по всему, очень проста: человек, пишущий вот такие вот `if`'ы, на уровне подсознания *не владеет концепцией логического выражения*, то есть он не может убедить сам себя в том, что «условие», фигурирующее во всяческих ветвлениях и циклах — это такое же арифметическое выражение, оно *вычисляется*, давая *значение*. По этой же причине в программах на Паскале часто можно увидеть что-то вроде

```
if flag = true then
```

— это означает, что автор программы не может заставить себя воспринимать условие как что-то иное, нежели бинарное отношение вроде равенства или неравенства. Вынести вычисление сложного (иногда в пять-шесть строк, а то и длиннее) условия в отдельную функцию такие программисты тоже не могут, им это просто не приходит в голову.

Сравнение с нулём в программах на Си, в отличие от Паскаля, не обязательно свидетельствует о том же самом, ведь ноль в Си означает не только логическую ложь, но и арифметическое значение; в частности, использовать результат функции `strcmp` в роли логического значения настоятельно не рекомендуется, поскольку оно таковым не является, и если нужно сравнить две строки на равенство, то лучше написать именно сравнение с нулём, а не логическое отрицание.

Итак, парадигмой, притом самостоятельной (ведь подобные странности могут встретиться не только в императивных программах) оказывается даже столь простой предмет, как логическое выражение — точнее даже не само по себе «выражение», а скорее «условие как вычисляемое значение». Попытаемся припомнить содержание предыдущих томов нашей книги и отметим ещё несколько случаев, когда мы в действительности имели дело с частными парадигмами, но об этом не задумывались.

9.4.1. Callback-функции

Рассматривая во втором томе (см. §4.9.2) указатели на функции, мы в качестве одной из иллюстраций их полезности привели так называемые ***callback-функции***, или *функции обратного вызова*. Этот приём может быть применён, когда одна подсистема программы (например, модуль), называемая в данном случае *сервером*, предоставляет другим подсистемам (*клиентам*) некие услуги, которые по каким-то причинам невозможно или нежелательно оказывать непосредственно в точке вызова соответствующей функции. Искомые услуги могут быть отложены по времени (если, скажем, услуга состоит в том, чтобы получить какую-то информацию от удалённого сервера), или они должны быть оказаны после длинной цепочки вызовов разнообразных функций, или на одно обращение может быть оказано *больше одного экземпляра* услуги (например, может быть найдено несколько экземпляров однотипных объектов в памяти), но при этом по каким-то причинам нежелательно использование динамических структур данных для передачи результатов.

Суть callback-функций состоит в том, что клиент, обращаясь к серверу за услугой, предоставляет серверу некую функцию, которую тот должен вызвать, когда будет готов оказать услугу. Например, если услуга состоит в том, чтобы получить данные через компьютерную сеть, подсистема-сервер дождётся момента, когда данные будут полностью готовы, и уже тогда вызовет функцию, полученную от подсистемы-клиента, передав ей эти данные через параметр. Если же услуга, например, состоит в поиске тех или иных объектов данных в какой-нибудь сложной структуре вроде дерева или хеш-таблицы, и этих объектов может быть найдено больше одного, то подсистема-сервер вызовет callback-функцию по одному разу для каждого найденного объекта, передавая через параметры этот объект. Как правило, во всяком случае при работе на языке Си, callback-функцию снабжают дополнительным параметром, обычно имеющим тип `void*`, который называется *указателем на пользовательские данные* и позволяет клиенту организовать изнутри обратных вызовов доступ к произвольным данным, требующимся при решении задачи; в §4.9.2 мы приводили пример использования такого параметра.

Некоторые программисты предпочитают не использовать механизм callback-функций из-за возникающего ощущения неуверенности. Дело тут в том, что в традиционную императивную модель мышления этот приём не укладывается: при реализации подсистемы-клиента мы имеем дело с некими *действиями*, которые должны произойти в нашей среде исполнения, но про которые мы точно не знаем, когда, в какой последовательности и даже в каком количестве эти действия произойдут. Для человека, привыкшего мыслить императивными категориями, такая ситуация непривычна и вызывает определённый дискомфорт.

Следует подчеркнуть, что весь этот дискомфорт — лишь следствие недостаточной гибкости мышления. У программистов, имеющих опыт работы с функциональными языками, техника callback-функций обычно никаких проблем не вызывает. Впрочем, привыкнуть к нужному способу мышления можно и без такого опыта, это вопрос практики; главное — не зацикливаться на традиционном «последовательном» императивном восприятии программ.

Сделаем важное замечание: чтобы пояснить, в чём заключается техника callback-функций, нам пришлось привлечь термины «сервер» и «клиент». В самом деле, при использовании callback-функций задействуется ещё одна частная парадигма, с которой мы сталкивались ранее — ***клиент-серверная модель***. В роли клиента и сервера в данном случае выступают разные части одной программы, что никоим образом не отменяет их осмысления в качестве «действующего лица, оказывающего услуги по запросу» и «действующего лица, обращающегося за услугами».

9.4.2. Программирование в терминах явных состояний

Выше мы несколько раз упоминали ***событийно-управляемое программирование***, с которым впервые столкнулись во втором томе, рассматривая варианты построения сетевого сервера. Что событийно-управляемое программирование *само по себе* является парадигмой — несомненно; больше того, некоторые библиотеки, предназначенные для создания графических пользовательских интерфейсов, успешно скрывают факт существования главного цикла, в результате чего в восприятии программиста выполнение программы из единой последовательности действий превращается в нечто фрагментарное — выполнение отдельных подпрограмм-обработчиков. Но даже при использовании таких библиотек программист всё-таки *может* помнить о существовании главного цикла, пусть и скрытого от него, и тогда о программе можно продолжать думать в традиционных императивных терминах; если вспомнить о callback-функциях, то окажется, что в событийно-управляемой модели фактически используется этот приём: в роли сервера выступает реализация главного цикла, а обработчики событий оказываются как раз callback-функциями, которые глав-

ный цикл вызывает при наступлении события; в качестве оказываемой услуги следует рассматривать ***выборку событий***.

Если метаморфозы восприятия на этом закончатся, результат на выходе получится несколько печальный. Программа с графическим интерфейсом будет совершенно неожиданно для пользователя «уходить в себя» на ощутимые промежутки времени, ни на что не реагируя и вызывая раздражение; серверная программа, задерживая работу с клиентами, может вызвать разрыв соединений по тайм-ауту и, как следствие, оказаться неудовлетворительной в смысле надёжности. Если серверные программы столь низкого качества встречаются достаточно редко, то в программах, где событийно-ориентированная модель использована для построения GUI, сия печальная особенность проявляется в наши дни намного чаще, чем хотелось бы. Пользователи начали уже *привыкать* к раздражающим своими замираниями программам, как к чему-то неизбежному, а корпорации, торгующие цифровыми «железяками» (в особенности смартфонами), весьма грамотно используют сложившуюся ситуацию, чтобы продолжать втюхивать публике всё более и более «мощные» устройства: якобы чем мощнее процессор, тем лучше будут работать на нём программы и тем меньше они будут тормозить. «Эволюция» смартфонов дошла до того, что встроенные аккумуляторы не способны обеспечить питание процессора (а то и нескольких процессоров — такие смартфоны уже тоже существуют) даже в течение одного дня — скорость вычислений просто так не вырастет, за неё приходится расплачиваться подогревом окружающей среды. Потратив кучу денег на заведомо неадекватное своим функциям техническое устройство — фактически суперкомпьютер в корпусе телефона — пользователь в довершение всего вынужден таскать с собой зарядное устройство и внешний аккумулятор, и, что ещё хуже, *никто не замечает абсурдности происходящего*.

При этом даже не все программисты понимают, что, когда программа то и дело подвисает, проблема-то не в недостатке мощности процессора. Просто при написании программ их авторы начисто забывают о важнейшем требовании, накладываемом на обработчики событий: **программа не имеет права задерживаться в обработчике надолго**. Во втором томе мы обсуждали, что это «надолго» может в зависимости от задачи означать довольно разные количественные ограничения[18], но несомненно одно: **блокирующие системные вызовы в обработчике событий недопустимы категорически**. Если об этом забыть, никакая скорость процессора горю не поможет — длительность блокировок от неё не зависит.

Недопустимость блокировок влечёт за собой ещё одну парадигму, обсуждению которой мы во втором томе посвятили целый параграф (§6.4.5) — ***программирование в терминах явных состояний***, иногда называемое также ***автоматным программированием***, поскольку необходимый при этом стиль мышления очень похож на тот, что приходится применять при работе с формальными автоматами — простыми ***конечными*** (англ. *FSM — finite state machine*) и более сложными, такими как машина Тьюринга, автоматы с магазинной памятью и т. п.

Надо сказать, что этот стиль встречается не только в событийно-ориентированных программах. Совсем недавно, рассматривая итеративное решение для задачи обхода двоичного дерева (см. стр. 17) и сравнивая его с решением рекурсивным, мы отмечали, что в итеративном решении нам пришлось *явным образом* выделить информацию о *текущем состоянии обхода дерева*, тогда как рекурсивное решение использовало *ту же самую* информацию, представленную неявно — в форме *текущей позиции* исполнения в функциях, на настоящий момент вызванных, но ещё не завершённых.

Можно сказать, что «автоматное» программирование как раз и состоит в этом переходе от неявных составляющих состояния выполнения программы к их явному указанию в виде значений переменных. В примере с обходом дерева явное состояние было представлено односвязным списком, содержащим, во-первых, указатели на корневые узлы поддеревьев, рассмотрение которых началось, но ещё не закончилось, и, во-вторых, переменные, показывающие, что́ уже сделано и чего ещё не сделано с данным конкретным узлом (не сделано ещё ничего; начат обход левого поддерева, так что после возврата к этому узлу остаётся рассмотреть значение из самого этого узла и обойти правое поддерево; начат обход правого поддерева, так что после возврата к данному узлу следует завершить его рассмотрение, выбросив его из списка рассматриваемых узлов). В рекурсивном решении как раз вот это состояние дел с текущим узлом представляется неявно в виде текущей позиции исполнения в функции, вызванной для обработки данного узла с его поддеревьями.

Формально говоря, решение, обсуждавшееся в §9.2.1, не вполне соответствует автоматной парадигме, и виной тому небольшая оптимизация, на которую мы обратили внимание читателя в комментарии на стр. 18: поменяв состояние на следующее, мы позволили себе не завершать на этом обработку *шага автомата*, а сразу переходить к работе со следующим состоянием — для этого в операторе `switch` отсутствовали привычные `break`'и. Если вернуть эти `break`'и обратно — туда, где вместо них в тексте решения стоят комментарии «`no break here`» — мы окончательно приведём решение в соответствие модели программирования в терминах явных состояний (как мы уже отмечали, в данном случае результаты работы от этого не изменятся).

Попытаемся несколько формализовать требования, при выполнении которых можно сказать, что в данной программе или её фрагменте применено программирование в явных состояниях. Прежде всего признаем, что всё это не более чем частный случай императивного программирования, то есть программирования, основанного на модификациях — но вместо расплывчатых «модификаций среды выполнения» мы здесь говорим об *изменениях состояния формального автомата*, причём этот «автомат» может существовать только в нашем воображении, он, скорее всего, придуман специально для конкретной задачи и мы даже не потрудились дать ему формальное описание, но всё это не отменяет автомата как базовой абстракции. Подчеркнём ещё раз, что формальный автомат — это абстракция более общая, нежели «конечный автомат» (та же машина Тьюринга не является конечным автоматом, но, бесспорно, является автоматом формальным); фактически мы можем говорить об очередном формальном автомате всякий раз, когда мы можем указать, из чего состоит и чем определяется его состояние и по каким законам это состояние будет изменяться.

Состояние автомата должно описываться некоторой *единой* структурой данных — от простой целочисленной переменной (если нам для наших целей хватает возможностей простого конечного автомата) до сколь угодно сложных систем взаимосвязанных объектов в памяти; определяющим тут остаётся *единство* всех этих объектов в представлении программиста, чем бы такое «единство» ни было на самом деле. Например, программист вполне может заявить, что состояние его автомата определяется «вот этими вот семью локальными переменными», коль скоро переменные расположены рядом и их взаимосвязь очевидна из текста программы — хотя лучше было бы, наверное, объединить переменные в единую структуру, но это не обязательно; с другой стороны, заявлять, что состояние определяется набором переменных, описанных в разных местах, было бы уже несколько странно. Конечно, здесь сложно провести формальную границу допустимого, но при описании *человеческого мышления* вообще сложно говорить о формальностях, а мы ведь обсуждаем парадигмы программирования, то есть именно особенности *мышления* программиста.

Работа автомата должна быть разбита на ясно разделённые между собой *шаги*. Очередной шаг выполняется при наступлении какого-то события, внешнего по отношению к автомату. Таким событием может стать чтение очередного символа из потока, наступление определённого временно́го момента или что-то ещё, включая, разумеется, выборку события в событийно-управляемой программе; более того, программа, использующая автомат, может вообще потребовать от автомата выполнения его шагов, не располагая для этого никакими событиями — это неважно, с точки зрения автомата событием будет сам

[18] Впрочем, в программе с GUI за проведённую в обработчике одну десятую секунды автора этого обработчика следует по меньшей мере обработать канделябром.

факт этого внешнего требования. В нашем автомате, обходившем двоичное дерево, всё именно так и было — его шаги выполнялись подряд в цикле, который никаких событий не ждал. Если, скажем, нам придёт в голову написать эмулятор машины Тьюринга или другой подобной модели, мы, скорее всего, реализуем её в автоматном стиле, но никаких внешних событий у нас, опять-таки, здесь не будет.

Выполняя очередной шаг, автомат может как менять собственное состояние, так и производить какие-то внешние действия, но здесь имеется серьёзное ограничение: **любые действия, которые выполняет автомат, кроме изменения собственного состояния, не должны оказывать никакого влияния на работу самого автомата**. В частности, нельзя из одного шага автомата переходить в другой его шаг, требовать повторения шага (вызывать шаг из него самого), или каким бы то ни было образом пытаться управлять потоком внешних событий, в ответ на которые происходят шаги. Иначе говоря, **автомат не может решать, *когда* ему делать очередной шаг**, он лишь решает, ***что*** должно быть сделано в рамках шага, *если* (когда) от него потребовали сделать шаг. Например, если с помощью формального автомата мы анализируем цепочку символов, то автомат не может *сам* читать символы, то есть выполнять операцию чтения или выборки символа изнутри своих шагов — обязательно должен быть кто-то внешний, кто «скармливает» автомату эти символы один за другим, заставляя автомат проделать очередной свой шаг для обработки каждого из символов.

Принятие решения, что конкретно должно быть сделано за очередной шаг, может быть основано только на двух вещах: на собственном состоянии автомата и на наступившем событии, в ответ на которое производится шаг — если, конечно, эти события чем-то друг от друга отличаются.

Отметим ещё один неочевидный момент. Программе, использующей автомат, совершенно не обязательно обращаться к нему (инициировать выполнение его шагов) в цикле, хотя во многих случаях именно так и делается. Представьте себе, что у вас в некоторой сложной структуре данных (например, в дереве) хранятся значения, которые вашей программе потребовались *по одному* — например, это какие-нибудь одноразовые ключи, которые нужно передавать клиентам, связавшимся с вами через компьютерную сеть, по одному в каждом сеансе связи. В этом случае бессмысленно реализовывать обход дерева единой процедурой — хоть рекурсивной, хоть циклической. Процедура обхода выдаст вам *сразу все* значения из дерева, а вам такое значение нужно *одно*, причём каждый раз новое (очередное выбранное из дерева или что у вас там). В такой ситуации у вас остаётся едва ли не единственный выход — реализовать автомат, который на каждом своём шаге будет выдавать очередное нужное значение. Каждый раз, когда вам требуется значение (в нашем примере — когда очередной сетевой клиент от вас таковое запросил), вы обращаетесь к автомату, заставляя его сделать один шаг, и получаете искомое; здесь ни сам автомат, ни даже тот, кто его вызывает, не будет знать, когда случится следующий шаг.

9.4.3. Метапрограммирование

Приставка *мета-* в одном из своих значений соответствует, если можно так выразиться, применению чего-то к нему же самому, или, говоря строже, применение термина к явлению, которое этот термин исходно обозначает; так, если галактика — это скопление звёзд и других космических объектов, то метагалактика — скопление скоплений (галактика галактик); метаданные — это данные о данных, метаязык — язык для описания других языков и т. п. С учётом этого можно догадаться, что под ***метапрограммированием*** понимается написание программного кода, результатом выполнения которого будет программный код.

Наиболее очевидным вариантом здесь будет написать некую программу, которая сгенерирует и выдаст в свою очередь программу (возможно, на другом языке, а может быть, и на том же самом), и уже эта сгенерированная программа, будучи исполненной, позволит получить результаты, которые изначально были нужны. Так действительно иногда делают, хотя и сравнительно редко; намного чаще генерируется не целая программа, а какая-то её часть, например, отдельный модуль. Такую возможность следует как минимум принимать во внимание.

В примерах, которые мы рассматривали в нашей книге, пока ничего подобного не применялось, но с метапрограммированием как явлением мы уже сталкивались, и не раз. Речь в данном случае идёт о ***макросах*** и макропроцессировании. Тело макроса представляет собой код, превращающийся во фрагмент текста программы в ходе макрорасширения, то есть макроопределение — это и есть метакод, а его написание — метапрограммирование. Конечно, примитивный макропроцессор языка Си на серьёзный инструмент макропрограммирования, как говорится, не тянет; но вот рассматривавшийся в первом томе макропроцессор, встроенный в ассемблер NASM, позволяет заниматься метапрограммированием полноценно — благодаря алгоритмической полноте, обеспечиваемой макроповторениями и другими возможностями этого макропроцессора.

В следующих частях книги мы ещё много раз встретимся с разнообразными проявлениями метапрограммирования. Так, вскоре нам предстоит изучать шаблоны Си++, которые в определённом смысле тоже являют собой макросы; как мы убедимся позднее, в интерпретируемых языках программирования спектр возможностей метапрограммирования существенно расширяется. Но всё это дело будущего; пока стоит отметить, что метапрограммирование, несомненно, представляет собой парадигму. Чтобы представить себе не только то, как будет выполняться сам метакод, но и то, как будет вести себя код, ставший его результатом, требуется определённое интеллектуальное усилие и мышление, отличающееся от обычного.

Часть 10

Язык Си++, ООП и АТД

Эта часть нашей книги будет целиком посвящена изучению языка Си++ и парадигм программирования, которые в нём поддерживаются. Язык был предложен Бьёрном Страуструпом в начале восьмидесятых годов прошедшего столетия в качестве ответа на назревшую потребность в индустриальном объектно-ориентированном языке и изначально построен как расширение (хотя и не совсем строгое, как мы увидим чуть позже) хорошо знакомого нам языка Си; добавленные в язык инструменты обеспечивают поддержку как объектно-ориентированного программирования, так и целого ряда других парадигм — абстрактных типов данных, обобщённого программирования, обработки исключительных ситуаций и т. д.

Парадигму объектно-ориентированного программирования мы уже начинали обсуждать в предыдущей части (см. стр. 15); там же мы упомянули ещё одну важную парадигму — *абстрактные типы данных*.

В языке Си++ представлены средства как для ООП, так и для создания АТД, причём сложно сказать, какая из этих двух парадигм поддержана «лучше». Запутанное положение усугубляется ещё и тем, что для обеспечения «закрытости» (как объектов, так и АТД) Си++ вводит единый инструмент, так называемую *защиту*, и вдобавок на происходящее накладывает заметный отпечаток исходная императивная сущность чистого Си. К тому же ООП и АТД не исчерпывают новшеств, введённых в Си++ по сравнению с чистым Си; так, *обработка исключений* вообще никакого отношения к ООП не имеет, и у многих программистов этот факт вызывает недоверчивое удивление, поскольку в Си++ исключения тесно связаны с объектами и наследованием. Между ООП и «всем остальным» язык Си++ сам по себе никакой чёткой границы не проводит. Встречаются программисты, вообще не умеющие использовать ООП, но при этом уверенные, что то, что они делают — это ООП и есть.

Мы постараемся своевременно напоминать читателю, что в действительности абстрактные типы данных — это ещё не ООП. Сразу же оговоримся, что абстрактные типы данных сами по себе достаточно полезны, и эта польза не станет меньше от того, что их путают с объектами; но умение мыслить в терминах объектов и сообщений, то есть именно то, что составляет суть объектно-ориентированного программирования, тоже очень важно, в особенности при создании больших и сложных программных систем. Поэтому будет уместно посоветовать читателю регулярно вспоминать про объекты и сообщения, пока мышление в этих терминах не станет для него привычным делом.

Рассмотрев инструменты, имеющиеся в Си++, мы обязательно ещё раз вернёмся к вопросу о парадигмах, то есть о том, как, применяя возможности языка, можно осмысливать свою программу различными способами. Этот вопрос очень важен, поскольку всё, что есть (и может быть) в языке программирования — лишь некая *техническая* поддержка парадигм, тогда как сами парадигмы существуют лишь в мышлении программиста.

10.1. От Си к Си++

Как мы уже упоминали, Си++ построен как расширение Си, но в силу целого ряда исторических причин не является его строгим надмножеством. Программы, написанные на чистом Си, в большинстве случаев будут корректны с точки зрения языка Си++, но не всегда. Так, в Си++ присутствует целый ряд ключевых слов, которых не было в Си. Поэтому, например, программу, содержащую такое описание:

```
int try;
```

транслятор Си++ сочтёт ошибочной, ведь слово `try` — ключевое в Си++. Кроме того, в языке Си++ нет отдельного пространства имён для тегов структур; описание

```
struct mystruct {
    int a, b;
};
```

в программе на Си++ является описанием полноценного *типа* `mystruct`, тогда как на Си такое же описание означало бы лишь введение *тега структуры*. Иначе говоря, в Си++ мы можем описать переменную типа `mystruct`:

```
mystruct s1;
```

а в чистом Си нам приходилось использовать что-то вроде

```
struct mystruct s1;
```

(впрочем, Си++ поймёт и такое). С другой стороны, часто встречающиеся в старых программах на Си описания вида

```
typedef struct mystruct {
    int a, b;
} mystruct;
```

в программе на Си++ вызовут ошибку из-за возникающего конфликта имён; в языке Си такого конфликта не возникало, т. к. теги структур и имена типов относились к различным пространствам имён.

В ряде ситуаций компилятор Си++ ведёт себя строже, чем компилятор чистого Си, выдавая, например, ошибку там, где в Си возникло бы как максимум предупреждение. К таким ситуациям относятся, во-первых, вызовы необъявленных функций: в Си эта ситуация считается «нехорошей, но допустимой», так что, если забыть подключить какой-нибудь заголовочный файл, программа может благополучно оттранслироваться, хотя и с предупреждениями. В Си++ этот номер не проходит: если функция не описана, её вызов рассматривается как ошибка. Во-вторых, ошибкой будет неявное преобразование адресных выражений несовместимых типов: например, присваивание выражения типа `int*` переменной типа `double*` вызовет ошибку в Си++, тогда как в чистом Си выдавалось только предупреждение. Ошибкой в Си++ будет и неявное преобразование выражения типа `void*` в выражение другого адресного типа, в то время как компилятор чистого Си не выдаёт в такой ситуации даже предупреждения.

В чистом Си для обозначения «нулевого указателя», т. е. специального адресного значения, заведомо не соответствующего никакому адресу в оперативной памяти, используется макрос `NULL`. В Си++ на уровне спецификаций закреплено, что целочисленная константа `0` используется как для обозначения целого числа «ноль», так и для «нулевого указателя», причём вне всякой зависимости от того, каково на данной машине соответствующее «арифметическое» значение адреса. Конечно, макрос `NULL` использовать никто не запрещает (хотя бы из соображений совместимости с Си); правильнее будет сказать, что использовать его *не принято*.

В сравнении с чистым Си язык Си++ имеет ряд дополнительных возможностей, которые можно заметить ещё до введения средств, имеющих отношение к ООП или АТД. Отметим некоторые из них.

Для логических значений в Си++ введён специальный тип `bool`, состоящий из двух значений: `false` и `true`. Конечно, целые числа по-прежнему можно использовать в роли логических значений; более того, целому можно присвоить значение типа `bool`, при этом `false` превратится в `0`, а `true` — в `1`.

Символьные литералы вроде `'a'` или `'7'` в Си++ считаются константами типа `char`, тогда как в чистом Си они считались константами типа `int`. В чистом Си эту особенность семантики вполне можно не заметить, а вот в Си++, как мы увидим позже, конкретный тип константы бывает важен.

Описание (переменных и типов) стало в Си++ *оператором*, что позволяет вставлять его в произвольное место программы, а не только в начало блока; напомним, что в чистом Си нельзя описать переменную после того, как в блоке встретился хотя бы один оператор. Более того, в Си++ переменную можно описать *в заголовке цикла* ***for***, примерно так:

```
for (int i = 0; i < 10; ++i)
```

Наряду с привычными блочными комментариями, которые заключаются в знаки «/*» и «*/», в Си++ введены ***строчные комментарии***, обозначаемые знаком «//»; компилятор проигнорирует весь текст, написанный после такого знака, вплоть до ближайшего символа перевода строки. Есть и другие отличия, о которых вы постепенно узнаете.

10.2. О выборе подмножества Си++

Язык Си++ — это один из самых популярных инструментов современного профессионального программирования, и это обстоятельство накладывает заметный отпечаток как на сам язык, так и на возникшую вокруг него программистскую культуру. Большинство нынешних программистов не мыслит использования Си++ в отрыве от стандартной библиотеки шаблонов (STL); кроме того, сам язык под воздействием стандартизационного комитета и крупных фигур программистского сектора успел изрядно распухнуть и потерять концептуальную целостность. Полное описание всех возможностей языка и его стандартной библиотеки представляет собой книгу такого объёма, что ею вполне можно при желании воспользоваться в качестве оружия против врагов (в смысле чисто механическом); при этом объём материала, как ни странно, ещё не самая большая проблема при изучении Си++. К сожалению, традиционной ныне стала схема, когда в самом начале курса в примерах используются экзотические возможности библиотеки, но не говорится, как всё это устроено; в итоге теряется ощущение границы между возможностями языка и возможностями библиотеки[1]. Если при этом учесть нехватку учебного времени, получается, что до изложения действительно интересных концепций, именно того, чем Си++ отличается от всех остальных существующих языков, дело попросту не доходит.

В 1997 году вышло в свет третье издание книги автора языка Си++ Бьёрна Страуструпа, которая так и называется «Язык программирования Си++». В этом издании Страуструп впервые вводит STL и с самого начала книги пользуется в примерах контейнерными шаблонами, при этом рассказ о шаблонах как таковых расположен в книге ближе к концу. В 1999 году книга вышла в русском переводе. Автору этих строк приходилось сотрудничать (в том числе работать в одном проекте) как с программистами, изучавшими Си++ до выхода вышеупомянутого третьего издания, так и с теми, кто изучал язык позднее. Можно с уверенностью сказать, что восприятие языка Си++ у одних и у других совершенно разное, причём программисты «старой закваски» умеют работать с STL, но могут и обходиться без него, например, ради выигрыша в эффективности, тогда как, не побоюсь этого слова, *жертвы* третьего издания попросту не мыслят Си++ в отрыве от STL и, к величайшему сожалению, часто совершенно не задумываются, что же происходит внутри контейнерных классов. Более того, многие не умеют пользоваться механизмом шаблонов, поскольку STL вызывает ощущение (ошибочное!), что все шаблоны давно написаны.

Текст этой части ранее издавался отдельной книжкой — в 2008 году вышло её первое издание, в 2011 и 2012 гг. она переиздавалась; четвёртое издание вышло только в 2018 году — когда стало ясно, что с четвёртым томом нашего «Введения в профессию» дело затягивается. За шесть лет, прошедших между третьим и четвёртым изданием книжки по Си++, мир несколько изменился: группа международных террористов, по недоразумению называющихся комитетом по стандартизации Си++, развернула весьма бурную и эффективную деятельность по окончательному уничтожению этого языка. Вышедшие последовательно «стандарты» С++11, С++14 и, наконец, С++17 не переставали удивлять публику: каждый раз казалось, что более мрачного и безумного извращения придумать уже нельзя, и каждый раз выход очередного «стандарта» убедительно демонстрировал, что всё возможно; появившийся недавно С++20 как будто специально задуман как наглядное подтверждение, что предела этому процессу нет, разве что Си++ всё-таки окочурится. Если под «языком Си++» понимать С++17 или тем паче С++20, то о применении такого инструмента на практике не может быть никакой речи, т. е. с языком Си++ следует попрощаться, устроить торжественные похороны и поискать альтернативу; впрочем, то же самое можно сказать про все его «стандарты», начиная с самого первого, принятого в 1998 году — строго говоря, язык Си++ как уникальное явление был уничтожен именно тогда.

Проблема, к сожалению, в том, что альтернативы появляться не спешат. Нельзя сказать, что в мире вообще не создаются новые языки программирования, но по каким-то неведомым причинам ни один из этих новых языков просто в силу своего исходного дизайна не может претендовать на роль заменителя Си++ и тем более чистого Си. При этом рвение, с каким «стандартизаторы» последовательно убивают именно эти два языка программирования, заставляет подозревать, что в IT-индустрии здравый смысл как явление окончательно умер.

Примечательно здесь, пожалуй, вот что. Автору этих строк представляется очевидным, что язык Си (чистый Си) смог стать тем, чем он стал, и занять свою нынешнюю нишу благодаря двум очевидным свойствам: во-первых, ясной и жёсткой границе между самим языком и его библиотекой, сколь бы «стандартной» она ни была, и, во-вторых, достижимости *zero runtime*, т. е. возможности использования созданных компилятором объектных модулей без поддержки со стороны поставляемых с компилятором библиотек. В отсутствие любого из этих свойств язык программирования оказывается неприменим для программирования на «голом железе», т. е. для ядер операционных систем и для прошивок микроконтроллеров, и, как следствие, не может претендовать на универсальность. Тем удивительнее, сколь упорно создатели «стандартов» уничтожают оба этих свойства, причём как в Си++, так и в чистом Си.

Подавляющее большинство «современных программистов» реально не видят принципиальной разницы между Си++ и, к примеру, тем же Питоном или Джавой, а все «новые технологии» (включая и новые «стандарты») предпочитают встречать с каким-то нездоровым восторгом, полностью отключив способности к критическому мышлению. К счастью, кроме этого большинства, существует также и меньшинство, думать пока не разучившееся, в противном случае мировая IT-индустрия, скорее всего, уже давно перестала бы приносить хоть какую-то пользу. Сейчас, спустя два десятка лет после получения STL'ем «официального статуса», в числе тех, кто его принципиально не использует, можно обнаружить, например, такого монстра, как проект Qt; но, пожалуй, интереснее выглядит библиотека FLTK, в тексте которой принципиально не применяются не только возможности стандартной библиотеки Си++, но и шаблоны как таковые, и обработка исключений, и даже пространства имён (`namespaces`), не говоря уже о «возможностях» из новомодных «стандартов».

Судя по всему, в условиях, когда язык Си++ изувечен стандартизаторами, адекватных альтернатив не предвидится, а программировать на чём-то всё же нужно, сознательное использование таких усечённых подмножеств остаётся последним и единственным вариантом для тех, кто сохраняет разборчивость в выборе инструментов. Автору остаётся лишь пожелать читателям успехов на этом нетривиальном пути. Как STL, так и «возможности», пришедшие из так называемых «стандартов», в нашей книге сознательно игнорируются.

10.3. Методы, объекты и защита

В этой главе мы попытаемся показать читателю путь постепенного перехода от традиционного императивного стиля мышления к мышлению в терминах объектов и методов. Для этого мы, отталкиваясь от уже известных читателю средств чистого Си, шаг за шагом будем добавлять понятия и средства, пришедшие из области объектно-ориентированного программирования, пока не придём к полноценному ***классу***, имеющему ***конструктор***, ***деструктор*** и ***методы***.

10.3.1. Функции-члены (методы)

Для поддержки объектно-ориентированного программирования в Си++ вводятся понятия ***функций-членов*** и ***классов***. О классах мы расскажем позднее, а пока попытаемся понять, что же такое «функция-член».

Пусть нам потребовалась структура для представления комплексного числа через действительную и мнимую части:

```
struct str_complex {
    double re, im;
};
```

Введём теперь операцию вычисления модуля комплексного числа. На языке Си нам пришлось бы описать примерно такую функцию:

```
double magnitude(struct str_complex *c)
{
    return sqrt(c->re*c->re + c->im*c->im);
}
```

[1] Любителям громких заявлений о неотделимости стандартной библиотеки от языка мы предложим поискать стандартную библиотеку чистого Си, например, в ядре Linux.

Язык Си++ позволяет сделать то же самое несколько более элегантно, подчеркнув непосредственное отношение функции `magnitude` к сущности комплексного числа. Функцию мы внесём *внутрь* структуры, сделав как будто бы её частью:

```
struct str_complex {
    double re, im;
    double magnitude() { return sqrt(re*re + im*im); }
};
```

Обратите внимание, что в теле функции встречаются обращения к полям структуры прямо по именам, без уточнения, откуда такое имя берётся, как если бы это были простые переменные, а не поля. Теперь мы можем написать, например, такой код:

```
str_complex z;
double mod;
z.re = 2.7;
z.im = 3.8;
mod = z.magnitude();
```

Функция `magnitude` называется ***функцией-членом*** или ***методом*** структуры `str_complex`. Иногда речь идёт о *методе объекта* (в данном случае — объекта `z`), при этом имеется в виду метод того типа, к которому принадлежит объект. Как видно из примера, метод вызывается не сам по себе, а *для конкретного объекта*, и как раз из этого объекта берутся поля, когда в теле метода имеет место обращение к полю структуры по имени. В данном случае метод вызван для объекта `z`, так что в его теле упоминания идентификаторов `re` и `im` будут соответствовать полям `z.re` и `z.im`.

Вызов метода — это именно то, что в теории объектно-ориентированного программирования понимается под *отправкой сообщения объекту*. Иначе говоря, **термины «вызов метода» и «передача сообщения» являются синонимами**. Мы можем, следовательно, сказать, что в последней строчке вышеприведённого фрагмента кода мы *передали объекту* `z` *сообщение* ***`magnitude`***, означающее «посчитай свой модуль»; объект ответил на наше сообщение, причём в ответе содержалось число типа `double`, равное искомому модулю; получив этот ответ, мы присвоили его переменной `mod`.

10.3.2. Неявный указатель на объект

На самом деле вызов функции-члена (метода) объекта с точки зрения реализации представляет собой абсолютно то же самое, что и вызов обычной функции, первым параметром которой передаётся адрес объекта. В частности, в предыдущем параграфе мы заменили внешнюю функцию `magnitude`, получавшую адрес структуры, на метод `magnitude`, описанный внутри структуры, а вызов с явной передачей адреса — на вызов метода для объекта; при этом *на уровне машинного кода никаких изменений не произошло*. Говорят, что *функциям-членам при вызове их для объекта передаётся* ***неявный параметр*** — адрес объекта, для которого функция вызывается.

К этому параметру при желании можно обратиться явно; именем ему служит ключевое слово `this`. Можно воспринимать `this` как локальную константу, имеющую тип `A*`, где `A` — имя описываемой структуры. Например, описанную в предыдущем параграфе версию структуры можно было бы переписать так:

```
struct str_complex {
    double re, im;
    double magnitude() {
        return sqrt(this->re*this->re + this->im*this->im);
    }
};
```

В данном конкретном случае это не имеет особого смысла, однако бывают такие ситуации, в которых использование `this` оказывается необходимым; достаточно представить себе, что из метода нужно вызвать какую-либо функцию (обычную или метод другого объекта), аргументом которой должен стать как раз наш собственный объект, тот, для которого нас вызвали.

10.3.3. Защита. Понятие конструктора

Чтобы стать полноценным объектом, нашей структуре не хватает свойства закрытости. В самом деле, поля `re` и `im` доступны из любого места в программе, где доступна сама структура `str_complex`. Для того, чтобы скрыть детали реализации объекта, в языке Си++ введён механизм ***защиты***, который позволяет запретить доступ к некоторым частям структуры (как полям, так и функциям-методам) из любых мест программы, кроме тел функций-методов.

Для поддержания этого механизма в язык введены ключевые слова `public` и `private`, которыми в описании структуры могут быть помечены поля и методы, доступные извне структуры (`public:`) и, наоборот, доступные только из тел функций-методов (`private:`)[2]. Попытаемся переписать нашу структуру с использованием этих ключевых слов:

```
struct str_complex {
private:
    double re, im;
public:
    double magnitude() { return sqrt(re*re + im*im); }
};
```

Теперь поля `re` и `im` доступны только из тела метода `magnitude`. Легко заметить, однако, что пользоваться такой структурой мы не сможем, ведь в ней не предусмотрено никаких средств для задания значений полей `re` и `im`. Так, фрагмент кода, приведённый на стр. 36, попросту будет отвергнут компилятором, ведь поля теперь недоступны, в том числе и для присваивания. Решить проблему можно, введя соответствующий метод для задания значений полей. Описание нашей структуры могло бы выглядеть в этом случае, например, так:

```
struct str_complex {
private:
    double re, im;
public:
    void set(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double magnitude()
        { return sqrt(re*re + im*im); }
};
```

а пример использования объекта мы теперь перепишем так:

```
str_complex z;
double mod;
z.set(2.7, 3.8);
mod = z.magnitude();
```

У этого решения есть недостаток, притом настолько серьёзный, что компиляторы обычно выдают предупреждение при попытке компиляции такого объявления структуры. Дело в том, что с момента объявления переменной `z` до вызова функции `set` наш объект (переменная `z`) оказывается в *неопределённом* состоянии, т. е. попытки его использовать будут заведомо ошибочны. Программист должен будет об этом помнить, что, помимо прочего, противоречит базовому принципу ООП: вне объекта не следует строить предположений о его внутреннем состоянии, и для «неопределённого» состояния вряд ли стоит делать исключение. Очевидно, было бы лучше произвести инициализацию объекта непосредственно в момент его создания. Более того, было бы логично *запретить* создание объекта в обход инициализации.

Язык Си++ позволяет задавать компилятору чёткие инструкции относительно действий, необходимых (в том числе и в буквальном смысле — то есть таких, которые *нельзя обойти*) при инициализации объекта. Для этого вводится ещё одно важное понятие — ***конструктор объекта***. Конструктор — это функция-метод специального вида, ко-

[2] Несколько позже мы введём ещё и ключевое слово `protected`.

торая может, как и обычная функция, иметь параметры или не иметь их; тело этой функции как раз и представляет собой порядок действий, которые необходимо выполнить всякий раз, когда создаётся объект описываемого типа. Написав конструктор, мы тем самым «объясняем» компилятору, как — в соответствии с какой инструкцией — создавать новый объект данного типа, какие параметры должны быть для этого заданы и как воспользоваться значениями этих параметров.

Компилятор отличает конструкторы от обычных методов по имени, которое совпадает с именем описываемого типа (в данном случае структуры). Поскольку конструктор играет специальную роль и в явном виде обычно не вызывается[3], тип возвращаемого значения для конструктора указывать нельзя, он не возвращает никаких значений; в каком-то смысле результатом работы конструктора является сам объект, для которого его вызвали.

Проиллюстрируем сказанное, переписав структуру `str_complex`:

```
struct str_complex {
private:
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double magnitude()
        { return sqrt(re*re + im*im); }
};
```

Введя конструктор (функцию-метод с именем `str_complex`), мы сообщили компилятору, что для создания объекта типа `str_complex` нужно знать два числа типа `double`, и задали набор действий, подлежащих выполнению при создании такого объекта. В этом наборе действий говорится в том числе и о том, как воспользоваться заданными числами типа `double`: первое из них использовать в качестве действительной части, второе — в качестве мнимой части создаваемого комплексного числа.

Вообще в семантике Си++ любая переменная составного (структурного) типа создаётся с помощью конструктора. Во многих случаях компилятор считает конструктор существующим («неявно»), несмотря на то, что в программе конструктор не описан.

Код вычисления модуля теперь можно переписать вот так:

```
str_complex z(2.7, 3.8);
double mod;
mod = z.magnitude();
```

Более того, для такой операции нам не обязательно описывать переменную, имеющую имя. Мы могли бы написать и так:

```
double mod = str_complex(2.7, 3.8).magnitude();
```

Здесь мы создали ***анонимную переменную*** типа `str_complex` и для этой переменной вызвали метод `magnitude`. Анонимным переменным мы позже посвятим отдельный параграф.

Сделаем важное замечание. После введения конструктора, имеющего параметры, компилятор откажется создавать объект типа `str_complex` без указания требуемых конструктором двух значений, то есть предыдущая версия кода, содержавшая описание

```
str_complex z;
```

компилироваться больше не будет. Если это создаёт неудобства, можно заставить компилятор снова считать такие объявления корректными; о том, как это делается, мы узнаем из §10.4.3.

Синтаксис создания переменной по заданному параметру допустим в Си++ и для переменных встроенных типов. Так, описание

```
int v(7);
```

означает абсолютно то же самое, что и привычное по языку Си описание переменной с *инициализатором*

```
int v = 7;
```

Напомним на всякий случай, что ***инициализация*** — это совсем не то же самое, что ***присваивание***[4]. В чистом Си инициализация и присваивание обозначаются одним и тем же символом — знаком равенства, но разница между ними становится очевидной, если вспомнить, что мы можем, например, задать начальное значение для массива, тогда как *присваивать* массивы нельзя. Кроме того, стоит заметить, что для глобальных переменных инициализация не требует никаких действий, выполняемых процессором, ведь при старте программы в память загружается образ, уже содержащий нужные значения; присваивание же, очевидно, всегда представляет собой именно действие.

В Си++ разница между инициализацией и присваиванием ещё более существенна, и мы в этом неоднократно убедимся. В частности, важно понимать, что конструкторы используются при инициализации и не имеют никакого отношения к присваиванию. Как мы узнаем из дальнейшего изложения, Си++ позволяет взять под контроль не только инициализацию объекта, но и присваивание ему значений, но для этого предусмотрены отдельные средства, конструкторы тут ни при чём.

Возвращаясь к обсуждению защиты, отметим ещё один немаловажный момент. **Единицей защиты в Си++ является не объект, а тип (в данном случае структура) целиком**. Это означает, что из тел методов мы можем обращаться к закрытым полям не только «своего» объекта (того, для которого вызван метод), но и вообще любого объекта того же типа.

10.3.4. Зачем нужна защита

Смысл механизма защиты часто оказывается непонятен программистам, начинающим осваивать объектно-ориентированное программирование. Попробуем пояснить его, основываясь на нашем примере.

Представим себе, что наша структура `str_complex` используется в большой программе, активно работающей с комплексными числами. Может случиться так, что в программе будет очень часто требоваться вычисление модулей комплексных чисел. Более того, может оказаться и так, что именно модуль комплексного числа требуется нам даже чаще, чем его действительная и мнимая части. В такой ситуации наша программа будет проводить значительную часть времени своего выполнения в вычислениях модулей. Обнаружив это, мы можем прийти к выводу, что в данной конкретной задаче удобнее хранить комплексные числа в полярных координатах, а не в декартовых, то есть в виде модуля и аргумента, а не в виде действительной и мнимой частей.

Если мы не применяли защиту, то, скорее всего, все модули нашей программы, в которых используются комплексные числа, содержат обращения к полям `re` и `im`. Если в такой программе изменить способ хранения комплексного числа, убрав поля `re` и `im` и введя вместо них, скажем, поля `mod` и `arg` для хранения модуля и аргумента (т. е. полярных координат), то все части программы, использовавшие нашу структуру, перестанут компилироваться и нам придётся их исправлять. Если программа достаточно большая (а современные программные проекты часто состоят из многих сотен и даже тысяч модулей), такое редактирование может потребовать существенных трудозатрат, приведёт к внесению в программу новых ошибок и т. п., так что, вполне возможно, нам придётся отказаться от изменений, несмотря на всю их полезность.

Допустим теперь, что мы использовали защиту и сделали все поля недоступными откуда бы то ни было, кроме методов нашей структуры. Конечно, при использовании комплексных чисел часто бывает нужно узнать отдельно действительную и мнимую части числа, но для этого можно ввести специальные методы; для полноты картины введём заодно функцию вычисления аргумента, которая будет использовать стандартную функцию `atan2` для определения соответствующего арктангенса:

---

[3] На самом деле Си++ позволяет вызвать конструктор явным образом, но если вам это потребовалось — скорее всего, вы делаете что-то очень странное; мы рассматривать эту возможность не будем.

[4] См. рассуждение на стр. 12.

```
struct str_complex {
private:
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double get_re() { return re; }
    double get_im() { return im; }
    double magnitude() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
};
```

К полям `re` и `im` теперь не могут обращаться никакие части программы, кроме методов структуры, так что при переходе от хранения действительной и мнимой части к хранению полярных координат придётся переписывать только наши собственные методы, а весь остальной текст программы, из скольких бы модулей он ни состоял и что бы ни делал с комплексными числами, сохранится без изменений и будет работать как раньше. Конечно, методы `get_re` и `get_im` теперь станут гораздо сложнее, чем были, зато упростятся методы `magnitude` и `argument`:

```
struct str_complex {
private:
    double mod, arg;
public:
    str_complex(double re, double im) {
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re() { return mod * cos(arg); }
    double get_im() { return mod * sin(arg); }
    double magnitude() { return mod; }
    double argument() { return arg; }
};
```

В такой ситуации мы даже можем поддерживать две реализации структуры `str_complex`, между которыми выбор осуществляется директивами условной компиляции. Можно будет, не меняя текста программы, откомпилировать её с использованием одной реализации, измерить быстродействие, откомпилировать программу с использованием другой реализации, снова измерить быстродействие, сравнить результаты измерений и принять решение, какую реализацию задействовать. Если используемые в программе алгоритмы изменятся и нам снова станет выгодно хранить комплексные числа в декартовых координатах, то для возврата к старой модели вообще не придётся ничего редактировать.

Более того, мы можем в некий момент решить, что и те, и другие вычисления производить довольно накладно, а памяти нам не жалко, и начать хранить в объекте как декартово, так и полярное представление числа:

```
struct str_complex {
private:
    double re, im, mod, arg;
public:
    str_complex(double a_re, double a_im) {
        re = a_re; im = a_im;
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re() { return re; }
    double get_im() { return im; }
    double magnitude() { return mod; }
    double argument() { return arg; }
};
```

Теперь в нашем объекте четыре поля, причём каждую из двух пар можно вычислить, пользуясь значениями двух других, то есть значения наших полей не могут быть произвольными, а должны всегда находиться в некотором соотношении: в данном случае пары (`re`, `im`) и (`mod`, `arg`) должны задавать одно и то же комплексное число, хотя и разными способами. В таких случаях говорят, что хранимая информация ***избыточна***; в этом нет совершенно ничего страшного, если только мы можем обеспечить ***целостность*** информации, то есть гарантировать, что в нашей программе значения рассматриваемых полей всегда будут находиться в зафиксированном для них соотношении. Для обычной (открытой) структуры нам было бы тяжело это гарантировать, поскольку в любом месте программы мог бы появиться оператор, изменяющий только некоторые из взаимосвязанных значений и не изменяющий другие; если программа имеет существенный объём, и в особенности если над ней работают несколько программистов, уследить за правильностью использования такой структуры может оказаться проблематично. Когда же все поля закрыты, доступ к ним имеют только методы описываемой структуры, обычно находящиеся в рамках одного модуля. Объём такого модуля, как правило, невелик в сравнении со всей программой, ну а другие программисты либо вовсе не станут редактировать «чужой» модуль, либо, если такая необходимость всё же возникнет, для начала разберутся, как этот модуль должен функционировать.

Очень важно понять, что защита в языке Си++ предназначена не для того, чтобы защищать нас от врагов, но исключительно для защиты нас от самих себя, от собственных ошибок. Если задаться целью обойти механизм защиты, это можно сделать без особых проблем путём преобразования типов указателей или другими способами. Защита работает только в случае, если программисты не предпринимают целенаправленных действий по её обходу. Но обычно программисты этого не делают, поскольку понимают полезность механизма защиты и выгоды от его использования, а также и то, что попытки его обойти, скорее всего, приведут к внесению в программу ошибок.

10.3.5. Классы

Поскольку большинство внутренних полей объектов обычно закрыты, для описания объектов в Си++ используют специально введённый для этой цели тип составных переменных, называемый ***классом***. Класс — это тип переменной, напоминающий запись (структуру), но отличающийся тем, что к полям (членам) класса доступ по умолчанию есть только из методов самого этого класса.

Перепишем нашу реализацию комплексного числа в виде класса:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double GetRe() { return re; }
    double GetIm() { return im; }
    double Magnitude() { return sqrt(re*re + im*im); }
    double Argument() { return atan2(im, re); }
};
```

Всё, что описано в классе до слова `public:`, компилятор рассматривает как детали реализации класса и запрещает доступ к ним отовсюду, кроме тел функций-членов данного класса. Слово `public:` меняет режим защиты с закрытого на открытый, т. е. всё, что описано после этого слова (в данном случае это функции-члены `Complex`, `Magnitude` и др.) будет доступно во всём тексте программы.

Важно помнить, что **модель защиты, включаемая по умолчанию — это единственное отличие классов от структур**. Больше они ничем друг от друга не отличаются, по крайней мере, с точки зрения компилятора. Тем не менее обычно программисты используют ключевое слово `struct` для построения обычных открытых структур данных — таких же, какие применяются в чистом Си; если же задействуется что-то из того, чем Си++ отличается от Си — *объект* в смысле объектно-ориентированного программирования или некий абстрактный тип данных, относящийся к парадигме АТД — то применяют классы, чтобы показать читателю программы, что́ имеется в виду.

10.3.6. Деструкторы

Наряду с конструкторами, контролирующими процесс *создания* (инициализации) объектов, в языке Си++ предусмотрены также деструкторы, предназначенные для контроля над процессом *уничтожения* объекта.

Представим себе, что объект в ходе своей деятельности захватывает некий ресурс — например, открывает файл, выделяет динамическую память и т. п., причём об этом известно только самому объекту, т. е. на захваченный ресурс ссылаются только закрытые поля объекта (в примере с файлом это может означать, что дескриптор открытого файла хранится в закрытом поле объекта). Заметим теперь, что объект может в любой момент (неожиданно для себя) прекратить существование. Так, если объект — обычная локальная переменная в функции, то она исчезнет при возврате управления из этой функции; если создать объект в динамической памяти, то он перестанет существовать при освобождении этой памяти; даже если объект сделать глобальной переменной, то завершение программы тоже никто не отменял, при этом изредка встречаются и такие ресурсы, которые не освобождаются автоматически, даже когда завершается программа, захватившая их. Но ведь о том, что наш объект что-то там захватил, *никто, кроме него, не знает*! Как следствие, только сам объект может освободить всё, что он себе забрал, так что ему жизненно необходима возможность «привести свои дела в порядок» перед исчезновением, когда бы и по каким бы причинам это исчезновение ни случилось.

Именно для этого предназначаются деструкторы. ***Деструктор*** — это метод класса, вызов которого автоматически вставляется компилятором в код в любой ситуации, когда объект прекращает существование. Имя функции-деструктора образуется из имени описываемого типа (класса или структуры) добавлением спереди знака `~` (тильда); например, для класса `A` деструктор будет называться `~A`. Список параметров функции-деструктора всегда пуст, т. к. в языке отсутствуют средства передачи параметров деструктору. Деструктор играет специальную роль и, как и конструктор, в явном виде обычно не вызывается[5]; тип возвращаемого значения для деструктора не указывается — деструктор никогда не возвращает никаких значений.

Проиллюстрируем понятие деструктора на примере класса `File`, инкапсулирующего дескриптор файла[6]:

```
class File {
    int fd; // Дескриптор -1 означает, что файл не открыт
public:
    File() { fd = -1; }
        // На момент создания объекта мы ещё ничего не открыли
    bool OpenRO(const char *name) {
        fd = open(name, O_RDONLY);
        return fd != -1;
    }   // метод пытается открыть файл на чтение,
        // возвращает true в случае успеха,
        // false в случае неудачи

    // ... методы работы с файлом ...

    ~File() { if(fd != -1) close(fd); }
         // Деструктор закрывает файл, если он открыт
};
```

Деструктор будет вызван в любой ситуации, когда объект типа `File` прекращает существование. Например, если объект был описан как локальный в функции, то при возврате из функции (в том числе и досрочном вызове оператора `return`) для этого объекта отработает деструктор. Вообще **при создании объекта ровно один раз отрабатывает конструктор, при уничтожении объекта ровно один раз отрабатывает деструктор**. За выполнением этого правила следит компилятор.

10.4. Абстрактные типы данных в Си++

Защита вместе с конструкторами и деструкторами, которые мы рассмотрели в предыдущей главе, используется как для создания *объектов* в терминах объектно-ориентированного программирования, так и для синтеза абстрактных типов данных; что касается методов (функций-членов), то для ООП они обязательны, тогда как для абстрактных типов данных это лишь один из возможных способов описания набора операций над вводимым типом.

В этой главе мы рассмотрим средства, которые вообще не имеют никакого отношения к парадигме объектно-ориентированного программирования, но это не делает их менее важными. Возможность ввести свои версии арифметических операций для пользовательских типов данных, контроль объектов за собственным созданием, копированием, присваиванием и ликвидацией, уникальная концепция типа-ссылки, позволяющая вынести в систему типов понятие *леводопустимого выражения* — всё это практически с самого начала стало визитной карточкой языка Си++.

Рассматривая все эти инструменты единым комплексом, мы можем заметить, что главный достигнутый результат здесь — это возможность для программиста ввести сколь угодно сложную абстракцию и работать с ней так же, как мы привыкли работать с элементарными встроенными типами, при желании полностью игнорируя детали реализации. Имеющиеся в Си++ средства создания абстрактных типов данных обладают выразительной мощью, близкой к принципиально возможному максимуму.

При правильной организации программы трудоёмкость её написания на Си++ может оказаться в разы ниже, чем создание такой же программы на чистом Си, и основной выигрыш по трудозатратам здесь дают именно средства генерации абстрактных типов данных, рассматриваемые ниже.

10.4.1. Перегрузка имён функций; декорирование

Для понимания целого ряда возможностей, которые мы будем постепенно вводить в следующих параграфах, нужно иметь представление о свойстве языка Си++, называемом ***перегрузкой имён функций***. Сразу оговоримся, что эта возможность не имеет никакого отношения ни к ООП, ни к АТД и вообще к каким бы то ни было парадигмам; это сугубо техническое решение, позволившее снять некоторые проблемы, которых вообще-то можно было изначально не допускать; как мы вскоре увидим, перегрузка функций сама порождает достаточно ощутимые технические проблемы.

Перегрузка имён состоит в том, что в рамках одной области видимости Си++ позволяет ввести *несколько* различных функций, имеющих одно имя, но предполагающих разное количество и/или типы параметров. Например, будет вполне корректен следующий код:

```
void print(int n) { printf("%d\n", n); }
void print(const char *s) { printf("%s\n", s); }
void print() { printf("Hello world\n"); }
```

При обработке вызова функции компилятор определяет, какую функцию с таким именем следует вызвать, используя количество и типы фактических параметров. Например:

```
print(50);                 // вызывается print(int)
print("Have a nice day"); // вызывается print(const char*)
print();                   // версия без параметров
```

Два типа, отличающихся только наличием или отсутствием модификатора `const`, с точки зрения механизма перегрузки всё-таки считаются различными, так что компилятор позволяет в одной области видимости ввести, например, следующее:

[5] В сноске 3 на стр. 37 мы упоминали, что Си++ позволяет вызвать конструктор явно, но делать этого не надо. То же самое относится и к деструкторам.

[6] Здесь, как и в предыдущих частях книги, мы снабжаем примеры программного текста комментариями на русском языке. В учебном пособии мы можем себе позволить такую вольность, но помните, что **в текстах реальных программ допустим только один язык — английский**, а русские буквы, как и вообще любые символы, не входящие в набор ASCII, в программах встречаться не должны, и к комментариям это тоже относится.

```
void func(int *p) { /* ... */ }
void func(const int *p) { /* ... */ }
```

Имя функции вместе с типами всех её параметров называют ***профилем функции***. Возможность перегрузки имён функций означает, что **в языке Си++, в отличие от чистого Си, функции идентифицируются не именами, а профилями**. Стоит иметь в виду, что **тип возвращаемого значения не считается частью профиля функции**, т. е. нельзя ввести две функции, заголовки которых различаются только типом возвращаемого значения. При этом **перегруженные функции могут возвращать значения разных типов**, это ничему не противоречит; технически возможно, например, сотворить что-нибудь вроде

```
int plus5(int x) { return x + 5; }
void plus5(int *x) { *x += 5; }
```

— если, конечно, вы сможете объяснить хотя бы сами себе, зачем это делать.

Введение перегруженных функций может привести к совершенно неожиданным последствиям. Так, следующий код сам по себе корректен:

```
void f(const char *str)
{
    printf("This is a string: %s\n", str ? str : "(null)");
}
void f(float f)
{
    printf("This is a floating point number: %f\n", f);
}
```

Компилятор вполне справится с вызовами `f("string")` и `f(2.5)`, поскольку никаких разночтений тут не возникает. Более того, даже вызов `f(1)` будет успешно откомпилирован, так как целое можно неявно преобразовать к типу `float`, но не к типу `const char*`. Однако вызов `f(0)` будет расценён компилятором как ошибочный, поскольку компилятор с совершенно одинаковым успехом может рассматривать `0` и как константу типа `float`, и как адресную константу («нулевой указатель»), а оснований выбрать тот, а не иной вариант у него нет. Между тем, если бы в программе присутствовала только одна из двух вышеописанных функций `f` (любая!), вызов `f(0)` был бы заведомо корректен.

Отметим один интересный казус. Формально говоря, перегрузка функций есть одно из проявлений ***статического полиморфизма***, причём, в отличие от случаев, которые мы упоминали на стр. 16, здесь мы имеем полиморфизм, вводимый пользователем, а не встроенный в язык. И тем не менее ни к объектно-ориентированному программированию, ни к абстрактным типам данных перегрузка прямого отношения не имеет.

Надо сказать, что перегрузка функций в общем случае — возможность, мягко говоря, сомнительная. В обычных обстоятельствах нет никакого смысла «экономить» идентификаторы; если нам зачем-то потребовалось ввести три «родственные» функции для печати, как в примере выше, никто не мешает сделать это без всякой перегрузки, например, так:

```
void print_int(int n) { printf("%d\n", n); }
void print_str(const char *s) { printf("%s\n", s); }
void print_hello() { printf("Hello world\n"); }
```

В Си++ перегрузка была введена, как говорят в таких случаях, *ad hoc*, чтобы справиться с двумя вполне конкретными проблемами, причём обе эти проблемы возникли (а если называть вещи своими именами — *были созданы*) практически на ровном месте. Про одну из них читатель, возможно, уже догадался. В §10.3.3 мы ввели понятие ***конструктора***, указав, что это метод специального вида, задающий *инструкцию, как создавать объекты данного типа*; там же мы отметили, что компилятор отличает конструкторы от обычных методов по имени, которое совпадает с именем описываемого типа. Но что если объект одного типа нам захочется в зависимости от обстоятельств создавать разными способами? Механизм перегрузки позволяет ввести для одного типа несколько разных конструкторов, отличающихся принимаемыми параметрами, и, как мы увидим чуть позже, в Си++ эта возможность эксплуатируется в хвост и в гриву.

Вторая причина введения перегрузки функций — точнее, «проблема», которую введение перегрузки позволило снять — станет ясна из текста следующего параграфа, в котором мы рассмотрим переопределение символов арифметических операций для типов, введённых пользователем. В Си++ для такого переопределения в программе описываются функции с именами специального вида, которые приходится различать по типам аргументов.

Рискнём заявить, что обе проблемы проще было изначально не создавать, особенно с учётом того, к каким несуразно тяжёлым последствиям в итоге привело введение перегрузки. До определённого момента на эти последствия (явление, называемое «декорированием имён» или «манглингом») можно не обращать внимания, так что их описание, идущее ниже, набрано уменьшенным шрифтом. Но прежде дадим читателю одну вполне универсальную рекомендацию: **будьте готовы к тому, что перегрузка встретится вам в чужом коде, но сами никогда не применяйте её для обычных функций**. Случаи конструкторов и переопределённых операций не оставляют иного выбора; во всех остальных ситуациях перегрузка имён не нужна и только затрудняет восприятие программы.

Появление в Си++ перегрузки имён функций ведёт к определённым трудностям при компоновке программ из модулей, написанных на разных языках — или, если говорить честно, при использовании в одной программе модулей, написанных на Си++, на чистом Си и (гораздо реже) на языке ассемблера[7]. Если в ваши планы не входит использование отдельных модулей на чистом Си, можете пропустить остаток этого параграфа — но лучше прочитайте, чтобы хотя бы примерно знать, где вас могут поджидать очередные грабли.

Как мы знаем (см. т. 1, §3.7.3), объектные файлы, получающиеся в результате компиляции из отдельных модулей, содержат образы областей памяти, помеченные ***метками***, и указания для редактора связей, что в определённые места нужно подставить итоговые адреса, которые получатся из меток, содержащихся в других модулях. Редактор связей по сути примитивен, он не знает ничего не только о перегрузке имён функций, но и вообще о функциях, для него функция — это просто образ области памяти. Компиляторы чистого Си в качестве метки для кода функции обычно используют имя этой функции (а для глобальных переменных — имена этих переменных); компилятор Си++ поступает так же с глобальными переменными, но не с функциями, ведь «благодаря» перегрузке имён в программе может оказаться больше одной функции с заданным именем.

Здесь на сцене появляется очередной монстр, который по-английски называется *name mangling*; наиболее часто это переводится на русский словосочетанием «***декорирование имён***», которое, увы, совершенно не отражает сути. Между прочим, слово *mangling* правильнее было бы перевести как «коверканье» — именно так оно переводится в других (непрограммистских) контекстах. Программисты, разумеется, в повседневной речи используют транслитерацию «*манглинг*», и труднопроизносимость этого слова их не останавливает.

Как бы мы ни называли это явление, суть его в том, что для обозначения функции в объектном файле компилятор использует не её имя, а некое неудобоваримое чёрт-те что, содержащее закодированную информацию обо всём профиле функции, то есть о её имени и типах всех её параметров. Например, имя функции

```
double foo(double a, int b);
```

наш компилятор g++ превратит в «`_Z3foodi`»; здесь `_Z` — это «волшебный» префикс, обозначающий «декорированное» имя, затем следует десятичное число, обозначающее длину исходного имени функции (в нашем случае 3, поскольку имя `foo` состоит из трёх букв), потом идёт собственно имя, а за ним — закодированные типы параметров. Для встроенных типов всё довольно просто — как мы видим, `double` превратилось в букву `d`, `int` — в `i`; с пользовательскими типами компилятору приходится изворачиваться сильнее — например, параметр типа `const struct str1*` превратился бы в «`PK4str1`». Забегая вперёд, отметим, что декорированное имя функции, параметрами которой служат типы, инстанциированные из шаблонов, обычно очень хочется *развидеть*.

Пикантности ситуации добавляет ещё и то, что разные компиляторы (а в некоторых особо патологических случаях — даже разные версии одного компилятора) могут использовать различные алгоритмы декорирования — разумеется, несовместимые между собой. Впрочем, в этом плане в последние годы наметился определённый прогресс — во всяком случае, g++ не меняет схему декорирования, начиная с версий 3.*, и ту же

[7] Говорят, иногда в смешанных проектах применяются ещё модули на Фортране, но автор книги такого не встречал.

самую схему используют несколько других компиляторов, включая, например, clang. Так или иначе, **библиотеки, написанные на Си++, можно переводить в объектный код только для использования с тем же компилятором**, совместимости с другими компиляторами никто пообещать не может (хотя иногда она и есть). Эта проблема никак не затрагивает тех (несомненно разумных) людей, которые используют и распространяют библиотеки исключительно в форме исходного кода, ну а тех (не очень хороших) людей, кто по каким-то причинам исходные тексты своих библиотек распространять не хочет, заставляет собирать объектные варианты своих библиотек под каждый компилятор; следует признать, что и в этом ничего слишком трудного нет, ведь компиляторов Си++ не так много.

Хуже другое: видя *один и тот же заголовок функции*, компиляторы Си и Си++ используют *разные* имена для объектного кода этой функции. Если взять модуль, написанный на чистом Си и содержащий функцию вроде приведённой выше `foo`, откомпилировать его компилятором чистого Си, а его заголовочный файл, содержащий заголовок той же самой функции, подключить с помощью `#include` из модуля, написанного на Си++, компилируемого компилятором Си++ и при этом содержащего обращение к функции `foo`, итоговая программа не пройдёт компоновку. В самом деле, в объектном модуле, полученном компилятором чистого Си, функция будет называться просто `foo`, тогда как компилятор Си++ построит обращение к ней через имя `_Z3foodi`; редактор связей попытается найти в предоставленных ему файлах такую метку, но, понятное дело, не найдёт.

К счастью, Си++ предусматривает решение для этой проблемы. Если заголовки некоторых функций заключить в тело директивы `extern "C"` примерно так:

```
extern "C" {
// ...
double foo(double a, int b);
// ...
}
```

— то для этих функций компилятор будет применять соглашения, характерные для Си; на самом деле конкретный язык задаётся строковым литералом после слова `extern`, в данном случае `"C"`; впрочем, большинство компиляторов поддерживает только один такой «внешний» язык — Си. Декорирование имён для этих функций будет отключено. Отметим, что тело директивы `extern` обычно не снабжают структурным отступом, поскольку во многих случаях это тело охватывает весь заголовочный файл.

Основных способов применения директивы `extern "C"` два. Если автор модуля, написанного на Си, о нас не позаботился, проще всего будет заключить в `extern "C"` *саму директиву `#include`*, подключающую заголовочный файл, так что в результате компилятор увидит всё его содержимое уже внутри `extern "C"`. Выглядит это примерно так:

```
extern "C" {
#include "foo.h"
}
```

Если же мы пишем модуль на чистом Си, но исходно предполагаем, что он будет использоваться в программах, написанных как на Си, так и на Си++, будет вполне осмысленно в начале и в конце заголовочного файла разместить начало и окончание директивы `extern "C"`. Естественно, нужно позаботиться о том, чтобы их видел только компилятор Си++, а компилятор чистого Си не видел — в чистом Си такой директивы, собственно говоря, нет. Но и этот вопрос легко решается благодаря макросимволу `__cplusplus`, который всегда определён, когда используется компилятор Си++. Заголовочный файл `foo.h` с учётом защиты от повторного включения (см. т. 2, §4.7.4) может выглядеть примерно так:

```
#ifndef FOO_H_SENTRY
#define FOO_H_SENTRY

#ifdef __cplusplus
extern "C" {
#endif

// ...

double foo(double a, int b);

// ...

#ifdef __cplusplus
}
#endif

#endif
```

К декорированию имён компилятор вынужден прибегать не только из-за перегрузки функций, но и из-за имён, вложенных в *области видимости*; позже мы увидим, что такими областями являются, например, классы и структуры: их члены имеют сложную форму имён. Однако с этой проблемой можно было бы справиться, не ломая совместимость с Си, когда речь идёт о *простых функциях* (не методах классов). Например, метод `f` из класса `C` мог бы на уровне объектного кода иметь имя `C@@f` или даже просто `C::f`, если линкер допускает двоеточие в именах меток (впрочем, обычно это не так).

Перегрузка имён функций в этом плане кажется довольно неудачной идеей. Как мы уже отмечали выше, введена она в основном по двум причинам: во-первых, чтобы иметь возможность описать в одном классе или структуре *несколько конструкторов*, а во-вторых, как мы будем обсуждать позже, чтобы можно было перегружать символы стандартных операций для типов, введённых пользователем, для чего потребовалось позволить *нескольким* разным функциям иметь одно и то же имя, например, `operator+`. Но обе эти проблемы могли бы быть решены без введения перегрузки: никто не мешает (теоретически) давать конструкторам разные имена, например, просто помечая их специальным словом `constructor` (или каким-нибудь ещё), а функции-операции вроде `operator+` могли бы быть не функциями, а чем-то вроде макросов или специальных директив, либо можно было бы потребовать, чтобы функции с именами такого вида всегда были «подставляемыми» (`inline`) и не были бы, как следствие, видны линкеру; если операция предполагает сложную реализацию, её можно было бы выполнить в функции с обычным именем, а из `operator+` её просто вызвать.

Мы в очередной раз видим, что язык Си++ не идеален; увы, достойная альтернатива ему не спешит появляться. Так или иначе, уроки эпохи Си++ можно было бы учесть при создании нового языка программирования — хотя, судя по происходящему в мире IT (особенно по новым «стандартам» Си++), рассчитывать на это в ближайшие годы не приходится.

10.4.2. Переопределение символов стандартных операций

Вернёмся к нашему примеру — классу, описывающему комплексное число. В реальной задаче мы вряд ли захотим ограничиться только операциями взятия модуля и разложения на компоненты (действительную и мнимую части, модуль и аргумент). Скорее всего, нам потребуются также сложение, вычитание, умножение и деление.

Язык Си++ предоставляет возможности записи таких действий с помощью привычных символов арифметических операций. Кроме того, логично предусмотреть возможность узнать действительную и мнимую части комплексного числа, если вдруг нам это понадобится. Для этого дополним наш класс ещё несколькими функциями-членами, имена которых будут выглядеть довольно необычно. Эти функции как раз и будут описывать действия, которые должен, по нашему замыслу, означать символ той или иной арифметической операции. Отметим, что методы в этом примере будут обращаться к закрытым полям не только «своего» объекта, но и объекта, переданного как параметр. Это не нарушает защиту: как уже говорилось на стр. 37, единицей защиты является не объект, а класс или структура как таковые. Итак, пишем:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double Magnitude() { return sqrt(re*re + im*im); }
    double Argument() { return atan2(im, re); }
    double GetRe() { return re; }
    double GetIm() { return im; }
    Complex operator+(Complex op2) {
        Complex res(re + op2.re, im + op2.im);
        return res;
    }
    Complex operator-(Complex op2) {
        Complex res(re - op2.re, im - op2.im);
        return res;
    }
    Complex operator*(Complex op2) {
        Complex res(re*op2.re - im*op2.im,
                    re*op2.im + im*op2.re);
        return res;
    }
    Complex operator/(Complex op2) {
         double dvs = op2.re*op2.re+op2.im*op2.im;
         Complex res((re*op2.re + im*op2.im)/dvs,
                     (im*op2.re - re*op2.im)/dvs);
         return res;
    }
};
```

Поясним, что слово `operator` является зарезервированным (ключевым) словом языка Си++. Стоящие подряд две лексемы `operator` и `+` образуют *имя функции*, которую можно вызвать и как обычную функцию:

```
a = b.operator+(c);
```

— однако, в отличие от обычной функции, появление функции-операции позволяет записать то же самое в более привычном для математика виде:

```
a = b + c;
```

Теперь мы можем, к примеру, узнать, каков будет модуль суммы двух комплексных чисел:

```
Complex c1(2.7, 3.8);
Complex c2(1.15, -7.1);
double m = (c1+c2).Magnitude();
```

Как мы увидим позже, аналогичным образом в языке Си++ можно переопределить символы любых операций, включая присваивания, индексирование, вызов функции и прочую «экзотику». Существуют только две[8] операции, которые нельзя переопределить: это тернарная ***условная операция*** (`a ? b : c`) и операция выборки поля из структуры или класса (точка). Интересно, что операция выборки поля по указателю («стрелка») переопределяема, хотя и несколько странным способом. Ко всем этим вопросам мы вернёмся позже.

Несомненно, возможность вводить арифметические операции для пользовательских типов — это ещё одно проявление ***статического полиморфизма***, как и рассмотренная в предыдущем параграфе перегрузка функций. Мы уже упоминали, что функции, имя которых содержит слово `operator`, стали одной из причин, по которым в Си++ вообще появилась перегрузка имён функций; как мы увидим позже, такие функции не всегда вводятся как методы классов, обычная функция тоже может иметь такое «странное» имя и вводить пользовательский смысл для арифметической операции; различать операции, имеющие одно и то же обозначение (например, `+`), компилятор вынужден по *типам аргументов* — то есть ему приходится на основании информации о типах выбирать одну из многих функций, а это и есть, собственно говоря, перегрузка. Но если про перегрузку функций в общем виде мы говорили, что она не имеет никакого отношения к интересующим нас парадигмам (и вообще к каким бы то ни было парадигмам), то перегрузка символов стандартных операций — это весьма существенный элемент поддержки пользовательских абстрактных типов данных.

Довольно неприятная терминологическая путаница[9] возникает из-за английского слова *operator*, которое так и хочется перевести на русский как «оператор», прочитав его «как написано». Увы, делать этого нельзя, ведь русское слово «оператор» в программировании имеет совершенно иное значение: так мы называем законченные структурные единицы текста программы, такие как оператор ветвления (`if`), операторы циклов (`for`, `while`, `do-while`) и другие управляющие конструкции. Напомним, что в Си (и в Си++, естественно, тоже) можно превратить в *оператор* произвольное арифметическое выражение, добавив к нему точку с запятой; такой оператор называется «оператором вычисления выражения ради побочного эффекта». В частности, когда мы вызываем функцию, игнорируя при этом её возвращаемое значение, мы применяем именно этот оператор.

Английское слово *operator* обозначает сущность совершенно иную — то, что мы по-русски обычно называем «операциями»: всевозможные арифметические действия (сложение, вычитание и т. п.), операции сравнения, побитовые и логические операции, операции с адресами и указателями, а если речь идёт о Си и его потомках — то ещё и операции присваивания. Иначе говоря, по-английски *operator* — это нечто такое, что может встречаться в *выражении* наряду с константами, переменными и скобками.

Ничего удивительного в именно таком использовании слова *operator* нет. Читатель, знакомый с высшей математикой, несомненно, встречал такие термины, как «линейный оператор», «оператор дифференцирования» и т. п., которыми обычно обозначаются отображения из некоторого пространства в него само, т. е. попросту *функции*, имеющие некое рассматриваемое пространство одновременно в качестве области определения и области значений. Когда речь идёт о таких вот — математических — «операторах», соответствующим английским термином окажется как раз слово *operator*. Ну а арифметические операции, понятное дело, с этими *operator*'ами состоят в самом близком родстве, так что для обозначения (в математике) символов вроде плюса, минуса или деления англоговорящие математики используют тот же самый термин; естественно, переняли этот термин и программисты.

Что же касается структурных единиц программного текста, называемых по-русски «операторами», то этой сущности соответствует совершенно иной английский термин: *statement* (буквально переводится как «утверждение»). Как так получилось, что английское *statement* переводят на русский словом «оператор», кто это первым придумал, создав для грядущих поколений русскоязычных программистов неприятную терминологическую проблему — вопрос, конечно, интересный; к сожалению, сделать уже ничего нельзя, термин «оператор» намертво врос в русскую программистскую лексику именно в этом совершенно нелогичном для него значении.

10.4.3. Конструктор умолчания. Массивы объектов

Возвращаясь к нашему классу `Complex`, напомним (см. стр. 37), что описать переменную типа `Complex` привычным нам образом, без всяких параметров, нельзя, т. к. для единственного конструктора класса `Complex` требуются два параметра. Так, следующий код будет некорректен:

```
Complex sum; // ОШИБКА - нет параметров для конструктора
sum = c1+c2;
```

Кроме того, у нас возникнут сложности при создании массивов комплексных чисел; обычный массив заранее заданного размера мы создать сможем, но только при указании инициализатора, имеющего весьма нетривиальный синтаксис; забегая вперёд, отметим, что *динамический* массив комплексных чисел мы создать не сможем вообще никак, поскольку синтаксис языка не позволяет задать параметры для конструкторов каждого элемента массива.

Снять возникшие проблемы позволяет введение ещё одного конструктора. Напомним, что компилятор считает конструктором функцию-член класса (или структуры), имя которой совпадает с именем класса (или структуры). Благодаря свойству перегрузки функций в Си++ мы можем ввести в одной области видимости несколько функций с одним и тем же именем; это относится, разумеется, и к конструкторам. Нужно только, чтобы функции, имеющие одинаковые имена (в данном случае — конструкторы), различались количеством и/или типом параметров. Итак, добавим в класс `Complex` ещё один конструктор:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    Complex() { re = 0; im = 0; }
// ...
```

Теперь мы можем описывать переменные типа `Complex`, не задавая никаких параметров. Поэтому конструктор, имеющий пустой список параметров, называют ***конструктором по умолчанию***. Его наличие делает возможными, в частности, следующие описания:

```
Complex z3;      // используем конструктор по умолчанию
Complex v[50];   // конструктор будет вызван 50 раз,
                 // т.е. для каждого элемента
```

10.4.4. Конструкторы преобразования

При создании программ на языках, обладающих типизацией, нередко возникает потребность использовать значение одного типа там, где по смыслу предполагается значение другого типа. Один из простейших примеров такой ситуации — сложение (или другая арифметическая операция) числа с плавающей точкой и целого числа. В языке Си в подобных случаях производится так называемое *неявное преобразование типов*. Так, если мы опишем переменную `a` как целочисленную, а переменную `b` как имеющую тип `float`, и после этого используем в

[8] Некоторые авторы включают в этот список ещё и «операцию» раскрытия области видимости; это не вполне корректно, поскольку раскрытие области видимости *не является операцией*. К этому вопросу мы вернёмся в §10.4.13.

[9] Об этом мы предупреждали ещё в первом томе, когда впервые столкнулись со словом «оператор» при изучении Паскаля (см. т. 1, стр. 79); вот мы и добрались до ситуации, когда слово «оператор» в качестве перевода термина «statement» начинает откровенно мешать.

программе выражение `a + b`, компилятор *неявно* преобразует значение переменной `a` к типу `float`, и только после этого выполнит сложение. Точно так же, если в программе описана функция `void f(float)`, мы можем записать вызов `f(25)`, и компилятор сочтёт такой код корректным, т. к. «знает», каким образом из целого числа сделать число с плавающей точкой; иначе говоря, в языке Си присутствует правило преобразования значений типа `int` в значения типа `float`. Естественно, язык Си++ также позволяет производить подобные неявные преобразования, но, помимо этого, в нём есть средства указания правил преобразования для *типов данных, введённых пользователем*. Одним из таких средств являются *конструкторы преобразования*.

Очевидно, что задать правило преобразования значений типа `A` в значения типа `B` — это то же самое, что задать инструкцию по созданию объекта типа `B` по имеющемуся значению типа `A`. В самом деле, в разобранном выше примере компилятор сначала строил значение типа `float`, используя в качестве отправной точки имеющееся значение типа `int`; разумеется, компилятору для этого необходимо «знать», как это делать. Как отмечалось в §10.3.3, инструкции компилятору Си++ по созданию объектов класса на основе заданных параметров даются путем описания конструкторов с соответствующими параметрами. Таким образом, если ввести в классе `B` конструктор, получающий параметр типа `A`, это как раз и будет инструкция по созданию объекта типа `B` по имеющемуся значению типа `A`. Представляется поэтому вполне логичным использовать такие инструкции для *выполнения неявного преобразования типов*.

Итак, конструктор, который получает на вход ровно один параметр, имеющий тип, отличный от описываемого, называется ***конструктором преобразования***[10] и используется компилятором не только в случае явного создания объекта, но и для проведения неявного преобразования типов. Проиллюстрируем сказанное на примере введённого ранее класса `Complex`. Ясно, что по смыслу комплексных чисел любое действительное число может быть преобразовано к комплексному добавлением нулевой мнимой части точно так же, как целое число преобразуется к числу с плавающей точкой добавлением нулевой дробной части. Чтобы выразить это соотношение между действительными и комплексными числами, дополним класс следующим конструктором:

```
Complex(double a) { re = a; im = 0; }
```

С одной стороны, этот конструктор позволяет нам описывать комплексные числа с указанием одного действительного параметра, например:

```
Complex c(9.7);
```

С другой стороны, если в программе имеется функция

```
void f(Complex a);
```

— мы можем вызвать её для действительного параметра:

```
f(2.7);
```

Благодаря наличию в классе `Complex` конструктора преобразования компилятор успешно обработает такой вызов; для этого с помощью конструктора преобразования будет создан ***временный объект*** типа `Complex`, который и будет подан на вход функции `f`.

10.4.5. Ссылки

Понятие ***ссылки***, которому посвящён этот параграф, оказывается ключевым для понимания дальнейшего материала. Между тем, ничего похожего нет ни в чистом Си, ни в других языках программирования, так что ссылки часто вызывают у начинающих определённые трудности. Постарайтесь поэтому подойти к изучению этого параграфа особенно внимательно, а при возникновении вопросов обязательно задайте их вашему преподавателю или кому-то ещё, кто сможет вам объяснить происходящее.

Ссылка в Си++ — это особый вид объектов данных, реализуемый путём хранения адреса какой-то переменной, но, в отличие от указателя, ссылка семантически эквивалентна той переменной, на которую она ссылается. Иначе говоря, любые операции над ссылкой будут на самом деле производиться над той переменной, адрес которой содержится в ссылке. Надо отметить, что это относится и к присваиваниям, и к взятию адреса — вообще ко всем операциям, какие могут быть; саму ссылку, таким образом, вообще невозможно изменить, её значение (переменная, на которую она ссылается) задаётся в момент создания ссылки и остаётся неизменным в течение всего срока её существования.

Синтаксически тип данных «ссылка» описывается аналогично указателю, только вместо символа «`*`» используется символ «`&`»[11]. Проиллюстрируем сказанное простым примером:

```
int i;       // целочисленная переменная
int *p = &i; // указатель на переменную i
int &r = i;  // ссылка на переменную i

i++;         // увеличить i на 1
(*p)++;      // то же самое через указатель
r++;         // то же самое по ссылке
```

Обычно говорят о «переменных ссылочного типа» наравне с переменными других типов, хотя ссылки не вполне корректно называть «переменными», поскольку изменить значение *самой ссылки* нельзя: всё, что мы с ней попытаемся сделать, на самом деле будет сделано с той переменной, на которую ссылка ссылается.

Из-за неизменности ссылок **переменную ссылочного типа нельзя описать без инициализации, то есть без задания начального значения**. Такое описание было бы заведомо бессмысленным, ведь если не задать значение ссылки с самого начала, то потом мы никаким способом осмысленное значение в ссылку уже не занесём. Пользуясь случаем, напомним ещё раз, что инициализация и присваивание — это совершенно разные сущности.

Описав в нашем примере ссылку на переменную `i`, мы фактически ввели *синоним* имени `i` — имя `r`, обозначающее тот же самый объект данных. Такое использование ссылок может показаться бессмысленным и действительно встречается очень редко. По-настоящему возможности ссылочного типа в Си++ раскрываются при передаче ссылок в функции и возврате их из функций в качестве значения. Так, благодаря ссылочному типу в нашем распоряжении оказывается отсутствовавший в языке Си механизм передачи параметров по ссылке[12]. Допустим, нам нужно написать функцию, находящую максимальный и минимальный элементы заданного массива чисел типа `float`. На чистом Си эта функция выглядела бы так:

```
void max_min(float *arr, int len, float *min, float *max)
{
    int i;
    *min = arr[0];
    *max = arr[0];
    for(i=1; i<len; i++) {
        if(*min>arr[i])
            *min = arr[i];
        if(*max<arr[i])
            *max = arr[i];
    }
}
```

а её вызов — например, так:

```
    float a[500];
    float min, max;
    // ...
    max_min(a, 500, &min, &max);
```

[10] Если только специальное значение такого конструктора не отменить директивой `explicit`. Подробно мы эту директиву рассматривать не будем, но помнить о её существовании полезно.

[11] Важно понимать, что в данном случае символ «`&`» не имеет ничего общего с операцией взятия адреса! Почему автор языка Си++ Б. Страуструп выбрал именно такое обозначение — вопрос открытый.

[12] Передача параметров по ссылке читателю уже знакома: в языке Паскаль такие параметры называются параметрами-переменными (var-parameters), см. т. 1, §2.3.4.

Поскольку из функции нужно вернуть больше одного значения, приходится использовать возврат через параметры. Между тем в языке Си все параметры передаются *по значению*, так что мы вынуждены вручную *имитировать* выходные параметры, передавая значение адреса переменной, подлежащей модификации; при вызове функции мы вынуждены применять операцию взятия адреса («`&`»). В самой функции вместо имени переменной приходится использовать леводопустимое выражение разыменования, т. е. ставить перед идентификаторами `min` и `max` символ операции разыменования «`*`», чтобы преобразовать адрес переменной в саму переменную.

С применением ссылок и код функции, и её вызов можно написать яснее:

```
void max_min(float *arr, int len, float &min, float &max)
{
    int i;
    min = arr[0];
    max = arr[0];
    for(i=1; i<len; i++) {
        if(min>arr[i])
            min = arr[i];
        if(max<arr[i])
            max = arr[i];
    }
}

    // ...
    max_min(a, 500, min, max);
```

Параметры `min` и `max`, объявленные на сей раз как ссылки, семантически представляют собой синонимы неких переменных типа `float`, так что все операции, которые производятся над `min` и `max`, на самом деле происходят над теми переменными, на которые эти параметры ссылаются. При вызове функции мы не применяем никаких операций взятия адреса, поскольку (с семантической точки зрения) нам для построения синонима нужна сама переменная, а не её адрес.

Ещё более интересные возможности открывает использование ссылочного типа как типа возвращаемого значения функции. Допустим, нам нужно произвести поиск целочисленной переменной в составе сложной структуры данных (например, искомая переменная является полем структуры, которая, в свою очередь, является элементом массива и т. п.), а затем либо использовать значение найденной переменной, либо выполнить над ней то или иное присваивание. В такой ситуации поиск переменной можно выделить в отдельную функцию, возвращающую ссылку на найденную переменную. Если заголовок такой функции имеет вид

```
int &find_var(/*params*/);
```

— то допустимы будут, например, такие действия:

```
int x = find_var(/*...*/) + 5;
find_var(/*...*/) = 3;
find_var(/*...*/) *= 10;
find_var(/*...*/)++;
int y = ++find_var(/*...*/);
```

10.4.6. Ссылки на константы

Модификатор `const` уже знаком нам по языку Си, хотя изначально он появился именно в Си++, а в чистый Си проник позднее, причём так и не был признан авторами языка. Напомним, что при описании адресных типов модификатор `const` позволяет указать, что область памяти, на которую указывает описываемый указатель, не подлежит изменению. Например:

```
const char *p;
    // p указывает на неизменяемую область памяти
p = "A string";
    // всё в порядке, переменная p может изменяться
*p = 'a';
    // ОШИБКА! Нельзя менять область памяти,
    // на которую указывает p
p[5] = 'b';     // Это также ОШИБКА
```

Подчеркнём ещё раз, что `const char *p` — это именно *указатель на константу*, а не *константный указатель*. Чтобы описать константу адресного типа, необходимо расположить ключевые слова в другом порядке:

```
char buf[20];
char * const p = buf+5;
     // p - константа-указатель, указывающая на
     // шестой элемент массива buf (buf[5])

p++;        // ОШИБКА, значение p не может быть изменено
*p = 'a';   // всё в порядке, изменяем значение buf[5]
p[5] = 'b'; // всё в порядке, изменяем значение buf[5+5]
```

Аналогично со ссылочными типами:

```
int i;
const int &r = i; // ссылка на константу
int x = r+5; // всё в порядке
i = 7;       // тоже всё в порядке
r = 12;  // ОШИБКА, значение по ссылке r нельзя менять

const int j = 5;
int &jr = j;  //  ОШИБКА, нельзя ссылаться
              //  на константу обычной ссылкой
const int &jcr = j; // всё в порядке
```

Ссылки на константы часто называют просто *константными ссылками*, хотя это не вполне правильно; но поскольку сама ссылка как таковая в принципе не может меняться в силу своей природы, ни к какой путанице это обычно не приводит.

Ссылки, снабжённые модификатором `const`, позволяют передавать в функции в качестве параметра адрес переменной вместо копирования значения, при этом не позволяя эту переменную менять, как и при обычной передаче по значению. Это может быть полезно, когда передаваемое значение (обычно это объект класса или структуры) имеет размер, существенно превышающий размер адреса. Так, если в ранее описанном классе `Complex` вместо

```
Complex operator+(Complex op2)
    { return Complex(re+op2.re, im+op2.im); }
```

написать

```
Complex operator+(const Complex &op2)
    { return Complex(re+op2.re, im+op2.im); }
```

— семантика кода не изменится, но физически (на уровне машинного кода) вместо копирования двух полей типа `double` будет происходить передача адреса существующей переменной и обращение по этому адресу.

10.4.7. Ссылки как семантический феномен

Не следует недооценивать важность ссылок как семантического изобретения. В определённом смысле именно ссылки можно считать самым важным новшеством языка Си++; всё остальное, что содержит этот язык, ранее встречалось в других языках. Введение ссылок открывает очень интересные перспективы, и сейчас мы попытаемся эти перспективы обозначить.

При описании языка Си обычно вводится понятие ***леводопустимого выражения*** (англ. *lvalue*; см. т. 2, §4.3.16), изначально обозначавшее всевозможные выражения, допустимые *слева* от знака присваивания (отсюда название). Результат вычисления такого выражения не просто представляет собой значение, а идентифицирует некую область памяти, в которой располагается значение. Простейшим примером леводопустимого выражения служит *имя переменной*, но этим, конечно

же, дело не ограничивается. Так, если `p` — некий указатель типа, отличного от `void`, то `*p` — леводопустимое выражение; если `v` — некий массив (или, опять же, адрес), то `v[i]` будет леводопустимым; если `q` — адрес структуры, содержащей поле `f`, то `q->f` — леводопустимое выражение, и так далее. Общий принцип тут такой: результатом вычисления леводопустимого выражения в действительности становится некая ***область памяти***, и если от выражения требуется значение, то оно берётся из этой области памяти; когда леводопустимое выражение стоит слева от присваивания, значение от него не требуется, а требуется сама область памяти как таковая, чтобы занести в неё новое значение. Когда леводопустимое выражение оказывается операндом таких операций, как `+=` или `++`, соответствующая область памяти используется и как источник значения, и место для размещения результата.

В чистом Си леводопустимость — свойство конкретного выражения, не типа, не переменной, не функции, а именно выражения. Ссылки, которые Страуструп придумал для Си++, замечательны тем, что позволяют оперировать понятием леводопустимости на уровне системы типов. Для этого достаточно считать, что идентификатор переменной, имеющей, например, тип `int`, сам по себе представляет выражение не типа `int`, а типа *ссылка на* `int`, и то же самое сказать про остальные случаи леводопустимых выражений: если `p` имеет тип `int*`, то `*p` — это не `int`, а `int&` (ссылка на `int`), если `f` — поле структуры, имеющее тип `double`, то `s.f` или `q->f` — выражения типа `double&`, если `str` — массив элементов типа `char`, то `str[i]` имеет тип `char&` и т. д.

Если принять такую терминологию, потребность в отдельном понятии «леводопустимого выражения» отпадает: вместо этого можно говорить, что слева от присваивания, а равно и в других «модифицирующих» случаях могут применяться ссылки и только они, если же ссылка применяется в выражении, в котором требуется *значение*, то ссылка автоматически преобразуется к своему значению (т. е. выполняется извлечение значения из памяти, известное также как ***обращение к переменной***).

Нетрудно видеть, что наличие константных ссылок позволяет отразить разницу между пресловутыми *modifiable lvalues* и *non-modifiable lvalues*: первым соответствуют обычные ссылки, вторым — константные. Остаётся лишь сожалеть, что столь простая и стройная семантическая концепция не была замечена создателями «стандартов» Си++: вместо того, чтобы избавиться от странных терминов *lvalue* и *rvalue*, заменив их ссылками, стандартизаторы, наоборот, ввели ещё *prvalues*, *xvalues* и вообще напрочь запутали описание языка. Интересно, что, например, авторы компилятора gcc, судя по выдаваемой этим компилятором диагностике (именно — по сообщениям об ошибках при поиске подходящей функции), потенциал ссылок в роли замены понятия *lvalue* заметили и воспользовались им.

На случай, если всё изложенное выше про «семантический феномен» показалось читателю чем-то вроде отвлечённой философии, приведём один небезынтересный пример. Пусть у нас есть две целочисленные переменные `x` и `y` и нам понадобилось присвоить какое-то значение (скажем, находящееся в переменной `z`) той из них, значение которой в настоящий момент меньше. Конечно, это можно сделать с помощью обычного `if`:

```
if(x < y)
    x = z;
else
    y = z;
```

В Си++ благодаря ссылкам можно сделать так:

```
(x < y ? x : y) = z;
```

В чистом Си такое не проходит, поскольку имена `x` и `y` в условном выражении соответствуют *значениям* этих переменных, а не им самим. Аналогичную конструкцию сделать всё же можно, но труднее:

```
*(x < y ? &x : &y) = z;
```

10.4.8. Константные методы

Представим себе, что мы описали некий класс (пусть он называется `A`), после чего создали функцию, получающую параметром адрес объекта такого класса, снабжённый модификатором `const`:

```
void f(const A *ptr)
{
    // ...
}
```

Внутри функции объект, на который указывает `ptr`, будет *константным*, то есть его будет нельзя изменять. Если класс `A` описывает полноценный объект, то все его поля скрыты, то есть взаимодействие с объектом возможно только через методы. В то же время метод может, вообще говоря, изменить внутреннее состояние (то есть значения скрытых полей) объекта, нарушив требование о его неизменности. Поэтому для обеспечения неизменности объекта компилятор вынужден запрещать вызовы методов.

Такая же ситуация возникнет, естественно, при передаче параметром *константной ссылки*, при возврате адресов и ссылок на константы из функций, а равно и в случае, если переменная-объект сама по себе изначально описана с модификатором `const` (хотя последнее используется сравнительно редко). Итак, объект вроде бы есть, но всё, что мы можем с ним делать — это вызывать его методы, а поскольку метод может изменить объект, компилятор нам вызывать методы не позволит. Получается, что с таким объектом мы вообще никак не сможем работать, если не предпримем специальных мер.

Для работы с константными объектами в языке Си++ предусмотрены ***константные методы***. Константным считается метод, после заголовка которого (но *перед* телом, если таковое присутствует) поставлен модификатор `const`:

```
class C1 {
    // ...
    void Method(int a, int b) const
        { /* ... */ }
    // ...
 };
```

В теле такого метода поля объекта доступны, но запрещены действия, изменяющие их или способные привести к их изменению, в том числе присваивание полям новых значений, присваивание адресов этих полей неконстантным указателям, передача их в функции по неконстантным ссылкам и т. п. Иначе говоря, относительно константного метода известно, что он заведомо не может изменить состояние объекта (значения его полей). Поэтому константные методы можно без опаски вызывать для объектов, которые нельзя изменять. Например, такой код:

```
void f(const MyClass *p)
{
    p->MyMethod();
}
```

будет корректным только в случае, если в классе `MyClass` метод `MyMethod` объявлен как константный.

При написании программ **рекомендуется все методы, которые по своему смыслу не должны изменять состояние объекта, обязательно помечать как константные, тем самым разрешая вызывать их для константных объектов**. В частности, в описанном ранее классе `Complex` все методы, кроме конструкторов, никаких изменений в поля класса не вносят. Чтобы с объектами типа `Complex` было удобнее работать, следует пометить все методы класса модификатором `const`:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    Complex(double a_re)
        { re = a_re; im = 0; }
    Complex() { re = 0; im = 0; }
    double Magnitude() const
```

```
        { return sqrt(re*re + im*im); }
    double Argument() const
        { return atan2(im, re); }
    double GetRe() const { return re; }
    double GetIm() const { return im; }
    Complex operator+(const Complex &op2) const
        { return Complex(re+op2.re, im+op2.im); }
    // ...
};
```

В теле константного метода нельзя для того же объекта вызывать методы, не являющиеся константными. Причина такого ограничения очевидна: неконстантный метод может изменить объект, а это внутри константного метода делать запрещено. Это можно объяснить и иначе. **Внутри константного метода произвольного класса или структуры `C` указатель `this` имеет тип `const C *`, а не просто `C *`**; вызов неконстантного метода означал бы, что мы передаём как неконстантный параметр значение, которое у нас самих константное, то есть мы снимаем модификатор `const`, а это делать компилятор запрещает.

10.4.9. Операции работы с динамической памятью

Как мы помним, язык Си сам по себе не включает средств работы с динамической памятью; создание и уничтожение динамических структур данных вынесено в библиотеку и производится обычно с помощью функций `malloc`, `calloc`, `realloc` и `free`. В языке Си++ структурные объекты могут обладать конструкторами и деструкторами; при создании и уничтожении таких объектов нужно не просто выделить или освободить память, но и вызвать соответственно конструктор и деструктор. Кроме того, при создании динамических объектов с помощью конструктора иного, нежели конструктор по умолчанию, необходима возможность указания параметров конструктора.

Функции `malloc` и `free` ничего не знают о конструкторах, деструкторах и параметрах, поэтому для создания и удаления объектов они непригодны. Более того, информацией о конструкторах и деструкторах обладает только компилятор, поэтому в языке Си++ вообще невозможно вынести работу с динамической памятью из языка в библиотеку без введения дополнительных средств.

Автор языка Си++ Бьёрн Страуструп решил пойти более простым путем и внёс в язык соответствующие синтаксические конструкции для создания и удаления объектов. Для создания в динамической памяти одиночного объекта (переменной произвольного типа) в языке Си++ используется операция `new`, в которой нужно указать имя типа создаваемого объекта. Например:

```
int *p;
p = new int;
```

Если создаваемый объект принадлежит к типу, имеющему конструктор, и возникает необходимость передать конструктору параметры, то эти параметры указываются в скобках после имени типа. Так, создание объекта описанного ранее типа `Complex` (см. §10.3.5) с указанием действительной и мнимой частей может выглядеть так:

```
Complex *p;
p = new Complex(2.4, 7.12);
```

Для удаления используется операция `delete`:

```
delete p;
```

Для создания и удаления динамических массивов используются специальные *векторные формы* операций `new` и `delete`, синтаксически отличающиеся наличием квадратных скобок:

```
int *p = new int[200]; // массив из 200 целых чисел
delete[] p;            // удаление массива
```

Эти формы отличаются тем, что соответствующие конструкторы и деструкторы вызываются для *каждого элемента массива*. Стоит заметить, что векторная форма операции `new` не имеет синтаксических средств для передачи параметров конструкторам, поэтому для создания массива элементов типа класс или структура *необходимо наличие у этого типа конструктора по умолчанию* (см. §10.4.3).

Важно помнить, что **объекты в динамической памяти, созданные с помощью векторной формы операции `new`, нельзя удалять с помощью обычной формы операции `delete` и наоборот**. Дело в том, что реализация менеджера динамической памяти вправе выделять память под обычные переменные и под массивы из разных областей динамической памяти, имеющих, возможно, различную организацию служебных структур данных. Также не следует удалять с помощью `delete` объекты, созданные функцией `malloc`, а объекты, созданные операциями `new` — с помощью `free`.

Скорее всего, ваш компилятор и прилагающаяся к нему библиотека построены так, что смешение разных способов выделения и освобождения памяти никаких негативных последствий не повлечёт. Это, однако, не повод так делать: при переходе на другой компилятор или даже на другую версию того же самого компилятора всё может внезапно сломаться.

10.4.10. Конструктор копирования

Рассмотрим следующую ситуацию. В реализации некоторого класса (назовём его `Cls1`) нам потребовался динамический массив, который мы создаём в теле конструктора класса; естественно, в деструктор следует поместить оператор для уничтожения этого массива:

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    ~Cls1() { delete[] p; }
    // ...
};
```

Теперь предположим, что кто-то создаёт в программе копию объекта класса `Cls1`. Такое может произойти, если объект окажется передан *по значению* в качестве параметра функции, например:

```
void f(Cls1 x)
{
    // ...
}
int main()
{
    // ...
    Cls1 c;
    f(c);
    // ...
}
```

Проанализируем происходящее со структурами данных при вызове функции `f`. Локальная переменная `x` является копией объекта `c`. Копия любого объекта данных создаётся путем обычного побитового копирования, если не указать иного. Следовательно, при копировании объекта класса `Cls1` скопирован окажется *указатель* на динамический массив; иначе говоря, у нас появятся два объекта, использующие один и тот же экземпляр динамического массива: оригинал объекта `c` и его локальная копия `x`. Возникшая ситуация показана на рис. 10.1.

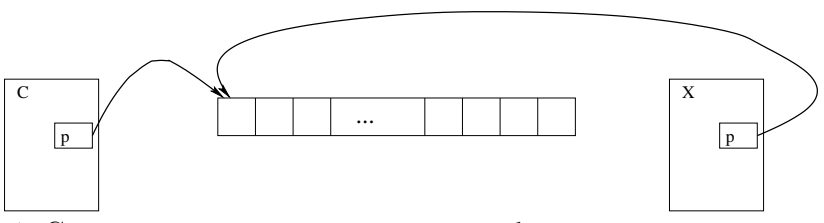

Рис. 10.1. Схема структуры данных после побитового копирования объекта

Даже если такое разделение не приведёт к немедленным ошибкам (например, функция `f` может изменить объект `x`, что отразится на внутреннем состоянии объекта `c`, чего мы могли и не ожидать), в любом

Рис. 10.2. Схема структуры данных после применения конструктора копирования

случае при выходе из функции `f` отработает деструктор для объекта `x`, который уничтожит динамический массив, в результате чего объект `c` окажется в заведомо ошибочном состоянии, поскольку будет содержать указатель на уничтоженный массив. К возникновению ошибки теперь приведёт любое действие с объектом `c`, если же никаких действий не предпринимать, то ошибка возникнет при уничтожении объекта `c`, когда деструктор попытается вновь освободить уже освобождённую память, которую ранее занимал массив.

Очевидно, что для объектов класса `Cls1` побитовое копирование не годится; нужно объяснить компилятору, как создавать копии этих объектов, чтобы всё оставалось корректно. В языке Си++ для этого предусмотрен специальный вариант конструктора, называемый ***конструктором копирования***. Конструктор копирования имеет ровно один параметр, причём тип этого параметра представляет собой *ссылку на объект данного (описываемого) типа — класса или структуры*; в большинстве случаев эту ссылку снабжают модификатором `const`, чтобы показать, что при создании копии исходный объект не изменится. Поскольку всякий конструктор есть, как мы знаем, инструкция компилятору, как создать объект данного типа, имея значения параметров, постольку конструктор копирования со своей ссылкой на объект того же типа оказывается инструкцией, как создать объект данного типа, уже имея один такой объект — то есть, попросту говоря, *как создать копию имеющегося объекта*. Снабдим конструктором копирования наш класс `Cls1`:

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    Cls1(const Cls1& a) {
        p = new int[20];
        for(int i=0; i<20; i++)
            p[i] = a.p[i];
    }
    ~Cls1() { delete[] p; }
    // ...
};
```

Теперь ситуация при вызове функции `f` будет выглядеть так, как показано на рис. 10.2 — у каждого объекта будет свой экземпляр массива `p`.

Отметим, что в Си++ запрещено описывать конструктор, принимающий единственным параметром объект того же класса по значению, а не по ссылке, и тому есть очевидная причина. По смыслу такой конструктор тоже был бы конструктором копирования, ведь он показывает, как надо создавать объект, имея объект того же типа; но поскольку объект он принимает по значению, этот объект при вызове такого конструктора должен быть *скопирован*, так что получается, что, чтобы вызвать такой конструктор, надо сначала применить *его же* для передачи (ему самому) параметра. Поэтому если бы конструктор такой формы не был запрещён, вызвать его всё равно было бы невозможно.

10.4.11. Временные и анонимные объекты

С анонимными и временными объектами мы уже встречались ранее (см. §10.3.3, стр. 37; §10.4.4, стр. 43). ***Анонимные объекты*** обычно применяются, когда объект создаётся, чтобы быть использованным лишь один раз; как правило, в такой ситуации давать объекту имя не хочется, к тому же код с использованием анонимного объекта оказывается более лаконичным и наглядным. Для введения в выражение анонимного объекта используют имя его конструктора, снабжённое списком фактических параметров (возможно, пустым). Пусть, например, у нас есть объекты `t` и `z` класса `Complex` и нам нужно занести в переменную `t` значение `z`, помноженное на мнимую единицу. Можно сделать это так:

```
Complex imag_1(0, 1);
t = z * imag_1;
```

а можно создать анонимный объект, что значительно удобнее:

```
t = z * Complex(0, 1);
```

С ***временными объектами*** ситуация несколько сложнее. В отличие от анонимных объектов, которые программист вводит в выражение в явном виде, объекты временные компилятор порождает самостоятельно, без их явного упоминания, хотя мы и можем в большинстве случаев однозначно предсказать появление временного объекта. Самый простой случай такого появления — передача параметра в функцию с использованием конструктора преобразования. Так, в примере на стр. 43 мы вызывали функцию `f`, предполагающую параметр типа `Complex`, а фактический аргумент представлял собой обычное число с плавающей точкой; используя введённый нами конструктор преобразования, компилятор создал *временный объект* типа `Complex`, который и был в итоге передан в функцию `f`.

Рассмотрим не столь очевидный случай появления временного объекта. Пусть у нас имеются переменные `a`, `b` и `c`, имеющие тип `Complex`, и мы пытаемся вычислить их сумму, например, так:

```
t = a + b + c;
```

Операции сложения в этом операторе будут выполняться, как обычно, слева направо, причём второе сложение в качестве своего «левого» аргумента получит результат первого сложения, который, очевидно, имеет тип `Complex`, то есть *является объектом типа* `Complex`. Ясно, что эту сумму `a` и `b` компилятор должен представить в виде объекта, чтобы *для него* вызвать операцию сложения, передав объект `c` в качестве параметра. Как вы уже догадались, компилятор создаёт для этого временный объект. Этот пример демонстрирует частный случай более общей ситуации, когда компилятор вынужден порождать временные объекты, а именно — при использовании в выражении вызова некоторой функции, которая возвращает значение типа объект, причём возврат производится именно по значению (а не, например, по ссылке). В данном случае в роли такой функции выступает `operator+`.

Несмотря на очевидное различие, анонимные и временные объекты имеют между собой много общего; анонимные объекты можно считать частным случаем временных[13]. В первую очередь отметим *время жизни* временных и анонимных объектов: они существуют до момента окончания вычисления выражения, содержащего их, после чего уничтожаются; например, в операторе вычисления выражения ради побочного эффекта временные объекты просуществуют до окончания выполнения оператора — «до точки с запятой». Если соответствующий класс имеет деструктор, этот деструктор будет вызван. Из этого правила есть одно исключение: если временный или анонимный объект был использован в качестве инициализатора ссылки, то такой объект будет существовать до тех пор, пока существует ссылка.

Второе правило, которое необходимо знать и учитывать, состоит в том, что **на временный или анонимный объект нельзя ссылаться неконстантной ссылкой**. Это ограничение особенно заметно при передаче параметров в функции: если функция принимает параметр типа «объект» по значению или по ссылке на константу, то в качестве такого параметра может быть использован временный или анонимный объект, тогда как если параметр передаётся по неконстантной ссылке, то временные и анонимные объекты для такого параметра не подходят. Это правило при ближайшем рассмотрении оказывается вполне

[13] Более того, Страуструп в своих книгах вообще не делает между ними явного различия, называя «временными объектами» обе рассматриваемые сущности.

логичным. В самом деле, наличие параметра-ссылки, если эта ссылка не обозначена как константная, обычно подразумевает, что соответствующий параметр используется как *выходной*, то есть предполагается, что через этот параметр функция *возвращает* некоторую информацию. Помещать такую информацию во временный объект крайне странно, ведь он исчезнет раньше, чем информация сможет быть использована.

Напрашивается простой методологический вывод: **используйте модификатор `const` для всех ссылок, для которых это возможно**. Неконстантные ссылки в качестве параметров функций следует использовать тогда и только тогда, когда функция заведомо должна модифицировать переменную, переданную через такой параметр, то есть параметр используется для передачи информации из функции к вызывающему — например, когда функция должна возвратить больше одного значения.

10.4.12. Значения параметров по умолчанию

Язык Си++ позволяет при объявлении функции (т. е. в том месте единицы трансляции, где впервые фигурирует прототип функции) задать для некоторых или всех её параметров ***значения по умолчанию***, в результате чего при вызове функции можно будет указать меньшее количество параметров. Недостающие параметры компилятор подставит сам. Например, опишем следующую функцию:

```
void f(int a = 3, const char *b = "string", int c = 5);
```

Теперь возможны такие вызовы:

```
f(5, "name", 10);
f(5, "name"); // то же, что f(5, "name", 5);
f(5);         // то же, что f(5, "string", 5);
f();          // то же, что f(3, "string", 5);
```

На значения по умолчанию накладываются определённые ограничения. Можно указать значение по умолчанию для любого количества параметров функции, но при этом **все параметры функции, следующие в списке параметров за первым, имеющим значение по умолчанию, должны также иметь значение по умолчанию**. Например, следующие описания корректны:

```
void f(int a = 0, int b = 10, int c = 20);
void f(int a, int b = 10, int c = 20);
void f(int a, int b, int c = 20);
```

а следующие — некорректны:

```
void f(int a = 0, int b, int c = 20); // ОШИБКА
void f(int a = 0, int b = 0, int c);  // ОШИБКА
void f(int a = 0, int b, int c);      // ОШИБКА
     // b и c должны иметь умолчание, т.к. его имеет a

void f(int a, int b = 10, int c);     // ОШИБКА
     // c должен иметь умолчание, т.к. его имеет b
```

Это ограничение введено в язык, чтобы упростить компилятору поиск подходящей функции. Если при вызове указано n фактических параметров, компилятор сопоставляет их с n *первыми* формальными параметрами функции. Так, если задана функция

```
int f(int a, const char *str = "name", int *p = 0);
```

то следующий фрагмент будет ошибочен:

```
int x;
f(5, &x);
```

поскольку фактический параметр `&x` имеет тип `int *`, а сопоставлен он будет строго со вторым параметром функции `f`, который имеет тип `const char *`.

Кроме того, выражение, задающее значение по умолчанию, также может стать причиной ошибки, а правила, связанные с этим, громоздки и неочевидны. Лучше всегда задавать умолчания константами времени компиляции, уж с ними-то точно проблем не будет.

В литературе встречается утверждение, что параметры по умолчанию представляют собой ещё одно проявление статического полиморфизма. На самом деле это утверждение сомнительно, поскольку никакого отношения к *различным типам аргументов* этот механизм не имеет; впрочем, вопрос тут, как водится, терминологический, а термины можно определять по-разному.

Введение параметров по умолчанию приводит к необходимости уточнения формулировок, связанных с конструкторами специальных видов. В §§10.4.3, 10.4.4 и 10.4.10 мы рассматривали специфические конструкторы, говоря, что:

- конструктор без параметров воспринимается компилятором как ***конструктор по умолчанию***;
- конструктор с одним параметром, имеющим тип, отличный от описываемого, воспринимается компилятором как ***конструктор преобразования***;
- конструктор с одним параметром, имеющим тип «ссылка на описываемый класс или структуру», воспринимается компилятором как ***конструктор копирования***.

Учитывая возможность задания значений параметров по умолчанию, следует говорить, что компилятор воспримет как конструктор специального вида такой конструктор, который *допускает его вызов* с соответствующими параметрами. Так, мы могли бы в классе `Complex` описать всего один конструктор, который бы служил и конструктором по умолчанию, и конструктором преобразования, и обычным конструктором от двух аргументов:

```
Complex(double a_re = 0, double a_im = 0)
    { re = a_re; im = a_im; }
```

В самом деле, такой конструктор может быть вызван и без параметров, то есть как конструктор по умолчанию, и с одним параметром типа `double`, то есть как конструктор преобразования из `double` в `Complex`.

10.4.13. Описание метода вне класса. Области видимости

До сих пор все описываемые нами методы состояли из одной-двух строк кода. Конечно, так получается далеко не всегда, а размер тела метода, вообще говоря, не ограничен, как не ограничен и размер тела обычной функции. Это, впрочем, не означает, что написание громоздких функций в чём-то правильно; как мы обсуждали ещё в первом томе (см. §2.4.4), идеальная подпрограмма должна укладываться в 25 строк или в крайнем случае быть чуть-чуть больше.

Для методов, тела которых описаны в заголовке (описании) класса, ограничения, налагаемые здравым смыслом (не компилятором!), оказываются намного жёстче. Описание класса при наличии в нём нескольких методов сколько-нибудь заметного объёма может стать (в силу своих размеров) совершенно недоступным для понимания. Чаще всего программисты читают описания классов, чтобы понять, как работать с объектами этого класса, а для этого нужно знать набор его публичных методов. Пока описание класса целиком умещается на экране, список его методов можно охватить одним взглядом; если же в классе описаны методы с громоздкими телами, для ознакомления с заголовками методов может потребоваться долгое перелистывание кода туда и обратно, причём в процессе поиска заголовка одного метода программист может успеть забыть про другие, так что изучение класса становится занятием долгим и утомительным.

Есть и ещё одна проблема с телами методов, которая возникает при написании многомодульных программ. В языке Си мы помещали описания структур, необходимые более чем в одном модуле, в заголовочные файлы. В Си++ с классами и структурами ситуация совершенно такая же: если мы хотим использовать некоторый класс или структуру в нескольких модулях, следует поместить описание этого класса или структуры в заголовочный файл и включить этот файл директивой

`#include "..."` во все нужные модули. Если при этом класс или структура содержит тела методов, это будет означать, что код, составляющий их тела, окажется откомпилирован в *каждом* из модулей. В итоговом исполняемом файле может оказаться значительное количество дублируемого кода: каждый из модулей, использующих наш класс, принесёт в исполняемый файл свою копию кода его методов.

Обе проблемы — громоздкость описания класса и дублирование объектного кода методов — решаются вынесением тел методов за пределы описания класса (называемого также ***заголовком класса***). При этом в заголовке класса оставляется только прототип (заголовок) функции-метода, то есть тип возвращаемого значения, имя метода, список формальных параметров и (возможно) модификатор `const`, после чего вместо тела метода ставится точка с запятой, как и после обычного прототипа функции. Тело функции-метода при этом описывается в другом месте, за пределами заголовка класса, возможно даже, что в другом файле — обычно так случается, если заголовок класса вынесен в заголовочный файл; тела методов при этом описываются в файле реализации соответствующего модуля, так что при трансляции модулей, *включающих* этот заголовочник, компилятор видит заголовок класса, но не видит реализацию его методов.

При описании функции-метода за пределами заголовка класса нужно указать компилятору, что речь идёт именно о методе определённого класса, а не о простой функции. Это делается с помощью ***символа раскрытия области видимости***, в роли которого в Си++ выступает два двоеточия «`::`» (иногда программисты называют этот символ «четвероточием»). Например, если мы описываем некий класс `C1` и в нём есть конструктор по умолчанию и некоторые методы `f` и `g`, вынесение описания методов за пределы класса может выглядеть так:

```
class C1 {
    // ...
public:
    C1();
    void f(int a, int b);
    int g(const char *str) const;
};

// ... //

C1::C1()
{
    // тело конструктора
}
void C1::f(int a, int b)
{
    // тело метода f
}

int C1::g(const char *str) const
{
    // тело метода g
}
```

Смысл «раскрытия области видимости» можно пояснить следующим образом. Имена полей и методов локализованы в классе: если в классе есть метод с именем `f`, то вне класса мы можем использовать идентификатор `f` для других целей либо вовсе не использовать его; появление идентификатора `f` вне класса не имеет никакого отношения к методу `f`, описанному в классе. Вместе с тем в некоторых случаях имя метода `f` всё же появляется вне класса именно как имя этого метода — например, при вызове его для объекта класса. Таким образом, в отличие от, например, локальных переменных в функциях, имена членов класса *доступны* вне класса (конечно, если они не закрыты механизмом защиты), но только если имеются указания на то, что требуется имя из класса; при вызове метода *для объекта* таким указанием считается тип этого объекта.

Говорят поэтому, что класс (или структура) представляет собой ***область видимости*** (или, если угодно, ***пространство имён*** — эти термины считаются в программировании синонимами), в которой могут существовать свои (принадлежащие ей) имена, не конфликтующие с именами вне её даже при совпадении идентификаторов. Имена, локализованные в области видимости, от этого не становятся, вообще говоря, недоступны; они доступны, но только при наличии указаний на область видимости. Символ раскрытия области видимости как раз и предназначен для таких указаний. Можно сказать, что один и тот же метод имеет внутри класса `C1` (то есть в телах его методов) имя `f`, а вне класса — имя `C1::f`.

Символ раскрытия области видимости применяется не только для описания методов вне заголовка класса. Если опустить имя области видимости слева от символа «`::`», подразумевается *глобальная* область видимости; например, если в вашем классе есть метод `func` и при этом в вашей программе есть обычная функция с таким же именем, вы можете к ней обратиться из методов вашего класса, назвав её `::func`. С другими случаями применения символа «`::`» мы столкнёмся позже при рассмотрении статических полей и методов.

Сделаем ещё одно важное замечание. Часто можно столкнуться с утверждением, что «четвероточие» — это якобы *операция* (и, в частности, её включают в список операций, которые нельзя переопределять). Чтобы опровергнуть это утверждение, достаточно заметить, что ни слева, ни справа от символа «`::`» не может стоять никакое *выражение*: ни имя области видимости, ни имя поля или метода не могут быть *вычислены*, их можно указывать только в явном виде. Если считать «`::`» операцией, она окажется «нуль-арной», то есть операцией без операндов; мало того, во многих случаях она ещё и не имеет *значения*. Но главное даже не это. Конструкции с символом «`::`» часто встречаются в таких контекстах, где просто не может быть ничего похожего на арифметическое выражение — например, в заголовках функций или в описаниях переменных. Логичнее считать, что символ «`::`» никакого отношения к операциям не имеет, это просто символ.

10.4.14. «Подставляемые» функции (inline)

К нынешнему моменту у читателя могло сложиться ощущение, что объектно-ориентированное программирование вместе с абстрактными типами данных — это жутко неэффективно из-за того, что к закрытым полям классов и структур приходится обращаться через функции-методы, а ведь вызов функции предполагает помещение в стек её параметров (как минимум указателя `this`), создание и последующую ликвидацию стекового фрейма и всё такое прочее. Для мало-мальски нетривиальных функций всё это не создаёт проблем, поскольку операции, связанные с техническим исполнением вызова функции и возврата из неё, обычно отнимают ничтожное время в сравнении с выполнением тела функции; но ведь мы неоднократно видели в примерах функции-методы, попросту возвращающие значение поля объекта, а скопировать поле из заданной области памяти можно одной командой `mov`!

Действительно, если бы компилятор оформлял все вызовы методов «по-честному», это могло бы привести к ощутимым потерям, но, к счастью, всё не так плохо. Дело в том, что компилятор Си++ некоторые функции считает «подставляемыми» (англ. *inline*; встречается также русский перевод «встраиваемые»). Когда в тексте программы встречается обращение к такой функции, компилятор вместо обычного вызова *подставляет машинный код тела функции*. После этого оптимизатор, работающий на уровне машинных команд, выкидывает всё лишнее, так что итоговый код становится ничуть не менее эффективным, чем если бы мы обращались к полям объекта без всяких вызовов методов — собственно говоря, на уровне машинного кода никаких вызовов и не происходит.

Конечно, такое имеет смысл лишь для функций с очень коротким телом. Вспомним теперь, что́ мы обсуждали в предыдущем параграфе: только очень короткие тела методов стоит оставлять в заголовке класса, а все прочие выносить за его пределы. Сопоставив одно с другим, мы сможем лучше понять логику, стоящую за следующим соглашением: **компилятор Си++ пытается обрабатывать как «подставляемые» все методы, тела которых описаны непосредственно в заголовке класса**.

Мы можем предложить компилятору обрабатывать в качестве подставляемой любую функцию, для этого достаточно перед её описанием поставить ключевое слово `inline`:

```
inline int f(int x)
{
   // ...
}
```

Естественно, сделать это можно и с методом, тело которого мы решили вынести за пределы заголовка класса, например, чтобы не загромождать этот заголовок.

С inline-функциями связано несколько интересных моментов, которые стоит понимать. Прежде всего отметим, что ни описание тела метода в заголовке класса, ни даже явным образом указанная директива `inline` в действительности ни к чему компилятор не обязывают: если по тем или иным причинам он считает, что функцию с таким телом будет эффективнее обрабатывать как обычную вызываемую функцию, а не как подставляемую, то именно так он с ней и поступит.

Второй момент не столь очевиден. Если в одном из ваших модулей вводится inline-функция, которую вы хотите сделать доступной для других модулей, то её *описание* — полностью, вместе с телом — следует вынести в заголовочный файл. Чтобы понять, почему это так, достаточно вспомнить, что редактор связей при сборке программы из объектных модулей расставляет адреса, по которым модули обращаются к объектам (переменным и функциям, или, говоря шире, областям памяти), введённым в других модулях; больше никаких изменений в код редактор связей внести не может, в том числе, естественно, не может он и подставить вместо вызова функции какой-то там фрагмент кода, будь он хоть трижды телом этой функции. Следовательно, подстановку кода inline-функции должен выполнить компилятор во время компиляции, но ведь при компиляции одного модуля компилятор не видит текста других модулей — собственно говоря, в этом и состоит суть *раздельной трансляции*. Всё, что знает компилятор о других модулях — это то, что написано в подключённых заголовочных файлах, они как раз для этого и создаются; так что, коль скоро компилятору для подстановки inline-функции нужно, очевидно, знать её тело, то это тело следует вынести в заголовочник.

Любопытно, что компилятор может создать как обычную, так и подставляемую версию для одной и той же функции. Например, компилятор попросту вынужден генерировать обычную (вызываемую) подпрограмму в случае, если где-то в вашей программе от этой подпрограммы берут адрес и заносят его в указатель на функцию. Впрочем, ни это, ни возможный отказ компилятора рассматривать функцию как «подставляемую» в любом случае не может повлиять на семантику итоговой программы — только на размер машинного кода и скорость выполнения.

Так или иначе, об inline-функциях стоит помнить хотя бы для того, чтобы не бояться потерять эффективность из-за вызовов коротких методов.

10.4.15. Инициализация членов класса в конструкторе

До сих пор мы задавали значения полей объекта с помощью обыкновенного присваивания в теле конструктора. Такой способ может оказаться неприемлемым; действительно, никто не мешает объявить в классе поле, имеющее, в свою очередь, тип класс, причём без конструктора по умолчанию. Рассмотрим эту ситуацию подробнее. Пусть `A` — класс, единственный конструктор которого требует на вход два параметра типа `int`:

```
class A {
    // ...
public:
    A(int x, int y) { /*...*/ }
    // ...
};
```

Опишем теперь класс `B`, в котором есть поле типа `A`. Для простоты картины будем считать, что в классе `B` нам нужен только конструктор по умолчанию (для других конструкторов ситуация будет полностью аналогична):

```
class B {
    A a;
public:
    B();
    // ...
};
```

Рассмотрим тело конструктора `B`. В нём (то есть во время его работы) *уже* доступно поле `a`, а для этого, как мы знаем, должен был отработать конструктор класса `A`. Но ведь единственный конструктор класса `A` требует параметров! Как же их ему передать?

Чтобы решить возникшую проблему, в Си++ введён специальный синтаксис для ***инициализации полей объекта***. При описании конструктора между его заголовком и началом тела (открывающей фигурной скобкой) можно поставить двоеточие, после которого через запятую перечислить вызовы конструкторов для некоторых или всех полей класса. В рассматриваемом примере это будет выглядеть так:

```
B::B() : a(2, 3) { /*...*/ }
```

Здесь мы указываем компилятору, что поле `a` следует инициализировать («сконструировать») с помощью конструктора от двух параметров (`2` и `3`). Обнаружив такой код, компилятор вставит вызов соответствующего конструктора класса `A` в самое начало тела конструктора `B`. Отметим, что так можно инициализировать любые поля, а не только поля типа класс. В частности, для нашего типа `Complex` конструкторы могли бы выглядеть и так:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im) : re(a_re), im(a_im) {}
    Complex(double a_re) : re(a_re), im(0) {}
    Complex() : re(0), im(0) {}
    // ...
```

Сделаем ещё одно важное замечание. **Инициализаторы полей должны следовать в списке после двоеточия в том же порядке, в котором сами поля описаны в классе**. Иное, формально говоря, не является ошибкой, но хороший компилятор обязательно выдаст предупреждение.

10.4.16. Перегрузка операций простыми функциями

В классе `Complex` мы перегружали символы арифметических операций в виде функций-методов. Благодаря наличию в классе конструктора преобразования возможно, например, прибавить к комплексному числу действительное:

```
Complex z, t;
// ...
z = t + 0.5;
```

(при этом константа `0.5` будет преобразована к объекту `Complex` с помощью конструктора преобразования). В то же время следующая операция окажется ошибочной:

```
z = 0.5 + t; // ошибка!
```

Дело тут в том, что метод `operator+` может быть вызван только для объекта класса `Complex`, а константа `0.5` таковым не является. В отличие от аргументов функций *объекты*, для которых вызываются методы, компилятор не преобразует.

Ситуацию довольно легко исправить. Дело в том, что в Си++ операцию сложения двух объектов можно представить не только как метод самого объекта (с одним параметром, по принципу «прибавь к себе этот параметр и скажи, что получится»), но и как стороннюю функцию, которая по двум заданным объектам строит новый объект. Итак, уберём `operator+` из класса `Complex`, а вне класса напишем следующее:

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.GetRe() + b.GetRe(), a.GetIm() + b.GetIm());
}
```

Теперь мы можем написать:

```
Complex z, t;
// ...
z = t + 0.5;
z = 0.5 + t;
```

и никаких ошибок это не вызовет. Можно считать, что выражение `a + b` компилятор пытается превратить в одно из двух следующих выражений:

```
a.operator+(b)
operator+(a, b)
```

и то же самое справедливо почти для всех бинарных операций; с унарными операциями ситуация аналогична, например, `~a` превращается в одно из

```
a.operator~()
operator~(a)
```

Интересно заметить, что приоритета ни тот, ни другой вариант не имеют; если вы предусмотрите в вашей программе и метод, и внешнюю функцию для одной и той же операции (и для одних и тех же типов операндов), компилятор откажется делать выбор между ними и при попытке использовать операцию в инфиксной форме выдаст ошибку.

Перегрузка символов стандартных операций в виде отдельных функций (а не методов в классах) может оказаться полезной также и в том случае, если нам необходимо ввести операцию для объектов некоторого класса, который мы по тем или иным причинам не можем изменить; например, этот класс может быть частью библиотеки, созданной другим программистским коллективом, и от неё у нас может не быть исходных текстов (к сожалению, несмотря на успехи движения Open Source, такие ситуации пока что не редкость).

10.4.17. Дружественные функции и классы

В некоторых случаях бывает полезно сделать исключения из запретов, налагаемых механизмом защиты. В языке Си++ класс или структура, имеющие защищённую часть, могут объявить ту или иную функцию своим «другом» (*friend*); в этом случае из тела такой ***дружественной функции*** все детали реализации класса или структуры будут доступны. Можно объявить «дружественным» также целый класс или структуру; эффект от этого будет такой же, как если бы мы объявили дружественными все их методы. Иначе говоря, если в классе `A` будет заявлено, что класс `B` является для него дружественным, то во всех методах класса `B` будут доступны все детали реализации класса `A`.

Применять механизм дружественных функций и классов следует с осторожностью. Мы обсуждали в §10.3.4, что защита деталей реализации класса — механизм очень полезный; необдуманные исключения из правил, установленных защитой, могут всё испортить. Стоит помнить, что функциям и классам, которых мы «добавили в друзья», никто не запрещает располагаться где угодно, в том числе в других единицах трансляции; грубо говоря, директивой `friend` мы заявляем транслятору, что нашим другом будет не какая-то конкретная функция или какой-то конкретный класс, а *любой класс, который вот так называется*, и *любая функция, имеющая такой профиль*. В таких условиях легко вообще потерять контроль за происходящим.

Механизм дружественных функций часто применяют при выносе перекрытых символов стандартных операций за пределы класса, как это предлагалось в предыдущем параграфе. В теле функции `operator+` мы были вынуждены пользоваться методами `GetRe` и `GetIm`, поскольку из функции, не являющейся формально методом класса `Complex`, прямого доступа к полям `re` и `im` нет. Однако использование методов доступа (так называемых ***аксессоров***) не всегда удобно, к тому же их может попросту не быть; некоторые детали реализации не следует делать доступными даже через метод. Здесь как раз и оказывается полезен механизм дружественных функций.

Для начала в заголовок класса `Complex` вставим директиву `friend`, объявив функцию `operator+` дружественной. Для этого нужно записать прототип дружественной функции, предварив его директивой `friend`:

```
class Complex {
    friend Complex operator+(const Complex&, const Complex&);
    // ...
```

Теперь мы можем переписать функцию `operator+`, используя напрямую поля складываемых объектов без обращения к методам `GetRe` и `GetIm`:

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.re + b.re, a.im + b.im);
}
```

Сравните это описание с тем, которое мы приводили на стр. 51.

Конечно, дружественной может быть и обычная функция:

```
class Cls1 {
    friend void f(int, const char *);
    // ...
};

void f(int, const char *)
{
    // здесь можно использовать закрытые поля Cls1
}
```

Чтобы сделать дружественным сразу целый класс, следует после слова `friend` написать «`class` *имя*»:

```
class A {
    friend class B;
    // ...
};
```

Использовать такой вариант «дружбы» следует с особенной осторожностью и только когда понятия, описываемые обоими классами, тесно связаны между собой; желательно сначала задать себе вопрос, нельзя ли обойтись без `friend`. Бóльшим злом, чем использование дружественных отношений, можно считать разве что снятие защиты с внутренних полей или введение публичных методов доступа к деталям, которые по смыслу должны быть скрыты: ясно, что дружественность всё-таки ограничивает объём кода, который придётся просмотреть при изменении реализации класса, так что использование `friend` предпочтительнее полного отказа от защиты. Позже мы рассмотрим примеры, в которых применение дружественных классов оказывается оправданным.

В заключение обсуждения директивы `friend` отметим один довольно популярный миф; из текста некоторых распространённых пособий по Си++ их читатели делают вывод, что (якобы) символы операций для класса можно переопределять либо функцией-членом (методом), либо функцией-другом. Не верьте! **Функция, не являющаяся методом класса и переопределяющая для объектов класса символ стандартной операции (то есть поименованная с использованием слова `operator`), *никоим образом не обязана* быть классу другом или кем-то ещё.** Просто такие функции сравнительно часто объявляют дружественными классу из соображений удобства их написания; но это совершенно не обязательно.

10.4.18. Переопределение операций присваивания

Как уже говорилось выше, в Си++ можно переопределить («перегрузить») символ любой операции, участвующей в выражениях, за исключением операции выборки поля (точка) и тернарной условной операции. При этом некоторые операции при их перегрузке имеют определённые особенности, которые необходимо учитывать.

Начнём с переопределения операций присваивания. Напомним, что присваивание считается в языках Си и Си++ *операцией*, а не оператором, как в Паскале и большинстве других языков: запись вида «`a =` *выражение*» сама по себе является выражением, имеющим значение, и может входить в состав других выражений. То же самое можно сказать и про операции присваивания, совмещённые с арифметическими действиями, такие как `+=`, `-=`, `<<=`, `|=` и т. п. Операции присваивания, как и другие операции, могут быть переопределены для классов и структур, введённых пользователем, но на них действует ограничение: **операции присваивания можно переопределять только как методы класса или структуры**, а определять их в виде обычных функций нельзя.

Обычно при переопределении операций присваивания учитывают, что простое присваивание возвращает значение, которое только что было присвоено. Из этих соображений из операции присваивания возвращают либо копию объекта, либо (что предпочтительнее) константную ссылку на объект, для которого операция была вызвана. Это, однако, не обязательно: язык Си++ не требует именно такого оформления операций присваивания. Можно, в частности, сделать операцию присваивания функцией типа `void`, то есть не возвращающей никакого значения. Например, для нашего класса `Complex` мы могли бы ввести такие операции:

```
class Complex {
    // ...
    const Complex& operator=(const Complex& c)
        { re = c.re; im = c.im; return *this; }
    const Complex& operator+=(const Complex& c)
        { re += c.re; im += c.im; return *this; }
    // ...
};
```

или, если нас не слишком волнуют семантические традиции, можно было бы написать и так:

```
class Complex {
    // ...
    void operator=(const Complex& c)
        { re = c.re; im = c.im; }
    void operator+=(const Complex& c)
        { re += c.re; im += c.im; }
    // ...
};
```

Здесь стоит ответить на один вопрос, часто возникающий у студентов. Операции `+=`, `-=` и другие подобные им являются с точки зрения языка Си++ полностью самостоятельными, то есть, например, операция `+=` никак не связана ни с операцией `=`, ни с операцией `+`. Если мы опишем в некотором классе операции `=` и `+`, это само по себе не даст нам возможности применять к объектам этого класса операцию `+=`, она должна быть описана отдельно. В принципе ей никто не мешает делать что-то совершенно иное, нежели последовательно применённые `+` и `=`, ведь компилятор никак не может проверить соответствие одного и другого; но лучше от таких трюков воздержаться, чтобы не создавать в программе лишних ребусов для разгадывания.

Аргумент операции присваивания не обязан иметь тот же тип, что и описываемый класс. Так, для комплексных чисел мы могли бы определить присваивание комплексной переменной действительного числа:

```
class Complex {
    // ...
    void operator=(double x) { re = x; im = 0; }
    // ...
};
```

Операция присваивания, аргумент которой представляет объект того же класса или ссылку на такой объект, как в предыдущем примере, имеет одну важную особенность: такая операция *генерируется неявно*, если её не описать. Неявные методы мы подробно обсудим в следующем параграфе.

10.4.19. Методы, возникающие неявно

Если описать в программе на Си++ структуру или даже класс, не содержащий (по крайней мере на первый взгляд) никаких конструкторов, то переменную такого типа всё же окажется возможным создать. Это вполне понятно с позиций здравого смысла: ведь структуры в Си++ часто используются в их исходной роли, пришедшей из языка Си, то есть в качестве обычной структуры данных, безо всяких методов. Между тем семантика языка Си++ подразумевает, что каждый экземпляр структуры или класса является объектом, а для каждого объекта при его создании вызывается конструктор. Возникающее противоречие решается введением понятия ***неявного конструктора***.

Неявный конструктор — это конструктор, который генерируется компилятором автоматически, несмотря на отсутствие соответствующего конструктора в коде, описывающем класс или структуру. Компилятор Си++ неявно генерирует только два вида конструкторов: конструкторы по умолчанию (то есть конструкторы без параметров) и конструкторы копирования. При этом **конструктор копирования неявно генерируется для любого класса или структуры, в которых программист не описал конструктор копирования явно**, то есть получается, что конструктор копирования на самом деле присутствует (явно или неявно) вообще в любом классе или структуре. Неявный конструктор копирования производит копирование наиболее очевидным способом: поля, которые сами имеют конструкторы копирования, копируются с помощью этих конструкторов, прочие поля — побитовым копированием.

С **конструктором по умолчанию** ситуация чуть сложнее: он **генерируется неявно, если программист не описал в структуре или классе вообще ни одного конструктора**. Если в классе или структуре явно описать хотя бы один конструктор (любой), компилятор не будет генерировать неявный конструктор по умолчанию, поскольку сочтёт, что программист взял заботу о конструировании в свои руки. Именно поэтому в примерах, которые рассматривались в §10.3.3, мы могли описывать переменные типа `str_complex` «по-сишному», пока не ввели первый конструктор, после чего возможность описания переменных этого типа без указания параметров нами была утрачена до тех пор, пока мы (уже сами, явно) не определили конструктор по умолчанию.

Неявная версия конструктора по умолчанию использует для инициализации полей класса, в свою очередь, конструкторы по умолчанию, определённые для соответствующих типов. Можно считать, что для встроенных типов, таких как числа или указатели, конструкторы по умолчанию тоже есть, просто они ничего не делают.

Аналогично обстоят дела и с деструкторами. Считается, что деструктор есть в любом классе и любой структуре, даже если мы его там не описали: в этом случае компилятор создаст его неявно. В простейшем случае такой неявный деструктор не будет делать ничего, но если в нашем классе или структуре есть поля, имеющие нетривиальные деструкторы, то наш неявный деструктор будет содержать их вызовы.

Последний метод, генерируемый неявно — это операция присваивания объекту того же типа. Это вполне понятно, ведь в чистом Си можно было присваивать между собой переменные, имеющие один структурный тип, и эта возможность была унаследована в Си++. Точнее, **если в некотором классе `A` не описать явным образом операцию присваивания с параметром, имеющим тот же тип `A` или ссылку на него, то компилятор неявно создаст операцию со следующим профилем**:

```
class A {
   // ...
   const A& operator=(const A& other);
   // ...
};
```

Такая неявным образом возникшая операция будет использовать для копирования содержимого каждого поля определённую для этого поля его собственную операцию присваивания, если она у него есть. Интересно, что для некоторых полей операции присваивания может не найтись — например, если поле имеет ссылочный тип или объявлено как константное. В этом случае компилятор не сможет сгенерировать неявную операцию присваивания.

Сделаем ещё одно важное замечание. Поскольку конструктор копирования может быть сгенерирован неявно, отсутствие явно описанного конструктора копирования не означает невозможности создания копии объекта. Существует, однако, **способ запретить копирование объектов некоторого класса**: для этого достаточно **описать конструктор копирования** явно, но сделать это **в приватной части класса**. Такой приём часто применяют для классов, объекты которых по смыслу копироваться не должны, чтобы исключить случайные ошибки, связанные, например, с передачей их по значению. Ясно, что объект, для которого копирование запрещено, не может быть ни передан по значению в функцию, ни возвращён из функции; это не исключает, разумеется, передачи по ссылке. Интересно, что применение объекта, копирование которого запрещено, в роли *поля* другого объекта сделает невозможной генерацию неявного конструктора копирования для этого второго объекта, так что его копирование будет запрещено автоматически; впрочем, это не мешает создать конструктор копирования явно. Аналогичный приём можно применить и для запрещения присваивания объектов: достаточно описать операцию присваивания с аргументом «константная ссылка на объект описываемого класса» *в приватной части класса*:

```
class A {
    // ...
private:
    A(const A& ref) {}              // no copying
    void operator=(const A& ref) {} // no assignments
};
```

Иногда в приватную часть класса убирают деструктор; этому приёму мы посвятим §10.6.11.

10.4.20. Переопределение операции индексирования

Операция извлечения элемента из массива, обозначаемая квадратными скобками, как известно, является арифметической операцией над указателем и целым числом; в языке Си выражение `a[b]` полностью эквивалентно выражению `*(a+b)`, что лишний раз подчёркивает арифметическую сущность индексирования. В языке Си++ операцию индексирования можно переопределить для объектов класса или структуры, тем самым заставив объект в некоторых случаях выглядеть синтаксически похожим на обычный массив или даже просто исполнять роль массива. Подобно операциям присваивания, **операция индексирования может быть переопределена только как метод класса или структуры**.

Для примера опишем класс, реализующий массив целых чисел с заранее неизвестным размером, который при обращении к несуществующим (пока) элементам автоматически увеличивается в размерах. Хранить элементы «массива» будем в динамически создаваемом массиве (настоящем), скрытом в приватной части класса. Исходно создадим массив размером 16 элементов, а при возникновении такой необходимости будем удваивать его размеры до тех пор, пока нужный нам индекс не станет допустимым.

При попытке копирования объекта такого класса возникнет проблема с разделением одного и того же массива в динамической памяти между двумя объектами класса; подробно эта проблема описана в §10.4.10. В нашей упрощённой версии мы просто запретим присваивание и копирование, описав фиктивные конструктор копирования и операцию присваивания в приватной части. Описание полноценного конструктора копирования и полноценной операции присваивания оставим читателю в качестве упражнения. Для начала напишем заголовок класса:

```
class IntArray {
    int *p;             // указатель на хранилище
    unsigned int size; // текущий размер хранилища
public:
    IntArray() {
        size = 16;
        p = new int[size];
    }
    ~IntArray() { delete[] p; }
    int& operator[](unsigned int idx);
private:
    void Resize(unsigned int required_index);
        // запретим копирование и присваивание
    void operator=(const IntArray& ref) {}
    IntArray(const IntArray& ref) {}
};
```

Тела конструктора и деструктора имеют сравнительно небольшой размер, поэтому мы совершенно спокойно можем оставить их внутри заголовка класса. Тело операции индексирования, напротив, будет состоять из нескольких строк, поэтому мы опишем его отдельно. Для лучшей ясности его реализации мы предусмотрели также вспомогательную функцию `Resize`, которая будет осуществлять изменение размера массива. Поскольку эта функция, очевидно, является деталью реализации и не предназначена для пользователя (с точки зрения пользователя наш массив представляется бесконечным), мы скрыли эту функцию в приватной части класса.

Читатель, возможно, обратил внимание на тип возвращаемого значения операции индексирования. Функция `operator[]` возвращает *ссылку* на соответствующий элемент массива, чтобы сделать возможным как выборку значения из массива, так и присваивание его элементам новых значений. Опишем теперь тело операции индексирования:

```
int& IntArray::operator[](unsigned int idx)
{
    if(idx >= size)
        Resize(idx);
    return p[idx];
}
```

Нам осталось описать функцию `Resize`:

```
void IntArray::Resize(unsigned int required_index)
{
    unsigned int new_size = size;
    while(new_size <= required_index)
        new_size *= 2;
    int *new_array = new int[new_size];
    for(unsigned int i = 0; i < size; i++)
        new_array[i] = p[i];
    delete[] p;
    p = new_array;
    size = new_size;
}
```

Теперь мы можем в программе использовать, например, такие конструкции:

```
    IntArray arr;
    arr[500] = 15;
    arr[1000] = 30;
    arr[10] = arr[500] + 1;
    arr[10]++;
```

Операция индексирования должна иметь ровно один параметр, но этот параметр, вообще говоря, может быть любого типа. Это позволяет создавать «массивы», использующие в качестве индекса текстовые строки или даже объекты других классов. Хранить элементы массива мы также можем любым удобным нам способом, например, в виде списка (в некоторых случаях это может быть оправданно). Можно организовать объект, выглядящий как массив, но хранящий свои элементы в файле на диске, так что операция индексирования реально будет представлять собой операцию чтения или записи в файл; встречаются и другие применения.

Заметим теперь, что объекты нашего класса не годятся для работы в константном варианте. Если такой объект передать в функцию через указатель или ссылку с модификатором `const`, мы немедленно обнаружим, что внутри функции не можем с этим объектом сделать *вообще ничего*: единственный метод, через который с объектом строится всё взаимодействие, модификатора `const` не имеет и, как следствие, не может быть вызван для объекта, который запрещено изменять. При этом вполне очевидно, что просто взять и навесить `const` на нашу операцию индексирования не получится: во-первых, из её тела вызывается метод `Resize`, который производит модификацию объекта в силу своей природы, а во-вторых, возвращает она неконстантную ссылку на *внутренности* объекта, так что через эту ссылку объект потом может быть изменён.

Самое время припомнить замечание, которое мы сделали в §10.4.1 при обсуждении перегрузки функций: два типа, различающиеся только наличием или отсутствием `const`, считаются разными, в том числе и с точки зрения механизма перегрузки. Чтобы понять, какое отношение это имеет к нашей ситуации с операцией индексирования, стоит вспомнить о существовании неявного параметра `this`; после этого, возможно, для нас не станет сюрпризом, что Си++ допускает в одном классе или структуре существование двух разных методов, отличающихся только словом `const` в конце заголовка: для неконстантных объектов компилятор будет в таком случае вызывать метод без `const`, для константных — его же const-версию.

Это позволяет нам написать отдельную версию операции индексирования, которая будет применяться к таким объектам `IntArray`, изменение которых запрещено. Возвращать в такой ситуации, конечно, следует уже не ссылку на элемент массива, а его значение, так что компилятор не позволит к результату вызова нашего метода применить присваивание или другую модифицирующую операцию. Кроме того, в этом варианте не следует, очевидно, пытаться менять размер массива, если затребованный индекс оказался слишком большим; в самом деле, мы ведь вообще не должны ничего менять в объекте, таковы условия. Остаётся вопрос, какое значение вернуть в таком случае. За неимением лучшей идеи мы вернём ноль, хотя в зависимости от конкретной задачи возможны другие решения.

Итак, в заголовок класса добавляем (перед директивой `private`) следующую строку:

```
int operator[](unsigned int idx) const;
```

Реализация новой версии метода будет такой:

```
int IntArray::operator[](unsigned int idx) const
{
    if(idx >= size)
        return 0;
    return p[idx];
}
```

Теперь, получив в своё распоряжение объект класса `IntArray`, не подлежащий модификации, мы всё же сможем с ним работать, нельзя будет только поменять его состояние.

Конечно, такая «перегрузка по `const`» возможна для любых методов, не только для операции индексирования, просто в других случаях она обычно *не нужна*. Впрочем, на практике встречается и такое, хотя и редко.

В завершение обсуждения нашего псевдомассива рискнём предостеречь читателя от практического применения подобных классов и объектов — таких, которые не просто «выглядят» похожими на массивы, но и действительно реализуют функциональность обыкновенного массива. Такие объекты обычно называют ***контейнерами***. Во-первых, **трудозатраты на поддержание самого контейнера, на таскание его реализации за собой из проекта в проект и т. п.** сводит на нет любые выгоды от его применения, а неизбежно возрастающая сложность отладки и сопровождения может привести к накладным расходам, *на порядки* превышающим копеечную экономию времени на стадии кодирования. Во-вторых, **сама идея использовать закрытый объект как хранилище неких элементов изначально ущербна**: хранилище (такое, например, как обычный массив) должно быть *прозрачно* для своего владельца, тогда как объект (неважно, из какой он пришёл парадигмы — ООП или АТД) обязан быть наглухо закрытым.

10.4.21. Переопределение операций ++ и --

Переопределение операций инкремента и декремента имеет свои особенности, обусловленные тем, что в Си и Си++ каждая из операций `++` и `--` допускает две формы: префиксную (`++i`) и постфиксную (`i++`). Строго говоря, эти две формы представляют собой *различные* операции. Если бы не это, можно было бы переопределять инкремент и декремент как обычные унарные операции, то есть вне классов в виде функции от одного аргумента, либо в классе/структуре в виде метода без параметров.

Чтобы отличать друг от друга переопределение префиксной и постфиксной формы, в Си++ принято соглашение, выглядящее довольно неожиданно. Поскольку префиксная форма более традиционна для унарных операций, обычным способом переопределяется именно она. Иначе говоря, если мы переопределяем соответствующие операции введением метода в классе или структуре, то метод с именем `operator++` или `operator--` без параметров определяет *префиксную* форму соответствующей операции (например, `++i`); глобальная функция (не метод), имеющая имя `operator++` (`operator--`) и *один* параметр, тоже соответствует префиксной форме операции инкремента (декремента).

Что касается *постфиксной* формы, то она переопределяется функцией с тем же именем `operator++` (`operator--`), но имеющей дополнительный (фиктивный) параметр типа `int`. Наличие параметра благодаря перегрузке имён функций позволяет иметь две функции с одним и тем же именем; значение параметра, введённого исключительно ради этого различия, реально никогда не используется. Для переопределения постфиксной формы операции инкремента или декремента мы можем воспользоваться либо методом класса/структуры с именем `operator++` (`operator--`) и *одним* параметром (фиктивным), либо глобальной функцией с таким же именем, имеющей *два* параметра (второй из них фиктивный). Например, пусть класс `A` описан следующим образом:

```
class A {
public:
    void operator++() { printf("first\n"); }
    void operator--() { printf("second\n"); }
    void operator++(int) { printf("third\n"); }
    void operator--(int) { printf("fourth\n"); }
};
```

Рассмотрим теперь фрагмент кода:

```
A a;
++a;   // first
a++;   // third
--a;   // second
a--;   // fourth
```

В результате выполнения этого фрагмента будут напечатаны (в столбик) слова `first`, `third`, `second` и `fourth` — именно в таком порядке.

В рассмотренном примере наши операции ничего не возвращали; между тем исходно префиксная и постфиксная формы этих операций различаются именно возвращаемым значением. При этом операция в префиксной форме возвращает *новое* значение переменной, к которой она применена, что позволяет выдержать семантику этой операции, вернув сам объект или (для оптимизации) константную ссылку на него. С операцией в постфиксной форме дела обстоят несколько хуже: согласно классической семантике она должна вернуть *старое* значение, в то время как объект уже имеет *новое* значение. Чтобы выдержать семантику в этой форме, приходится в теле функции `operator++(int)` создавать временную (локальную) копию объекта, а возвращать значение, сохранённое в этой копии, приходится обязательно *по значению*, ведь возвращать ссылку на локальную копию было бы ошибкой: копия исчезнет в момент завершения работы функции, то есть *до того, как вызывающая функция успеет воспользоваться возвращённым результатом*. Проиллюстрируем сказанное на примере операций инкремента для класса, инкапсулирующего целочисленную переменную и повторяющего её семантику:

```
class MyInt {
    int i;
public:
    MyInt(int x) : i(x) {}
    const MyInt& operator++() { i++; return *this; }
    MyInt operator++(int)
        { MyInt tmp(*this); i++; return tmp; }
    // ...
```

В этом примере объект класса занимает столько же памяти, сколько и обычная целочисленная переменная, так что двойное копирование объекта (при создании переменной `tmp` и при возврате значения) не приводит к существенным потерям. Однако для более сложных классов педантичное следование классической семантике для постфиксных операций `++` и `--` может обойтись неоправданно дорого.

10.4.22. Переопределение операции `->`

Операция `->` переопределяется, пожалуй, наиболее экзотическим способом. Напомним, что эта операция в языке Си означает выборку поля из структуры, на которую указывает заданный адрес. В Си++ она означает практически то же самое; естественно, с её помощью можно также обращаться и к методам, а работать она может как со структурами, так и с классами. Перегрузка этой операции обычно требуется нам, если мы хотим создать объект, ведущий себя подобно указателю, но при этом выполняющий некие дополнительные действия.

Напомним для начала один неочевидный факт, с которым мы уже встречались при изучении чистого Си (см. т. 2, §4.5.1). Операция `->`, несмотря на её внешний вид, считается *унарной*, поскольку имеет всего один аргумент (указатель). Действительно, пусть у нас описана структура

```
struct s1 {
    int a, b;
};
```

и имеется указатель на неё:

```
s1 *p = new s1;
```

Теперь обращение к полю `a` будет выглядеть так: `p->a`. Здесь как будто бы два операнда, `p` и `a`; но ведь `a` — это *имя поля*, которое никоим образом не является самостоятельным выражением, не имеет ни типа, ни значения! Иначе говоря, на месте `a` не может стоять ничего, кроме имени поля. Язык не содержит никаких выражений, *вычисляющихся* в значение, способное заменить имя поля. Единственным полноценным операндом в выражении `p->a` остаётся адрес, представленный указателем `p`: в отличие от `a`, на месте `p` может стоять выражение произвольной сложности, вычисляющее значение типа `s1*` (адрес структуры `s1`). Итак, мы имеем дело с *унарной* операцией, полным именем которой можно считать `->a` (операция выборки поля `a`). Можно при желании считать, что символ `->` задаёт целое семейство операций — столько же, сколько в структуре есть полей (а для Си++ — ещё и методов). Впрочем, будучи семантически безупречным, на практике такое рассмотрение нам не понадобится; здесь мы приводим его только для иллюстрации сказанного.

Вернёмся к вопросу о переопределении операции `->`. Наряду с многими другими операциями `->` переопределяется исключительно методами класса или структуры, то есть не может быть переопределена отдельной функцией. Чтобы избежать рутинной работы по переопределению отдельных операций для выборки каждого поля, в Си++ принято неочевидное, но вполне работающее соглашение: операция `->` определяется методом `operator->`, не имеющим параметров (как и обычные унарные операции), но при этом метод обязан возвращать либо *адрес некой другой структуры или класса*, либо объект (или ссылку на объект), для которого, в свою очередь, операция `->` переопределена как метод. При использовании перегруженной операции `->` компилятор вставляет в код последовательно вызовы методов `operator->`, пока очередной метод не окажется возвращающим обычный адрес структуры или класса; именно по этому адресу и производится в итоге выборка заданного поля. Впрочем, чаще всего `operator->` сразу же возвращает простой адрес, так что никаких последовательностей вызовов не требуется.

Попробуем проиллюстрировать сказанное на примере. Пусть у нас есть структура `s1` и нам нужен класс, объекты которого будут вести себя как указатели на `s1`, но при исчезновении такого «указателя» (например, при завершении функции, в которой он описан как локальная переменная) объект, на который он указывает, будет уничтожаться. Начнём описание класса:

```
class Pointer_s1 {
    s1 *p;
public:
```

Будем предполагать, что хранящийся внутри объекта простой указатель на `s1` может быть нулевым, что следует расценивать как отсутствие объекта; естественно, в этом случае удалять ничего не будем. Опишем теперь конструктор и деструктор:

```
    Pointer_s1(s1 *ptr = 0) : p(ptr) {}
    ~Pointer_s1() { if(p) delete p; }
```

Для удобства работы введём операцию присваивания обычного адреса:

```
    s1 *operator=(s1 *ptr) {
        if(p)
            delete p;
        p = ptr;
        return p;
    }
```

Заметим, что побитовое копирование и присваивание объектов класса `Pointer_s1` заведомо приведёт к ошибкам, так как один и тот же объект типа `s1` в этих случаях будет удаляться дважды. Во избежание этого запретим присваивание и копирование объектов класса `Pointer_s1`, убрав конструктор копирования и соответствующую операцию присваивания в приватную часть:

```
private:
    Pointer_s1(const Pointer_s1&) {}     // no copying
    void operator=(const Pointer_s1&) {} // no assignments
```

Опишем теперь операции, которые превратят объекты нашего класса в подобие указателя на `s1`, а именно операции разыменования (унарную `*`) и выборки поля (`->`), и завершим описание класса:

```
public:
    s1 &operator*() const { return *p; }
    s1 *operator->() const { return p; }
};
```

Такой класс удобно использовать, например, внутри функций. Так, если в начале некоторой функции описать объект класса `Pointer_s1`, то ему можно будет присваивать адреса новых экземпляров `s1`, при этом он будет каждый раз удалять старый экземпляр, а когда работа функции завершится, автоматически удалит последний из экземпляров `s1`:

```
int f()
{
    Pointer_s1 p;
    p = new s1;
    p->a = 25;
    p->b = p->a + 36;
    // ...
    // при завершении f память будет освобождена
}
```

10.4.23. Переопределение операции вызова функции

Операция вызова функции, синтаксически представляющая собой постфиксную операцию, символом которой служат круглые скобки (возможно, содержащие список параметров), может быть, как и другие операции, переопределена. Поскольку вызов функции обозначается круглыми скобками, имя функции, переопределяющей эту операцию, будет состоять из слова `operator` и круглых скобок; как обычно, после имени функции записывается список формальных параметров. Например, операция вызова функции без параметров записывается так:

```
void operator()() { /* тело */ }
```

Подобно операциям присваивания и индексирования, вызов функции переопределяется только методом класса.

Для примера опишем класс `Fun`, в котором переопределены операции вызова функции без параметров, а также с одним и двумя целочисленными параметрами:

```
class Fun {
public:
    void operator()()
        { printf("fun0\n"); }
    void operator()(int a)
        { printf("fun1: %d\n", a); }
    void operator()(int a, int b)
        { printf("fun2: %d %d\n", a, b); }
};
```

Теперь мы можем написать следующий фрагмент кода:

```
Fun f;
f(); f(100); f(25, 36);
```

В результате выполнения этого фрагмента будет напечатано:

```
fun0
fun1: 100
fun2: 25 36
```

Классы, для которых определена операция вызова функции — иначе говоря, классы, объекты которых можно использовать подобно именам функций — часто называют ***функторами***.

10.4.24. Переопределение операции преобразования типа

Рассмотрим следующий фрагмент кода:

```
int i;
double d;
// ...
i = d;
```

В строчке, содержащей присваивание, задействована (неявно) ***операция преобразования типа выражения***, позволяющая в данном случае построить значение типа `int` на основе имеющегося значения типа `double`. Аналогичную возможность неявного преобразования можно предусмотреть и для классов, введённых программистом. Один способ мы уже знаем — это конструктор преобразования (см. §10.4.4); но в некоторых случаях применить конструктор преобразования не удаётся. В частности, конструктором можно задать преобразование из базового типа в тип, описанный пользователем, но не наоборот. Кроме того, иногда бывает по каким-то причинам невозможно изменить описание некоторого класса, но в программе нужно преобразование в объекты этого класса. Бывают и другие (довольно экзотические) ситуации, когда мы не можем задать способ преобразования путём модификации того типа, *к которому* производится преобразование, но при этом у нас есть возможность изменить (дополнить) описание того типа, *значения которого подлежат преобразованию*. В таких случаях применяют переопределение операции преобразования типа.

Операция неявного преобразования типов определяется методом, имя которого состоит из слова `operator` и имени типа, к которому будет происходить преобразование. Тип возвращаемого значения для такой функции не указывается, так как он определяется именем. Например:

```
class A {
    // ...
public:
    // ...
    operator int() const { /* ... */ }
};
```

Наличие в классе `A` операции преобразования к `int` делает возможным, например, такой код:

```
A a;
int x;
// ...
x = a;
```

Естественно, операция преобразования может быть переопределена только как метод класса или структуры.

Учтите, что чрезмерное увлечение переопределениями операции преобразования часто приводит к тому, что компилятор находит больше одного способа преобразования от одного типа к другому и в результате не применяет ни одного из них, выдавая ошибку. Поэтому на практике переопределение операции преобразования типа почти не используется; для получения значений нужного типа из объектов классов или структур обычно применяют специально для этого вводимые методы (обыкновенные, без слова `operator`), вызываемые явно. Пример крайне редкой ситуации, когда операция преобразования типа оказывается действительно нужна, мы рассмотрим в следующем параграфе.

10.4.25. Пример: разреженный массив

Под ***разреженным массивом*** понимается крупный (например, состоящий из нескольких миллионов элементов) массив, большинство элементов которого равны одному и тому же значению, чаще всего нулю, и лишь некоторые элементы, количество которых очень мало в сравнении с размером массива, от этого значения отличаются. Естественно, в такой ситуации целесообразно хранить в памяти лишь те элементы массива, значения которых отличны от нуля (или другого общего значения).

Попробуем написать на Си++ такой класс, который ведёт себя как целочисленный массив потенциально бесконечного размера[14] в том смысле, что к объекту этого класса применима операция индексирования (квадратные скобки), однако объект реально хранит только те элементы, значения которых отличаются от нуля. Назовём этот класс `SparseArrayInt`. Для простоты картины будем считать, что количество ненулевых элементов массива настолько незначительно, что даже линейный поиск среди них может нас устроить по быстродействию. Это

[14] На самом деле мы будем использовать в качестве индекса тип `int`, надеясь, что его разрядности хватит; поменять его на какой-нибудь `long long` — дело техники, а если не хватит и этого, то вы, скорее всего, делаете что-то не то.

позволит хранить ненулевые элементы массива в виде списка пар «индекс/значение». Все детали реализации, включая и структуру, представляющую звено списка, скроем в приватной части класса, чтобы в будущем можно было безболезненно сменить реализацию на более сложную.

Сразу же припомним сказанное в §10.4.20 насчёт возможности работы с константными объектами и договоримся, что в нашем классе будут реализованы два варианта операции индексирования: обычная и «константная», допускающая вызов для константных объектов. С версией для константных объектов всё просто: нужно пробежаться по имеющемуся списку пар, и если там найдётся нужный индекс — вернуть соответствующее значение, если же мы прошли весь список и искомого индекса не обнаружили — вернуть наше «общее значение» (в нашем примере это всегда будет ноль).

Основная проблема при реализации разреженного массива возникает в связи с поведением операции индексирования, *допускающей* модификацию объекта. Выражение «элемент массива» может встречаться как в правой, так и в левой части присваиваний, и в обоих случаях используется одна и та же функция-метод класса `SparseArrayInt` — `operator[]`, причём, если только сам объект не константный, **компилятор в обоих случаях выберет тот вариант метода, который без слова** `const`. Если бы все элементы массива хранились в памяти, как это было в примере из §10.4.20, можно было бы просто возвратить ссылку на место в памяти, где хранится элемент, соответствующий заданному индексу. С разреженным массивом так действовать не получится, поскольку в большинстве случаев элемент в памяти не хранится. При этом просто возвращать значение (без всякой ссылки) тоже не годится, ведь тогда выражение, содержащее нашу операцию индексирования, будет нельзя применять в левой части присваивания.

Итак, что же делать в случае, если операция индексирования вызвана для индекса, для которого элемент в памяти не хранится, поскольку равен нулю? Одно из возможных решений — завести в памяти соответствующий элемент, пусть даже и с нулевым значением, и вернуть ссылку на него. Это, конечно, позволит выполнять присваивания, но приведёт к тому, что при каждом обращении к нулевым элементам (в том числе на чтение, а не на запись) такие элементы будут заводиться в памяти, и вскоре окажется, что изрядное количество памяти занято теми самыми нулевыми значениями, хранения которых требовалось избежать.

Схему можно несколько усовершенствовать, если каждый раз при вызове операции индексирования сохранять в приватной части объекта значение индекса или даже адрес заводимого звена списка. Тогда при следующем вызове можно будет проверить, хранит ли запись, заведённая при предыдущем вызове, число, отличное от нуля, и если нет, то удалить её, освободив память. У такого варианта реализации есть, однако, свой недостаток: на заведение и удаление элементов будет расходоваться заметное время.

В идеале хотелось бы иметь возможность вызывать разные функции-методы для случаев, когда происходит обращение к элементу массива на чтение и когда индексирование используется для присваивания элементу нового значения. В первом случае можно было бы тогда возвращать просто значение элемента или ноль, если элемента в памяти нет; во втором случае можно было бы без особых проблем создавать новый элемент и возвращать ссылку на поле, хранящее значение; это, впрочем, не исключает необходимости проверки при следующем вызове, не был ли предыдущему элементу присвоен ноль (в том числе для случаев, когда элемент уже существовал). К сожалению, возможности введения разных методов индексирования для этих двух случаев в Си++ нет, как нет и возможности определить из тела операции индексирования, вызвана она из левой части присваивания или нет. Это вполне можно считать недостатком Си++.

Впрочем, такой механизм можно *смоделировать*; именно так мы и поступим. Никто не мешает нам описать небольшой вспомогательный класс, объекты которого как раз и будет возвращать наша операция индексирования. Объект этого класса будет просто хранить всю имеющую отношение к делу информацию, а в нашем случае такой информации немного: адрес главного объекта (то есть объекта разреженного массива), с которым нужно работать, и значение индекса, переданное в качестве параметра операции индексирования.

Имея эту информацию и доступ к объекту массива, наш вспомогательный объект в зависимости от ситуации может произвести любые действия, которые потребуются. Что же касается определения, какая из возможных ситуаций имеет место, то у вспомогательного объекта для этого есть существенно больше возможностей, чем было в теле операции индексирования: ведь все операции, применяемые к результату операции индексирования, как раз и будут на самом деле применены к этому объекту. Например, достаточно определить для нашего вспомогательного класса операции преобразования к типу `int` и присваивания, чтобы сделать возможным и корректным следующий код:

```
SparseArrayInt arr;
int x;
// ...
x = arr[500];  // чтение
arr[300] = 50; // запись (возможно создание элемента)
arr[200] = 0;  // удаление элемента, если таковой есть
```

К сожалению, недостаток есть и у такой реализации. Проблема в том, что обычное присваивание — это далеко не единственная операция, способная изменить значение целочисленной переменной. Пользователь вполне может попытаться использовать, например, такие выражения:

```
arr[15] += 100;
x = arr[200]++;
y = --arr[200];
```

Чтобы все подобные операции работали, нам придётся для нашего вспомогательного объекта определить их все явным образом, что потребует изрядного объёма работы: языки Си и Си++ поддерживают десять операций присваивания, совмещённых с арифметическими действиями (`+=`, `-=`, `<<=` и т. п.), плюс к тому четыре операции инкремента/декремента (`++` и `--` в префиксной и постфиксной форме). В нашем примере мы ограничимся описанием операции `+=` и двух форм операции `++`. Остальные операции читатель без труда опишет самостоятельно по аналогии.

Для удобства реализации всех модифицирующих операций целесообразно в классе вспомогательного объекта ввести две внутренние функции. Функция `Provide` будет отыскивать в списке элемент, соответствующий заданному в объекте индексу, и возвращать ссылку на поле, содержащее значение; если соответствующего элемента в списке нет, функция будет его создавать. Вторая функция, `Remove`, будет удалять из списка элемент с индексом, хранящимся в объекте, если таковой в списке присутствует. Вспомогательный класс назовём `Interm` от английского *intermediate*.

Отметим, что копировать и присваивать объекты класса `SparseArrayInt` — идея вряд ли правильная; скорее всего, если что-то такое происходит, то это ошибка — например, пользователь случайно передал объект класса в функцию по значению, а не по ссылке. Поэтому мы запретим копирование и присваивание объектов нашего класса, убрав соответствующие методы в приватную часть. С учётом этого заголовок класса будет выглядеть так:

```
class SparseArrayInt {
    struct Item {
        int index;
        int value;
        Item *next;
    };
    Item *first;
public:
    SparseArrayInt() : first(0) {}
    ~SparseArrayInt();
    class Interm {
        friend class SparseArrayInt;
        SparseArrayInt *master;
        int index;
```

```
        Interm(SparseArrayInt *a_master, int ind)
            : master(a_master), index(ind) {}
        int& Provide();
        void Remove();
    public:
        operator int();
        int operator=(int x);
        int operator+=(int x);
        int operator++();
        int operator++(int);
    };
    friend class Interm;

    Interm operator[](int idx)
        { return Interm(this, idx); }
    int operator[](int idx) const;
private:
    SparseArrayInt(const SparseArrayInt&) {}
    void operator=(const SparseArrayInt&) {}
};
```

В самом классе `SparseArrayInt`, как можно заметить, оказывается не так много функций: бóльшая часть реализации оказывается вытеснена в методы вспомогательного класса. Вне заголовка класса мы опишем только деструктор и «константную» версию операции индексирования:

```
SparseArrayInt::~SparseArrayInt()
{
    while(first) {
        Item *tmp = first;
        first = first->next;
        delete tmp;
    }
}

int SparseArrayInt::operator[](int idx) const
{
    Item *tmp;
    for(tmp = first; tmp; tmp = tmp->next)
        if(tmp->index == idx)
            return tmp->value;
    return 0;
}
```

Опишем теперь методы класса `SparseArrayInt::Interm`, в которых и заключена основная функциональность нашего разреженного массива:

```
int SparseArrayInt::Interm::operator=(int x)
{
    if(x == 0)
        Remove();
    else
        Provide() = x;
    return x;
}

int SparseArrayInt::Interm::operator+=(int x)
{
    int& location = Provide();
    location += x;
    int res = location;
    if(res == 0)
        Remove();
    return res;
}

int SparseArrayInt::Interm::operator++()
{
    int& location = Provide();
    location++;
    int res = location;
    if(location == 0)
        Remove();
    return res;
}

int SparseArrayInt::Interm::operator++(int)
{
    int& location = Provide();
    int res = location;
    location++;
    if(location == 0)
        Remove();
    return res;
}

SparseArrayInt::Interm::operator int()
{
    Item *tmp;
    for(tmp = master->first; tmp; tmp = tmp->next)
        if(tmp->index == index)
            return tmp->value;
    return 0;
}
```

Осталось описать вспомогательные функции `Provide` и `Remove`:

```
int& SparseArrayInt::Interm::Provide()
{
    Item *tmp;
    for(tmp = master->first; tmp; tmp = tmp->next)
        if(tmp->index == index)
            return tmp->value;
    tmp = new Item;
    tmp->index = index;
    tmp->next = master->first;
    master->first = tmp;
    return tmp->value;
}

void SparseArrayInt::Interm::Remove()
{
    Item **tmp;
    for(tmp = &(master->first); *tmp; tmp = &(*tmp)->next) {
        if((*tmp)->index == index) {
            Item *to_delete = *tmp;
            *tmp = (*tmp)->next;
            delete to_delete;
            return;
        }
    }
}
```

10.4.26. Статические поля и методы

Иногда бывают полезны переменные и методы, которые, с одной стороны, доступны только из класса и/или воспринимаются как его часть, но, с другой стороны, не привязаны ни к какому из объектов и могут использоваться даже тогда, когда ни одного объекта данного класса не существует. В языке Си++ такие члены класса называются ***статическими*** и обозначаются ключевым словом `static`[15].

Статическое поле класса представляет собой, по сути, обычную переменную, время жизни которой совпадает с временем выполнения программы — как у глобальных переменных. При этом правила видимости для статического поля класса точно такие же, как и для обычных полей: статическое поле входит в область видимости класса, так что при упоминании статического поля вне методов класса требуется использовать символ раскрытия области видимости «`::`» (см. §10.4.13). На статическое поле распространяются обычные правила защиты: если его расположить в приватной части класса, доступ к такому полю будет возможен только из методов класса и друзей класса, если же статическое поле оставить в публичной части, мы получим просто глобальную переменную с необычно выглядящим именем.

[15] Не путайте их с локальными переменными и функциями модулей, а также с локальными переменными функций, которые сохраняют своё значение при повторном входе в функцию; как и в языке Си, в Си++ эти сущности тоже обозначаются словом `static`, но практически ничего общего не имеют со статическими членами класса.

Сущность статических полей класса можно пояснить и иначе: статическое поле — это такое «хитрое» поле, которое существует в одном экземпляре для всего класса, вне зависимости от количества объектов этого класса; даже если ни одного объекта нет, статическое поле всё равно существует, коль скоро оно описано в классе. Получается, что статическое поле, в отличие от обычного, не является составной частью *объекта* и принадлежит *классу* как единому целому.

Статическое поле объявляется в классе точно так же, как и обычное поле класса, только с добавлением спереди слова `static`:

```
class A {
    // ...
    static int the_static_field;
    // ...
};
```

Нужно учесть очень важное отличие статического поля от обычного: если объявление обычного поля само по себе уже означает, что в объекте под это поле будет выделено место, то объявление статического поля — это именно объявление, само по себе оно не предполагает выделения памяти; статическое поле требуется ещё *описать* в одном из модулей.

Припомнив язык Си, можно заметить здесь прямую аналогию с объявлением глобальных переменных с директивой `extern` (см. т. 2, §4.7.2). Чтобы понять, почему объявление статического поля в классе считается именно объявлением, а не описанием, подумайте, что будет, если заголовок класса вынести в заголовочный файл, а сам этот файл — включить из нескольких модулей.

Итак, чтобы завершить создание статического поля, нужно *вне заголовка класса* поместить его описание:

```
int A::the_static_field = 0;
```

Обычно определения статических полей снабжают инициализацией, как в нашем примере, хотя это и не обязательно. Обратите внимание, что если ваше описание класса вынесено в заголовочный файл, то **описания статических полей следует обязательно поместить в файл реализации одного из модулей**, ни в коем случае не в заголовочник!

Повторим, что введённое таким образом статическое поле будет существовать в программе **в единственном экземпляре и в течение всего времени выполнения программы** вне всякой зависимости от того, будут ли в вашей программе заводиться объекты класса `A` и в каких количествах. Если поместить объявление статического поля в приватной части класса, то соответствующее имя будет доступно только в методах класса и в дружественных функциях. Объявление можно поместить и в открытую часть класса, тогда оно будет доступно отовсюду, а обратиться к нему можно будет как через существующий объект, так и без всякого объекта с помощью явного раскрытия области видимости:

```
A a;
a.the_static_field = 15;  // правильно
A::the_static_field = 15; // тоже правильно
```

Статические поля схожи с глобальными переменными в том числе и тем, что в них программа может *накапливать глобальное состояние*, в результате чего для стороннего наблюдателя поведение одних и тех же функций окажется изменяющимся по неочевидным законам. Кроме того, статические поля могут существенно затруднить масштабирование программы. Так, если вам зачем-то понадобился список объектов определённого класса и вы ввели для этой цели статический указатель на первый элемент такого списка, то это означает, что в программе такой список может быть только один, а если когда-нибудь понадобится два таких списка (из объектов одного и того же типа), программу придётся существенно переделать.

Поэтому **использование статических полей рекомендуется в одной и только одной ситуации: когда такое поле представляет собой константу, то есть его значение никогда не изменяется во время работы программы**. Например, статическим полем можно представить какой-нибудь крупный массив, содержащий *неизменные* данные, необходимые для работы объектов данного класса, но не требующиеся нигде за его пределами. Примером такого массива могут служить, например, таблица переходов между состояниями конечного автомата в классе, реализующем этот конечный автомат; массив строк с возможными диагностическими сообщениями; таблица соответствия пользовательских команд (строк) вызываемым функциям, и т. п.

Статический метод — это особый вид функции-метода, которая, являясь методом класса и имея доступ к его закрытым деталям реализации, при этом **вызывается независимо от объектов класса**. Первоначально статические методы предназначались для работы со статическими полями, но получившийся механизм нашёл в итоге существенно более широкий спектр применений. Объявление статического метода аналогично объявлению обычного метода, но перед ним ставится ключевое слово `static`, как в следующем примере:

```
class Cls {
    // ...
    static int TheStaticMethod(int a, int b);
    // ...
};
```

Отметим, что при выносе тела статического метода за пределы заголовка класса слово `static` в его описании не используется, компилятору хватает слова `static`, поставленного в объявлении (заголовке) метода в классе.

Обращение к статическому методу, как и к статическому полю, возможно как через объект класса, так и без такового, с помощью символа раскрытия области видимости:

```
Cls::TheStaticMethod(5, 15); // всё правильно
Cls c;
c.TheStaticMethod(5, 15); // так тоже можно
```

Поскольку статическая функция-метод может быть вызвана без объекта, у неё, как следствие, отсутствует неявный параметр `this` (см. §10.3.2). Кроме всего прочего это означает, что такая функция не может просто так обращаться, как другие методы, к полям объекта, ведь объекта у неё нет. Вызывать нестатические функции-методы статическая функция тоже просто так не может, поскольку нестатические методы должны вызываться для конкретного объекта. Ситуация кардинально меняется, если статическая функция тем или иным способом всё-таки получает доступ к объекту своего класса. Такое вполне может произойти и никоим образом не противоречит определению статического метода: действительно, никто не мешает передать объект через один из явно обозначенных параметров; вполне возможно получить доступ к объекту своего класса через глобальные переменные или с помощью глобальных функций; наконец, **статическая функция вполне может создать объект сама**.

Итак, несмотря на отсутствие указаний на конкретный объект при вызове статической функции, она в некоторых случаях может получить доступ к объекту своего класса. И тогда **статическая функция, как и любой метод класса, может обращаться к закрытым (приватным) полям и методам объекта** (действительно, как уже говорилось ранее, единицей защиты в Си++ является не объект, а класс или структура целиком). Наличие в Си++ статических методов позволяет в ряде случаев применять весьма изящные приёмы программирования. Например, мы вполне можем убрать в закрытую часть класса все конструкторы, запретив таким образом создание объектов данного класса извне его самого, и поручить создание объектов статическому методу, который можно вызвать, не имея ни одного объекта.

10.5. Обработка исключительных ситуаций

Прежде чем начать рассказ об обработке исключений в Си++, мы должны предостеречь читателя от возможных последствий некритического восприятия этого инструмента. Общая идея исключительных ситуаций, как вы увидите чуть позже, чрезвычайно полезна, но то, как этот инструмент представлен в языке Си++, вызывает определён-

ное недоумение. Чтобы реализовать передачу управления в строгом соответствии со спецификацией Си++, компилятор вынужден идти на неочевидные и дорогостоящие трюки — вставлять в программу большое количество практически бесполезного машинного кода, использовать громоздкую «невидимую» библиотечную поддержку, снижать быстродействие; коль скоро мы упомянули быстродействие, отметим, что если уж ваша программа перешла в состояние обработки исключительной ситуации, то происходить это будет по компьютерным меркам просто *безумно* медленно.

Многие программисты, даже не относящие себя к консерваторам, подобным автору этих строк, считают использование исключений недопустимым; большинство компиляторов Си++ позволяет полностью отключить этот механизм. Возможно, при минималистическом подходе, который мы приняли в нашей книге в отношении Си++, логичнее было бы вообще не рассказывать об исключениях, и такой вариант, смеем вас заверить, рассматривался весьма серьёзно, но, несмотря на всё сказанное, эта глава в книге сохранилась, и тому есть целый ряд причин.

Прежде всего следует отметить, что использование исключений существенно снижает трудоёмкость написания программ, во всяком случае, если сравнивать с программами, в которых, несмотря на отсутствие механизма исключений, все возникающие ошибочные ситуации скрупулёзно проверяются и обрабатываются. Можно ли ради такого выигрыша в трудоёмкости пожертвовать эффективностью — вопрос, на который следует отвечать, исходя из особенностей конкретной решаемой задачи.

Но есть здесь и другой момент, пожалуй, даже более важный. Недостатки, присущие механизму исключений языка Си++, обусловлены не только и не столько самой идеей исключений, сколько довольно странным подходом, который избрал для воплощения этой идеи Бьёрн Страуструп. Обработка исключений как таковая существует не только в Си++, и далеко не всегда она столь же убийственна для эффективности полученной программы. Можно легко представить себе другой подход, который ничуть не снизил бы выразительной мощи исключений, но при этом не заставлял бы программистов ломать копья по поводу допустимости его применения.

Для читателей, которые знают, о чём идёт речь, поясним: если бы конкретная исключительная ситуация идентифицировалась в Си++ не *типом объекта*, как это сделано сейчас, а *адресом переменной* произвольного типа, глобальной, динамической или даже стековой, для которой программист обязан гарантировать, что она не перестанет существовать ни к моменту возбуждения исключения, ни к моменту его обработки, то обработка исключений не тянула бы за собой динамического монстра, именуемого *системой идентификации типов во время исполнения* (RTTI), а пометка и размотка стека могли бы работать проще и быстрее.

С учётом всего этого можно сказать, что знакомство с обработкой исключений будет читателю полезно, даже если он примет для себя решение никогда не использовать её в программах на Си++. Существуют другие языки, предусматривающие аналогичные средства; в последующих частях нашей книги мы познакомимся с несколькими из этих языков. Кроме того, в будущем возможно (и, пожалуй, неизбежно) появление какого-то нового языка программирования; будем надеяться, что в нём всё-таки учтут негативный опыт Си++.

Сразу хотелось бы отметить ещё один момент. Если вы всё же решите использовать исключения в своих программах, то, по крайней мере, **никогда не применяйте механизм обработки исключительных ситуаций для *штатного* выхода из вложенных управляющих конструкций или глубоких цепочек вызовов функций**. Когда в вашей программе возникла та или иная ошибка, вам уже обычно всё равно, сколько времени уйдёт на её обработку: в большинстве случаев выход из положения требует вмешательства пользователя (человека), а его «перетормозить» надо ещё исхитриться. Совсем другое дело, если никакой ошибки нет. Исключения работают медленно до такой степени, что программа, использующая их для штатного возврата управления из вложенного вычислительного контекста, может (легко!) проводить в обработке исключений процентов этак 90 всего своего времени работы: коллеги рассказывали автору про реальные случаи, когда быстродействие программы повышалось более чем на порядок (конкретно — в семнадцать раз) из-за замены выхода по исключению на простой `return` и несколько глобальных флажков.

Как говорится, кто предупреждён — тот вооружён.

10.5.1. Ошибочные ситуации и проблемы их обработки

Любая сколько-нибудь нетривиальная программа содержит фрагменты, которые в некоторых обстоятельствах не могут отработать корректно. Например, программа, анализирующая содержимое заданного файла, не сможет работать, не сумев открыть файл на чтение; программа, работающая по компьютерной сети, не сможет работать, если не работает сеть или если недоступен нужный сервер; функция, производящая сложные вычисления, не может корректно их завершить, если в ходе вычислений потребовалось деление на ноль или, например, вычисление логарифма по единичному основанию, и т. п. Подобные ситуации называют ошибочными, однако же это совершенно не обязательно означает, что ошибся программист, писавший программу. В случае с файлом виноват в возникновении ошибочной ситуации, скорее всего, пользователь, в случае с сетью — обслуживающий персонал сети или сервера; деление на ноль может быть следствием ошибки программиста, но может стать также и результатом некорректных исходных данных. Иначе говоря, мы при написании программ часто сталкиваемся со случаями, когда успешная работа нашей программы зависит от внешних условий, которые мы сами гарантированно обеспечить не можем. В таких случаях приходится предусматривать в программе *обработку ошибок*.

Проверить все нужные условия обычно несложно. Существенно сложнее может оказаться следующий вопрос: а что же делать, если условия оказались неудовлетворительны — не открылся файл, не установилось соединение, в делителе оказался ноль, — то есть возникла та самая ошибочная ситуация? Многие студенты поступают просто, дёшево и сердито: печатают какое-нибудь сообщение (очень часто просто `"ERROR"`) и завершают выполнение программы, например, вызовом `exit`. В реальной жизни такой вариант обычно недопустим. Чтобы понять причины этой недопустимости, представьте себе, что вы долго набирали текст в каком-нибудь текстовом редакторе, потом при сохранении *случайно* ввели неправильное имя директории или, например, попытались записать файл на защищённый носитель. Если бы автор текстового редактора обрабатывал ошибки «по-студенчески», программа редактора бы немедленно завершилась, уничтожив все результаты вашей работы. Маловероятно, что такое могло бы вам понравиться.

Более того, не всегда и не везде можно так вот просто «напечатать сообщение». Например, при программировании оконного приложения под MS Windows никакого потока стандартного вывода в распоряжении программиста нет, так что вместо печати приходится создавать модальный диалог с соответствующим текстом и кнопками. Под ОС Unix всё не так плохо, поток стандартного вывода есть всегда, есть даже специальный поток для вывода сообщений об ошибках; проблема только в том, что далеко не всегда результаты вывода в эти потоки кто-то читает, и в некоторых случаях следует вместо них пользоваться системой журнализации. Если же написанный нами код работает в ядре операционной системы или это прошивка для какого-нибудь микроконтроллера, то возможность «выдать сообщение об ошибке» может попросту отсутствовать как явление.

Кроме того, не все ошибочные ситуации фатальны. В вышеупомянутом примере с редактором текстов в ошибке, допущенной пользователем, вовсе нет ничего страшного: нужно просто констатировать факт ошибки и попросить пользователя ввести другое имя файла. Точно так же может не быть фатальной ошибка, связанная с установлением сетевого соединения, ведь во многих случаях можно обратиться к серверу-дублёру[16]. Даже деление на ноль в некоторых случаях может оказаться делом поправимым, как, например, при исследовании некоторых некорректно поставленных задач: обычно в этих случаях головная программа знает, как можно скорректировать исходные данные, чтобы

[16]Такое возможно при обращениях к системе доменных имён (DNS), а в некоторых случаях — и при отправке электронной почты, и в других ситуациях тоже.

избежать деления на ноль, и нужную комбинацию ищет методом проб и ошибок. Наконец, один и тот же код может использоваться в совершенно разных программах, причём для одних ошибка может оказаться фатальной, для других — не стоящей даже упоминания. Например, многие программы выполняют поиск нужного файла в разных директориях, просто пытаясь его открыть и продолжая поиск дальше после каждой неудачи; в этом случае ошибка вообще перестаёт быть ошибкой, тогда как в какой-нибудь другой программе ошибка при открытии файла делает дальнейшее выполнение бессмысленным.

Из всего вышесказанного можно сделать один очень серьёзный и важный вывод: **принимать решение о том, что делать при возникновении ошибочной ситуации — это прерогатива головной программы**. Завершать программу, а также и выполнять тем или иным способом выдачу сообщений может только функция `main` и те из вызываемых ею функций, которые для этого специально предназначены. Что же касается кода, не попавшего в эту категорию, то его дело в случае возникновения ошибки — оповестить об этом головную программу, и не более того.

Звучит это достаточно просто, однако все, кто имеет опыт написания даже не очень сложных программ, знают, с каким трудом это воплощается на практике. В простых языках программирования, таких как Си или Паскаль, приходится из подпрограмм в случае возникновения ошибок возвращать специальные значения, а в точке вызова подпрограммы проверять, не вернула ли она значение, соответствующее ошибке. В большинстве случаев при этом вызвавшая подпрограмма тоже вынуждена возвращать признак ошибки и т. д. Представьте себе теперь, что мы пишем на Си и у нас функция `f1` вызвала функцию `f2`, та в свою очередь — функцию `f3` и т. д., а в функции, скажем, `f10` возможно возникновение ошибочной ситуации. Тогда нужно предусмотреть значение, возвращаемое `f10` в случае этой ошибки, в функции `f9` сделать проверку и в случае ошибки тоже возвратить специальное значение — и так во всех функциях.

Теоретически именно так всё и должно делаться, все возможные ошибочные ситуации должны проверяться и т. д., но на практике аккуратное соблюдение требований по обработке ошибок может оказаться настолько трудоёмким, что программисты прибегают к известному «алгоритму решения всех проблем», а именно — (1) поднять вверх правую руку и (2) резко махнуть ею, одновременно произнося что-то вроде «а, ладно». Есть даже шуточная фраза на эту тему: «никогда не проверяйте на ошибки, которые вы не знаете, как обработать». Разумеется, такое обычно даром не проходит. Ошибка, которую не стали обрабатывать в надежде, что она никогда не произойдёт, проявится в самый неподходящий момент, причём чем дальше находится заказчик и чем больше денег он заплатил, тем выше вероятность, что наша программа сломается именно у него.

Пытаясь снять остроту проблемы, создатели языков программирования предусматривают специальные возможности для обработки ошибочных ситуаций. В языке Си++ соответствующий механизм называется ***обработкой исключений*** (англ. *exception handling*).

10.5.2. Общая идея механизма исключений

Отметим ещё один недостаток использования специальных возвращаемых значений для индикации ошибки. В некоторых случаях среди всех значений возвращаемого функцией типа просто нет ни одного неиспользуемого; отличный пример этого — функция `atoi`, переводящая строковое представление целого числа в значение типа `int`. Возвращает она, естественно, число типа `int`, а поскольку любое такое число имеет строковое представление, то и вернуть (при отсутствии каких-либо ошибок) она может, вообще говоря, любое значение типа `int`. В то же время возможно, что в строке, поданной `atoi` на вход, содержится текст, не являющийся представлением какого-либо числа (например, состоящий из букв, а не из цифр). Разработчики функции решили в такой ситуации за неимением лучшего возвращать ноль, но это не решает проблему, ведь получается, что функция возвращает одно и то же и для строки, содержащей белиберду, и для строки `"0"`, которая вполне корректна. Это приводит нас к идее ***исключения***, которое представляет собой *особый способ завершения функции*: если в языках, не поддерживающих механизм исключений, функция могла только вернуть одно из возможных значений, то в языках, поддерживающих исключения, функция может либо вернуть значение, либо *возбудить исключение*.

Напомним (мы уже упоминали это на стр. 34), что исключения вовсе не являются составной частью парадигмы объектно-ориентированного программирования, как это почему-то часто утверждается. Напротив, они скорее относятся к *функциональному программированию*; позднее при обсуждении языка Scheme мы рассмотрим функцию `call/cc` и механизм *континуаций*, который представляет собой *обобщение* механизма исключений; в Common Lisp, а также в языках Пролог и Tcl, с которыми мы тоже вскоре познакомимся, континуации не поддерживаются, но при этом представлены механизмы, по смыслу аналогичные средствам обработки исключений Си++. Больше того, в Scheme исключения тоже есть — безотносительно континуаций.

При возбуждении исключения программа переходит в особое состояние, в котором все активные на текущий момент функции досрочно завершаются одна за другой, пока не найдётся такая, которая может справиться с данным типом исключительных ситуаций. Чтобы понять, о чём идёт речь, вспомним пример, приведённый в предыдущем параграфе: функция `f1` вызвала функцию `f2`, `f2` вызвала функцию `f3` и т. д. вплоть до `f10`, где и возникает ошибка. При этом нам совершенно не обязательно делать какие-либо проверки в функциях `f9`, `f8` и т. д., вполне достаточно пометить, например, функцию `f1` как умеющую справляться с данным типом ошибок. Тогда при возникновении ошибки во время исполнения `f10` будет завершена досрочно и она сама, и вызвавшая её функция `f9`, и `f8`, и так вплоть до `f1`, которая и обработает ошибку. Иначе говоря, если некая функция `g` вызывает другую функцию `h`, а эта последняя возбуждает исключение, для которого в `g` нет обработчика, то это эквивалентно тому, как если бы функция `g` сама возбудила то же самое исключение.

Иногда всю обработку ошибок делают в функции `main`. Она для этого удобна, поскольку остаётся активной всё время исполнения программы[17] и, как следствие, может обработать исключение, возбуждённое практически в любой части программы.

10.5.3. Возбуждение исключений

При возникновении ситуации, трактуемой как ошибочная и требующая обработки в качестве исключительной, следует *возбудить исключение* с помощью оператора `throw` (англ. *бросить*). У этого оператора имеется один параметр, в качестве которого может выступать выражение почти что произвольного[18] типа, в том числе как любого базового (`int`, `float`, ...), так и производного (указатель и т. п.), а равно и определённого пользователем, включая, что немаловажно, классы и структуры. Тип выражения в операторе `throw` будем называть ***типом исключения***, а значение выражения — ***значением исключения***.

Для примера рассмотрим функцию, которая принимает на вход имя (текстового) файла, а возвращает количество строк (т. е. символов перевода строки) в этом файле. Ситуация, когда файл не удалось открыть, для такой функции оказывается заведомо исключительной; в этом случае в языке Си нам пришлось бы возвращать специальное значение (например, `-1`), а при вызове проверять, не его ли вернула функция. В Си++ это можно сделать проще:

```
unsigned int line_count_in_file(const char *file_name)
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw "couldn't open the file";
    int n = 0;
```

[17] Строго говоря, это не совсем так, поскольку есть ещё конструкторы и деструкторы глобальных объектов, которые отрабатывают до и после функции `main`, но в некотором приближении можно этим пренебречь.

[18] В роли объекта исключения не могут выступать типы, для которых недоступны конструктор копирования и/или деструктор; почему это так, станет понятно позднее.

```
    int c = 0;
    while((c = fgetc(f)) != EOF)
        if(c == '\n')
            n++;
    fclose(f);
    return n;
}
```

В этом примере мы в качестве исключения бросили строку (точнее, адрес её начала, т. е. значение типа `const char*`). Мы могли бы бросить выражение другого типа; например, следующий оператор бросает исключение типа `int`:

```
throw 27;
```

Чаще, однако, в роли исключений выступают объекты специально введённых для этой цели классов; разговор об этом впереди.

10.5.4. Обработка исключений

Обработку исключительных ситуаций следует предусматривать, естественно, только в тех местах, в которых мы можем с такой ситуацией справиться. К примеру, если дальнейшая работа после обнаружения ошибки требует обращения к пользователю, то обработку этой ошибки следует вставить в одну из функций, наделённых правом вести диалог с пользователем. Как уже говорилось, часто обработчик помещают в функцию `main`, но, конечно, не всегда.

Общий принцип организации обработки исключений таков. Из программы выделяется некоторый блок кода, исполнение которого *предположительно* может привести к возникновению исключительных ситуаций. Этот блок заключается в специальное синтаксическое обрамление и снабжается одной или несколькими инструкциями, как следует действовать при возникновении исключения того или иного типа. При этом такие инструкции действуют в отношении исключений соответствующих типов, **возникших в программе с момента входа в помеченный блок и до момента выхода из него**, в том числе в функциях, вызванных из блока прямо или косвенно. Глубина вызовов в данном случае роли не играет.

Для пометки блока с целью обработки исключительных ситуаций в Си++ используется ключевое слово `try` (англ. *попытаться*), за которым следует собственно блок, то есть последовательность операторов, заключённая в фигурные скобки. Сразу после `try`-блока помещают один или больше ***обработчиков исключений***. Обработчик исключений синтаксически несколько напоминает функцию с одним параметром, хотя это только внешнее сходство; на самом деле обработчик начинается с ключевого слова `catch` (англ. *поймать*), сразу после него ставятся круглые скобки, внутри которых — описание формального параметра (точно так, как это делается при описании функции с одним параметром). Тип параметра задаёт тип обрабатываемого исключения, а по имени параметра можно получить доступ к значению исключения.

Для примера припомним функцию из предыдущего параграфа, подсчитывающую количество строк в файле, и на основе этой функции напишем программу целиком. Будем считать, что функция `line_count_in_file` находится в отдельном модуле, так что в программе нам потребуется только её прототип.

```
#include <stdio.h>

unsigned int line_count_in_file(const char *file_name);

int main(int argc, char **argv)
{
    if(argc < 2) {
        fprintf(stderr, "No file name\n");
        return 1;
    }
    try {
        int res = line_count_in_file(argv[1]);
        printf("The file %s contains %d lines\n",
               argv[1], res);
    }
    catch(const char *exception) {
        fprintf(stderr, "Exception (string): %s\n",
                exception);
        return 2;
    }
    return 0;
}
```

Эта программа, если всё будет в порядке, напечатает имя файла и количество строк в нём; если же открыть файл не удастся, функция `line_count_in_file` переведёт программу в состояние исключительной ситуации, а соответствующий обработчик будет найден в функции `main`. Ни остаток функции `line_count_in_file`, ни остаток `try`-блока в этом случае выполняться не будут, управление окажется сразу передано в обработчик (`catch`-блок).

В этом примере исключение может возникнуть в функции, непосредственно вызванной из `try`-блока, но на самом деле это не важно: с таким же успехом оно могло возникнуть в функции, вызванной из `line_count_in_file` прямо или косвенно на любую глубину.

Обратите внимание, что мы в данном случае ловим исключение `const char*`, а не просто `char*`, поскольку именно такой тип (адрес константы типа `char`) имеет в Си и Си++ строковый литерал — строка, заключённая в двойные кавычки. Если убрать модификатор `const`, исключение поймано не будет.

В рассмотренном примере мы предусмотрели всего один обработчик исключения, хотя язык Си++ допускает произвольное их количество (не менее одного). Если бы из нашего `try`-блока вызывалось больше функций и потенциально они могли бы привести к исключительным ситуациям других типов, можно было бы написать что-то вроде следующего:

```
try {
    // ...
}
catch(const char *x) {
    // ...
}
catch(int x) {
    // ...
}
```

Здесь важно понимать несколько простых правил. Во-первых, **обработчики (`catch`-блоки) рассматриваются по порядку, один за другим, причём сработать может только один из них или ни одного**. Это значит, в частности, что писать два обработчика одного типа или типов, сводимых один к другому, нет никакого смысла.

Правила преобразования типов при обработке исключений несколько отличаются от правил, применяемых при вызове функций; мы рассмотрим эти правила в одном из следующих параграфов.

Во-вторых, **обработчик может поймать только исключение, возникшее во время работы соответствующего `try`-блока**. В частности, обработчик не может поймать исключение, которое выбросил один из предшествующих обработчиков, относящихся к тому же `try`-блоку.

Наконец, **если исключение не поймано ни одним из обработчиков, оно выбрасывается дальше, как если бы фрагмент кода, в котором исключение возникло, вовсе не был обрамлён никаким `try`-блоком**. На пути одного исключения может оказаться несколько `try`-блоков, как бы вложенных друг в друга (во всяком случае, в смысле временны́х периодов исполнения), и в некоторых из них может не оказаться подходящего обработчика. В таком случае досрочное завершение активных функций и уничтожение стековых фреймов продолжается дальше, пока не будет найден `try`-блок с подходящим обработчиком.

В обработчиках исключений можно использовать оператор `throw` без параметров. В такой форме он бросает то исключение, которое только что было поймано. Так делают в случаях, когда полностью справиться с

возникшей ситуацией в данном месте программы невозможно, но прежде чем бросить исключение дальше, нужно произвести ещё какие-то действия. Примеры таких ситуаций рассматриваются в следующем параграфе.

10.5.5. Обработчики с многоточием

В некоторых случаях оказывается осмысленным обработчик, способный поймать *произвольное исключение независимо от его типа*. Такой обработчик записывается так же, как и обычный, только вместо типа исключения и имени параметра в скобках указывается многоточие. Конечно, значение исключения в этом случае использовано быть не может, но в некоторых случаях это и не важно. Допустим, мы подключаем к нашей программе модули, написанные другими программистами, и не уверены, что знаем все типы исключений, генерируемые этими модулями. В таком случае правильно было бы расположить в функции `main` примерно такой `try`-блок:

```
int main()
{
    try {
        // здесь пишем весь код главной функции
        return 0;
    }
    catch(const char *x) {
        fprintf(stderr, "Exception (string): %s\n", x);
    }
    catch(int x) {
        fprintf(stderr, "Exception (int): %d\n", x);
    }
    catch(...) {
        fprintf(stderr, "Something strange caught\n");
    }
    return 1;
}
```

Обработчики с многоточием делают особенно актуальной форму `throw` без параметров; её использование позволяет поймать исключение произвольного типа, выполнить какие-то действия, после чего бросить исключение дальше. Это может оказаться полезным, если в нашем коде локально захватываются те или иные ресурсы (выделяется память, открываются файлы и т. п.) и их следует освободить, прежде чем функция окажется завершена. Рассмотрим для примера функцию, в которой выделяется динамический массив целых чисел:

```
void f(int n)
{
    int *p = new int[n];
    // весь остальной код функции
    delete[] p;
}
```

Если во время выполнения кода, находящегося между `new` и `delete`, возникнет исключение, то `delete` не будет выполнен и массив, на который указывал `p`, будет продолжать занимать память, то есть станет мусором. Проблема снимается, если весь этот код заключить в `try`-блок, после которого есть обработчик для исключений произвольного типа, который, прежде чем бросить исключение дальше, освобождает память:

```
void f(int n)
{
    int *p = new int[n];
    try {
        // весь остальной код функции
    }
    catch(...) {
        delete[] p;
        throw;
    }
    delete[] p;
}
```

10.5.6. Объект класса в роли исключения

Обычно при обнаружении исключительной ситуации желательно передать обработчику максимум информации о возникших трудностях. Встроенные типы данных плохо пригодны для этого. Действительно, в рассмотренном выше примере было бы неплохо передать обработчику не только текстовое сообщение `couldn't open the file`, но и имя файла, и значение переменной `errno` сразу после выполнения `fopen`, которое может помочь в анализе проблемы. Мы, однако, ограничились в примере одной строковой константой, чтобы не формировать строку, содержащую всю перечисленную информацию — ведь это потребовало бы значительного увеличения кода и создало бы определённые трудности с освобождением памяти от такой строки, поскольку саму строку пришлось бы создавать динамически.

Кроме того, есть и ещё одна проблема. Программа может использовать несколько библиотек, причём изготовленных разными производителями; также программа может быть разделена на несколько подсистем, создаваемых более или менее независимо. Весьма желательно иметь некий универсальный способ разделения исключений по признаку их возникновения в той или иной подсистеме программы или библиотеке. Встроенные типы для этого не подходят совсем, поскольку идея использования тех же текстовых строк в качестве исключений может одновременно прийти в голову авторам сразу всех используемых нами библиотек и половины наших подсистем.

Именно поэтому чаще всего в качестве типа исключения выступает специально для этой цели описанный класс. С одной стороны, в объект класса можно поместить всю информацию, какая нам только может показаться полезной для обработчика; действительно, никто ведь не мешает описать в классе столько полей, сколько нам захочется. С другой стороны, каждая подсистема и каждая библиотека в этой ситуации, скорее всего, введёт свои собственные классы для представления исключений. В головной программе мы сможем обрабатывать такие классы отдельными обработчиками, что позволит разделить обработку исключений, сгенерированных в разных подсистемах программы.

Отметим один очень важный момент. Как правило, выбрасываемый в качестве исключения объект создаётся локально. Локальные объекты, естественно, исчезают вместе со стековым фреймом создавшей их функции; как следствие, тот объект, который поймается в обработчике исключения, заведомо не может быть тем же экземпляром объекта, который фигурировал в операторе `throw`, и является, как нетрудно догадаться, его копией. Поэтому **практически всегда в классе, используемом для исключений, описывают конструктор копирования**.

Для примера опишем класс, объект которого было бы удобно использовать в качестве исключения в нашей функции `line_count_in_file`. Объект класса будет хранить имя файла, текстовый комментарий (этот комментарий поможет понять, в каком месте программы произошла ошибка) и будет запоминать значение глобальной переменной `errno` на момент создания объекта. Для копирования строк мы опишем в приватной части класса вспомогательную функцию `strdup`, которая будет полностью аналогична одноимённой функции из библиотеки чистого Си, но память будет выделять с помощью `new[]`, а не `malloc`. Поскольку доступ к объекту этой функции не нужен, опишем её с квалификатором `static`.

```
class FileException {
    int err_code;
    char *filename;
    char *comment;
public:
    FileException(const char *fn, const char *cmt);
    FileException(const FileException& other);
    ~FileException();
    const char *GetName() const { return filename; }
    const char *GetComment() const { return comment; }
    int GetErrno() const { return err_code; }
```

```
private:
    static char *strdup(const char *str);
};
```

Описания конструкторов, деструктора и функции `strdup` вынесем за пределы заголовка класса:

```
FileException::FileException(const char *fn, const char *cmt)
{
    err_code = errno;
    filename = strdup(fn);
    comment = strdup(cmt);
}
FileException::FileException(const FileException& other)
{
    err_code = other.err_code;
    filename = strdup(other.filename);
    comment = strdup(other.comment);
}
FileException::~FileException()
{
    delete[] filename;
    delete[] comment;
}
char *FileException::strdup(const char *str)
{
    char *res = new char[strlen(str)+1];
    strcpy(res, str);
    return res;
}
```

Теперь мы можем изменить начало функции `line_count_in_file` (см. стр. 61) на следующее:

```
unsigned int line_count_in_file(const char *file_name)
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw FileException(file_name,
            "couldn't open for line counting");
  // ...
```

Обработчик (`catch`-блок) для такого исключения может быть, например, таким:

```
    catch(const FileException &ex) {
        fprintf(stderr, "File exception: %s (%s): %s\n",
                ex.GetName(), ex.GetComment(),
                strerror(ex.GetErrno()));
        return 1;
    }
```

10.5.7. Автоматическая очистка

В примере, рассмотренном на стр. 63, мы были вынуждены перехватывать и потом заново выбрасывать исключения произвольного вида, чтобы обеспечить корректное удаление локально выделенной динамической памяти. К счастью, так приходится действовать не всегда. Работа с исключениями изрядно облегчается тем, что **компилятор гарантирует вызов деструкторов для всех локальных объектов, у которых деструкторы есть, прежде чем функция завершится — по исключению ли или обычным путём**.

Пусть, например, имеется класс `A`, снабжённый деструктором (непустым, иначе не интересно), и описана функция

```
void f()
{
    A a;
    // ...
    g();
    // ...
}
```

В момент завершения работы функции `f` для объекта `a` будет вызван деструктор, причём произойдёт это вне зависимости от того, как именно `f` завершится — в том числе и если функция `g` выбросит исключение и именно оно станет причиной завершения `f`. Это свойство Си++ называется ***автоматической очисткой***.

10.5.8. Преобразования типов исключений

Тип исключения, указанный в обработчике (`catch`-блоке), не всегда точно совпадает с типом выражения, использованного в операторе `throw`, однако правила преобразования типов в данном случае отличаются от правил, действующих при вызове функций. Одно из основных отличий тут в том, что целочисленные типы при обработке исключений не преобразуются друг к другу, то есть, например, обработчик с указанным типом исключения `int` не поймает исключение типа `char` или `long`.

Допустимые преобразования проще будет явно перечислить. Во-первых, любой тип может быть преобразован к ссылке на этот тип, то есть если оператор `throw` использует выражение типа `A`, то такое исключение может быть обработано `catch`-блоком вида `catch(A &ref)`. Во-вторых, неконстантные указатели и ссылки могут быть преобразованы к константным, то есть, например, если мы бросим исключение типа `char*`, то оно может быть поймано не только как `char*`, но и как `const char*`. В обратную сторону преобразование запрещено, так что обработчик типа `char*` (без модификатора `const`) не может поймать выражение типа `const char*` и, в частности, не может обработать исключение, выброшенное с использованием строкового литерала, например такое:

```
throw "I can't work";
```

Последний (и, возможно, самый важный) вид преобразований имеет отношение к наследованию и полиморфизму. К обсуждению этой темы мы вернёмся после рассказа о наследовании.

10.6. Наследование и полиморфизм

Большая часть возможностей Си++, рассмотренных нами выше, не имеет, как это неоднократно подчёркивалось, прямого отношения к объектно-ориентированному программированию — кроме разве что методов, вызов которых рассматривается как «техническая реализация» передачи сообщения объекту; но, как мы отметили ещё в предыдущей части (см. стр. 16), если для поддержки объектов ограничиться только методами и инкапсуляцией, получится не вполне полноценный вариант ООП, который часто даже называют иначе — *object-based programming* (а не *object-oriented*).

Настала пора познакомиться с фирменной особенностью ООП — ***наследованием***, а заодно и с теми вариантами полиморфизма, которые имеются в виду при перечислении «трёх китов ООП» (в отличие, опять же, от статического полиморфизма, с которым мы неоднократно сталкивались и который никакого отношения к ООП не имеет).

10.6.1. Иерархические предметные области

Очень часто, и особенно в более-менее крупных компьютерных программах, объекты предметной области можно естественным образом объединить в некие категории, причём объекты каждой категории могут подразделяться на подкатегории и т. д., образуя таким образом ***иерархию типов объектов***. Например, в программе, моделирующей дорожное движение, могут рассматриваться объекты категории «участник дорожного движения», причём эта категория подразделяется на подкатегории «пешеходы» и «транспортные средства»; транспортные средства, в свою очередь, делятся на «велосипеды», «гужевые повозки» и «механические транспортные средства», эти последние — на трамваи, троллейбусы и автомобили, автомобили — на грузовые и легковые и т. д.

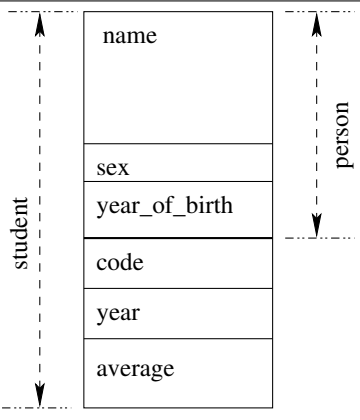

Рис. 10.3. Унаследованная структура данных

В такой иерархии каждая категория объектов, в том числе и такая, которая сама подразделяется на категории, обладает некоторыми свойствами, специфичными для данной категории, причём этими же свойствами обладают и все объекты её подкатегорий. Так, все автомобили имеют, по-видимому, характеристики «объём двигателя», «тип топлива», «объём топливного бака» и «количество топлива в баке», у них может заканчиваться топливо, над ними должна быть определена операция «заправка топлива». Все эти характеристики не имеют никакого смысла для объектов, не входящих в категорию «автомобили» — ни для трамваев, ни для велосипедов, ни тем более для пешеходов. С другой стороны, у грузовых автомобилей есть характеристика «объём кузова» и операции «погрузка» и «разгрузка», которых нет у автомобилей, не являющихся грузовыми. Наконец, такие характеристики, как текущая скорость и направление движения, очевидно, присущи каждому объекту рассматриваемой иерархии, вплоть до пешеходов.

Как видим, у нас возникают структуры данных, имеющие, с одной стороны, различный набор полей и обрабатывающих функций, но, с другой стороны, имеющие и некоторые общие поля, и определённые общие операции. Существуют весьма различные подходы к решению возникшего противоречия. При работе на языке Си чаще всего выходят из положения описанием структур данных, у которых первые несколько полей совпадают; при этом приходится то и дело изменять типы указателей, а это, как мы знаем, ведёт к труднообнаружимым ошибкам. Объектно-ориентированные языки программирования предлагают более корректный подход, называемый обычно ***наследованием***.

Объяснить наследование как технический приём не так просто; мы попытаемся применить не вполне традиционный подход, начав с рассмотрения объекта как простой структуры данных (каковой он на самом деле и является, просто мы об этом не всегда помним). Методы мы подключим к делу потом.

10.6.2. Наследование структур и полиморфизм адресов

Идеологически **наследование рассматривается как переход от общего к частному**, а в частных случаях с объектом обычно можно связать *больше* информации, нежели в случае общем.

В терминах структур данных, расположенных в памяти, наследование представляет собой добавление новых полей данных к ранее описанной структуре с целью *уточнения* сведений об описываемом объекте. Пусть, например, у нас имеется структура данных, описывающая персону (человека) и содержащая поля для имени, пола и года рождения:

```
struct person {
    char name[64];
    char sex; // 'm' or 'f'
    int year_of_birth;
};
```

Пусть теперь нам понадобилась структура данных, описывающая *студента* (частный случай персоны). Относительно студента нас может интересовать код специальности, номер курса и показатель успеваемости (средний балл). Вместе с тем, несомненно, студент — это тоже человек, так что ему присущи и свойства, общие для всех персон: имя, пол и год рождения; иначе говоря, мы можем перейти от общего (персона) к частному (студент), *добавив* уточняющие сведения, и сделать это можно с помощью механизма наследования. На Си++ это записывается так:

```
struct student : person {
    int code;
    int year;               // курс
    float average;
};
```

Если теперь описать переменную типа `student`, она будет иметь все свойства структуры `person` плюс дополнительные поля:

```
    student s1;
    strcpy(s1.name, "John Doe");
    s1.sex = 'm';
    s1.year_of_birth = 1989;
    s1.code = 51311;
    s1.year = 2;
    s1.average = 4.75;
```

Структура `person` в данном случае называется ***базовой*** или ***родительской***, а структура `student` — ***унаследованной*** или ***порождённой***, иногда ***дочерней***. Также употребляются термины «предок» и «потомок» (в данном случае соответственно для `person` и `student`).

Расположение полей переменной `s1` в памяти показано на рис. 10.3. Стоит специально отметить, что поля, относящиеся к части структуры `student`, унаследованной от `person`, располагаются на тех же местах (по тем же смещениям относительно начала), где они располагались бы в структуре `person`. Получается, что мы можем при необходимости работать с переменной `s1` точно так, как если бы она была переменной типа `person`, а не типа `student`. Важно понимать, что обратное неверно: в структуре `person` отсутствуют поля, которые ожидались бы от структуры `student`. В связи с этим **Си++ разрешает неявное преобразование адреса переменной порождённого типа в адрес объекта родительского типа**, в нашем примере `student*` в `person*`. Соответствующие преобразования разрешены как для указателей, так и для ссылок. Например, следующие строки полностью корректны:

```
    student s1;
    person *p;
    p = &s1;
    person &ref = s1;
```

Если в программе описана функция, принимающая параметром указатель или ссылку на объект типа `person`, ей без каких-либо сложностей можно передать соответственно указатель или ссылку на объект типа `student`:

```
void f(person &pers) { ... }
    // ...
    student s1;
    // ...
    f(s1); // корректно!
```

Это свойство предков, потомков и их адресов называется ***полиморфизмом адресов*** (реже — *полиморфизмом ссылок*); в отличие от рассмотренных ранее проявлений полиморфизма, этот новый его вид имеет к объектно-ориентированному программированию самое прямое отношение. Разрешённое в объектно-ориентированных языках неявное преобразование адресов (указателей) и ссылок от типа-потомка к типу-предку мы будем называть ***преобразованием по закону полиморфизма***.

Мы подразумеваем, что полиморфизм, какой бы его вид мы ни рассматривали, состоит в том, что операция или действие, обозначаемые одним и тем же символом, корректно выполняются для операндов или параметров различных типов. Часто встречается немного иное понимание полиморфизма: что одно и то же обозначение используется для *разных* действий, выбираемых в зависимости от типов параметров. Если полиморфизм понимать так, то наш «полиморфизм адресов» перестанет быть, собственно говоря, полиморфизмом, ведь здесь, напротив, действия (технически) выполняются совершенно одинаковые вне зависимости от типа; при этом рассмотренные ранее виды полиморфизма такому пониманию вполне соответствуют. Так или иначе, применяемая терминология может различаться от источника к источнику.

10.6.3. Методы и защита при наследовании

Поскольку наследование, несомненно, является составной частью объектно-ориентированного программирования, чаще всего оно применяется для типов, имеющих функции-члены (методы), и в роли таких типов классы выступают намного чаще, чем структуры. Технически это никоим образом не требование, ведь структуры от классов, как мы уже знаем (см. §10.3.5), отличаются только моделью защиты по умолчанию — и более ничем. Естественно, наследование возможно как для структур, так и для классов, причём классы можно наследовать от структур и наоборот; просто, как мы отмечали ранее, для описания объектов в смысле ООП программисты стараются использовать именно классы, а не структуры, чтобы показать, что здесь имеется в виду именно эта парадигма. В §9.1.3 мы отмечали, что само слово *класс* изначально относится к терминологии ООП — это множество объектов, устроенных одинаково, а с прагматической точки зрения — описание устройства объекта; в такой трактовке, даже если для создания объекта (в смысле ООП) мы воспользуемся структурой, полученный тип следует считать именно классом — если не в терминах языка Си++, то в терминах ООП. В дальнейшем изложении мы будем использовать слово «класс» для обозначения всех типов объектов, если таковые относятся к парадигме ООП, не уточняя, что это (в терминах Си++) может быть как класс, так и структура.

Считается, что построение класса-потомка на основе базового класса *может быть* (хотя и не обязано, и в большинстве случаев не является) *частной деталью реализации*, о которой окружающий мир знать не должен. В связи с этим в Си++ различают наследование открытое (`public`) и закрытое (`private`). Модель защиты для конкретного случая наследования указывается в заголовке класса непосредственно перед названием класса-предка, например:

```
class B : public A {
   /*...*/
};
```

или

```
class C : private A {
   /*...*/
};
```

Если в первом случае все свойства класса `A` (его открытые методы и поля, если таковые в нём есть) будут доступны для объектов класса `B` отовсюду, то во втором случае — только из методов класса `C`, а вся остальная программа вообще не будет знать, что класс `C` унаследован от `A`. Всё это работает и для структур.

Модель защиты для наследования можно не указывать, как мы это делали в примере на стр. 65. В этом случае будут действовать умолчания: для структуры наследование по умолчанию открытое, для класса — закрытое. **В реальной жизни потребность в закрытом наследовании возникает крайне редко**, поэтому при описании наследуемых классов обычно указывают `public`, а при описании наследуемых структур не указывают ничего, полагаясь на умолчания.

Заметим теперь, что для базового класса порождённый класс ничуть не «лучше» всей остальной программы и, как и любой недружественный фрагмент кода, не должен иметь доступа к деталям реализации класса. Поэтому, очевидно, закрытые поля и методы базового класса не будут доступны порождённым классам.

В реальных задачах часто возникает потребность в таких деталях интерфейса класса, которые предназначены только и исключительно для его потомков. Для таких случаев наряду с режимами `public` и `private` вводится ещё и третий режим защиты, `protected` (*защищённый*). Поля и методы, помеченные словом `protected`, будут доступны в самом классе (то есть в его методах), в дружественных функциях, а также в методах потомков данного класса[19]; во всей остальной программе доступ к ним будет запрещён.

Важно понимать, что поля и методы, имеющие режим защиты `protected`, не могут считаться в полном смысле слова деталями реализации, которые не касаются никого, кроме данного класса. Разница здесь достаточно принципиальна. Детали, помещённые в закрытую часть класса, можно безболезненно изменять, исправляя при этом только методы самого класса и дружественные функции, которые опять-таки перечислены в явном виде в заголовке класса. Иначе говоря, переделывая приватную часть класса, мы всегда знаем, где проходит граница кода, который может при этом «сломаться» и который нам, возможно, придётся исправлять. Напротив, особенности реализации класса, помеченные как `protected`, могут использоваться в самых неожиданных частях программы: достаточно кому-то где-то описать ещё одного потомка нашего класса, о котором мы не подозреваем. Поэтому, в отличие от приватных полей и методов, поля и методы с режимом `protected` должны обязательно документироваться, а к их изменению следует подходить столь же осторожно, как и к изменению открытой (публичной) части класса. Можно сказать, что `protected` — это хоть и особый, но всё же вид *открытых* особенностей класса.

Если мы работаем с объектом-потомком, в его классе-предке введён некий метод и нам этот метод доступен, то есть не скрыт от нас защитой, мы можем вызывать такой метод для объекта-потомка точно так же, как и для объекта-предка, причём, если не предпринять специальных мер, методы предка не будут ничего знать о том, что их вызывают для объекта-потомка, то есть будут работать так же, как и для объектов базового класса. Например, если мы сделаем класс «автомобиль» и предусмотрим для него операцию «заправка бензином», а потом породим от него класс «грузовик» с дополнительными свойствами, то операция «заправка бензином» будет доступна и для грузовика, причём реализация этой операции не будет ничего знать про особенности грузовиков и будет работать с объектом типа «грузовик» точно так же, как и с объектом типа «автомобиль».

Здесь работает описанный в предыдущем параграфе адресный полиморфизм в применении к параметру `this` (см. §10.3.2). В самом деле, пусть класс `B` унаследован от класса `A`, в котором есть функция (метод) `f`. Тогда указатель `this` в методе `f` имеет тип `A*` (или `const A*`, если метод константный). При вызове `f` для объекта класса `B` указатель `this` должен иметь тип `B*`, но, как мы знаем, `B*` разрешено неявно преобразовывать в `A*` по закону полиморфизма.

В терминах ООП это означает, что потомок умеет отвечать на все виды сообщений, предусмотренные для его предков; чтобы объекту можно было отправить такое сообщение (т. е. вызвать метод предка), достаточно иметь доступ к этому методу, то есть чтобы он не был скрыт защитой. Иначе говоря, с объектом-наследником можно при желании работать точно так же, как с объектом-предком, вообще не обращая внимания на тот факт, что мы имеем дело с объектом другого типа. Часто можно даже встретить утверждение, что **объект класса-потомка является также и объектом класса-предка**; учитывая, что на идеологическом уровне, как мы уже отмечали, наследование есть переход от общего к частному, объект-потомок оказывается *частным случаем* класса-предка. В самом деле, разве грузовик не является частным случаем автомобиля?

С другой стороны, как видно из рис. 10.3, объект порождённого класса *содержит* в себе объект базового класса в качестве своей части. Никакого противоречия тут нет, это просто две разные модели восприятия (можно даже сказать, что это разные парадигмы) или, если угодно, разные точки зрения: одна возникает при реализации средств языка (реализаторская семантика), вторая вытекает из логики проектирования программы (пользовательская семантика). К этому вопросу мы ещё вернёмся.

10.6.4. Конструирование и деструкция наследника

Конструкторы и деструкторы заслуживают особого упоминания при появлении наследования в программе. С какой бы точки зрения мы ни рассматривали порождённый объект, очевидно, что в момент его

[19] Для непосредственных потомков это верно всегда; если же речь идёт о *потомке потомка* или о ещё более дальнем «родстве», может оказаться так, что в цепочке наследников встретилось наследование закрытое, и тогда потомок может вообще не знать, что его предок построен на основе другого класса; естественно, такой потомок не будет иметь доступа ни к каким членам «засекреченного» дальнего предка, даже к публичным — и к защищённым тоже.

создания также создаётся и объект базового класса, причём неважно, считаем мы его *частью* порождённого объекта или же его *другой ипостасью*. Таким же точно образом при уничтожении порождённого объекта исчезает и базовый объект. Следовательно, при создании объекта порождённого класса должен отработать и конструктор базового класса, а при уничтожении такого объекта — деструктор базового класса. При этом, очевидно, в телах конструктора и деструктора порождённого класса должны быть доступны все части объекта, для которого они вызываются. Базовый объект, таким образом, должен быть инициализирован *раньше*, а уничтожен — *позже* объекта порождённого. Получается, что время существования объекта класса-наследника в его «базовой» ипостаси как бы чуть-чуть больше, чем время существования его же в качестве объекта своего собственного типа.

С деструктором дела обстоят достаточно просто: компилятор автоматически вставляет в код деструктора порождённого класса (в самый его конец) вызов деструктора базового класса, так что сначала отработает тело деструктора порождённого класса, а затем — тело деструктора базового класса. С конструктором всё тоже могло бы быть просто (сначала вызвать тело конструктора базового класса, потом тело конструктора класса порождённого), если бы не то обстоятельство, что конструкторы (в общем случае) могут требовать входных параметров. С похожей проблемой мы уже сталкивались при рассмотрении классов, имеющих поля типа класс (см. §10.4.15). Решается проблема в данном случае совершенно аналогично: в описании конструктора порождённого класса между заголовком и телом вставляется список инициализаторов, начинающийся с инициализатора базового класса (обозначаемого в данном случае именем класса) с указанием всех нужных параметров конструктора. Так, если `A` — базовый класс, конструктору которого требуются два параметра типа `int`, `B` — класс, унаследованный от `A`, снабжённый конструктором по умолчанию и имеющий поле `int i`:

```
class A {
    // ...
public:
    A(int p, int q) { /* ... */ }
    // ...
};

class B : public A {
    int i;
public:
    B();
    // ...
};
```

— то описание его конструктора может выглядеть так:

```
B::B() : A(2, 3), i(4)
{
    /* ... */
}
```

где 2 и 3 — параметры для конструктора базового класса, 4 — начальное значение поля `i`.

10.6.5. Виртуальные функции

До сих пор мы предполагали, что объект-потомок будет реагировать на сообщения, определённые для его предка, *точно так же*, как на них реагировал бы объект-предок. В ряде случаев это не отвечает потребностям моделируемой предметной области: бывает так, что на те или иные сообщения потомку желательно реагировать иначе, каким-то своим собственным способом. Достичь этого позволяет механизм ***виртуальных функций***; с его помощью автор класса-предка может предоставить потомкам возможность модификации поведения отдельных методов, причём как полной, так и частичной. Чтобы понять, как это происходит и зачем всё это нужно, мы для начала рассмотрим пример, а затем поясним, как соответствующий механизм устроен.

Итак, допустим, что перед нами стоит задача, связанная с компьютерной графикой, и нужно описать сцену (то есть набор графических элементов) в виде некоего списка или массива объектов, задающих графические объекты различного типа, например, отдельные пиксели, линии, окружности и т. п. Для начала опишем класс, объекты которого будут представлять в нашей сцене простейшие графические примитивы — отдельные пиксели. Ясно, что у пикселя имеются две координаты (их обычно задают числами с плавающей точкой) и цвет (целое число). Будем считать, что основные действия с графическим объектом, в том числе и с пикселями — это показать объект на экране (мы обозначим это действие функцией `Show`), убрать его с экрана (`Hide`) и переместить его в новую позицию (`Move`).

```
class Pixel {
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    void Show();
    void Hide();
    void Move(double nx, double ny);
};
```

Конкретика описания тел функций `Show` и `Hide` зависит от используемой графической библиотеки, правил пересчёта координат и т. п.; мы на этом останавливаться не будем, просто предположим, что эти функции где-то описаны. Используя их, мы очень легко можем описать функцию `Move`, ведь перемещение состоит в том, что объект сначала убирают с экрана, потом меняют его координаты и наконец снова показывают:

```
void Pixel::Move(double nx, double ny)
{
    Hide();
    x = nx;
    y = ny;
    Show();
}
```

Предположим теперь, что нам нужен также объект «окружность». Понятно, что такой объект обладает теми же свойствами (координаты и цвет) и в дополнение к ним характеризуется ещё радиусом. Поэтому мы воспользуемся уже имеющимся классом `Pixel` в качестве базового[20], а класс `Circle` от него *унаследуем*. Конечно, методам класса `Circle` потребуется знать координаты и цвет, хотя бы для того, чтобы уметь рисовать заданную окружность на экране. В нашем примере мы решим вопрос доступа самым простым (хотя и не самым лучшим) способом: сделаем поля базового класса защищенными (`protected`), а не закрытыми; для этого вставим в самое начало описания класса `Pixel` директиву `protected`:

```
class Pixel {
protected:
    double x;
    // ...
```

Теперь мы можем перейти к описанию класса `Circle` и для начала опишем поле, которого нам не хватает, чтобы превратить точку в окружность:

```
class Circle : public Pixel {
    double radius;
public:
```

Опишем теперь конструктор для класса `Circle`. При этом нам придётся принять на один параметр больше, чем в конструкторе класса `Pixel`. Кроме того, поскольку единственный конструктор базового класса требует указания трёх параметров, нужно будет задать параметры для вызова конструктора базового класса, как это обсуждалось на стр. 67. Окончательно описание выйдет таким:

[20] Здесь мы несколько нарушаем принципы объектно-ориентированного проектирования, поскольку окружность, очевидно, не является частным случаем точки. Мы исправим эту оплошность в одном из следующих параграфов.

```
Circle(double x, double y, double rad, int color)
    : Pixel(x, y, color), radius(rad) {}
```

Разумеется, в объекте необходимо описать функции `Show` и `Hide`, предназначенные для рисования и стирания с экрана окружности, которые будут отличаться от функций, предназначенных для одиночного пикселя; мы, как и при рассмотрении класса `Pixel`, воздержимся от описания конкретных тел этих функций, ограничившись их заголовками:

```
void Show();
void Hide();
```

Нетерпеливый читатель может предположить, что сейчас мы будем описывать функцию `Move`, аналогичную той, что описана выше для класса `Pixel`. Вместо этого мы предложим немного подумать. Легко заметить, что *функция `Move` для нового класса окажется абсолютно такой же, как и для класса базового*, то есть нам потребуется написать не только точно такой же заголовок, но и точно такое же, с точностью до последней буквы, тело функции! Это и неудивительно, ведь вне всякой зависимости от формы графической фигуры её перемещение по экрану производится абсолютно одинаково, в три шага: стереть, изменить координаты, нарисовать в новом месте.

Возникает естественный вопрос — нельзя ли использовать для класса `Circle` ту же самую функцию `Move`, которая уже описана для класса `Pixel`, не создавая новой версии этой функции для `Circle`. Легко заметить, что (если не принять специальных мер) простой вызов функции `Move` для объекта класса `Circle`, конечно, возможен (ведь это же публичная функция базового класса, а наследование произведено по публичной схеме), но *не приведёт к нужным нам результатам*, и вот почему. Во время трансляции тела функции `Pixel::Move` компилятор может ничего не знать о существовании потомков у класса `Pixel`, которые к тому же вводят свои версии функций `Show` и `Hide`. Поэтому компилятор, естественно, вставляет в машинный код функции `Pixel::Move` обычные вызовы функций `Pixel::Show` и `Pixel::Hide`, то есть инструкции `CALL` с жёстко заданными исполнительными адресами, не подразумевающие никаких изменений.

Если теперь вызвать функцию `Move` для объекта класса `Circle`, она для стирания с экрана объекта вызовет функцию `Hide` в той её версии, в которой она описана для класса `Pixel`. Эта функция выполнит действия по стиранию с экрана объекта-точки, тогда как стереть на самом деле нужно окружность. То же самое произойдёт с функцией `Show`: вместо окружности в новой позиции на экране будет отрисована точка. Ясно, что это совсем не то, что нам нужно.

Тем не менее кажется странным и неправильным описывать для класса `Circle` функцию, слово в слово повторяющую другую, которая уже имеется в базовом классе. И здесь нам на помощь приходит механизм ***виртуальных функций***. Кратко говоря, виртуальная функция (или виртуальный метод) — это функция, описывающая такое действие над объектом, относительно которого предполагается, что аналогичное действие будет определено и для объектов классов-потомков, но, возможно, для потомков оно будет выполняться иначе, чем для базового класса. В терминах теории ООП можно сказать, что виртуальным методом задаётся реакция объекта класса на некоторый тип сообщений в случае, если:

- предполагается, что у данного класса будут классы-потомки;
- объекты классов-потомков будут способны получать сообщения того же типа;
- объекты некоторых или всех потомков будут реагировать на эти сообщения иначе, чем это делает объект класса-предка.

Язык Си++ позволяет объявить в качестве виртуальной любую функцию-метод, кроме конструкторов и статических методов. Для этого перед заголовком функции ставится ключевое слово `virtual`. В отличие от вызовов обычных функций, вызовы функций виртуальных обрабатываются компилятором в предположении, что тип объекта, для которого вызывается функция, может отличаться от базового класса, и что для этого объекта может потребоваться вызов совсем другой функции, нежели для базового класса. В частности, если в классе `Pixel` поставить слово `virtual` перед каждой из функций `Show` и `Hide`, то при компиляции тела функции `Move` компилятор будет знать, что объект, на который указывает параметр `this`, может быть как объектом самого класса `Pixel`, так и объектом какого-нибудь его класса-потомка, причём для такого объекта может потребоваться вызов *других версий* функций `Show` и `Hide`, введённых в соответствующем потомке. Код, сгенерированный компилятором в такой ситуации, будет несколько сложнее, чем для обычного вызова функции, но зато он будет вызывать именно функцию, соответствующую типу объекта.

Попробуем понять, как (технически) это достигается. Если в классе описана хотя бы одна виртуальная функция, то компилятор вставляет во все объекты этого класса невидимое поле, называемое ***указателем на таблицу виртуальных методов*** (англ. *virtual method table pointer*, *vmtp*). Для всего класса создаётся (в одном экземпляре) неизменяемая ***таблица виртуальных методов***, содержащая указатели на каждую из описанных в классе виртуальных функций. Когда компилятор встречает вызов виртуальной функции, он вставляет в объектный код программы инструкцию извлечь из объекта значение поля `vmtp`, затем обратиться по полученному адресу к таблице виртуальных методов и из неё взять адрес нужной функции, после чего обратиться к функции, используя адрес.

Объект класса-потомка содержит, как мы знаем, все поля, присущие классу-предку. Это касается и невидимого поля `vmtp`, причём объекты класса-потомка имеют в этом поле иное значение, нежели объекты класса-предка. Для каждого класса-потомка компилятор создаёт (опять-таки в единственном экземпляре) свою собственную таблицу виртуальных методов, содержащую адреса соответствующих версий виртуальных функций; именно адрес этой таблицы заносится в поле `vmtp`. За инициализацию `vmtp` отвечает конструктор класса, в начало которого компилятор вставляет соответствующие команды.

Итак, объявим методы `Show` и `Hide` виртуальными в базовом классе:

```
class Pixel {
    // ...
    virtual void Show();
    virtual void Hide();
    // ...
};
```

Это позволит использовать функцию `Move` для любого из классов, унаследованных от `Pixel`, если только для этих классов описаны собственные (правильно работающие) функции `Show` и `Hide`.

Прежде чем завершить описание классов `Pixel` и `Circle`, отметим, что в классе, в котором имеется хотя бы одна виртуальная функция, рекомендуется всегда явно описывать деструктор и объявлять его виртуальным. Зачем это нужно, мы расскажем в §10.6.9. Сейчас просто сделаем это, чтобы компилятор не выдавал предупреждения. Окончательно заголовки наших классов примут следующий вид:

```
class Pixel {
protected:
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
    void Move(double nx, double ny);
};

class Circle : public Pixel {
    double radius;
public:
    Circle(double x, double y, double rad, int color)
        : Pixel(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Отметим, что в описании класса `Circle` слово `virtual` можно опустить; функции, совпадающие по профилю с виртуальными функциями базового класса, объявляются виртуальными автоматически.

Виртуальные функции можно вызывать не только из других методов того же класса, как в нашем примере; компилятор сгенерирует код для вызова через таблицу виртуальных методов при *любом* обращении к виртуальной функции, если тип объекта, для которого её вызывают, хотя бы теоретически может меняться[21]. Конечно, когда объект описан в виде обычной переменной типа класс и мы обращаемся к его методу, пусть и виртуальному, напрямую через имя этой переменной и точку, то есть в программе написано что-то вроде

```
A a;
// ...
a.func();   // всегда вызывается A::func; механизм
            // виртуальности не задействован
```

— то такое обращение, очевидно, бессмысленно реализовывать через виртуальность, ведь тип переменной известен во время компиляции и стать другим не может. Иное дело, если обращение к виртуальной функции записано через адрес объекта (ссылку или указатель). Это касается в том числе вызовов методов из тел других методов, где объект, как мы знаем, идентифицируется указателем `this` — но *не ограничивается* методами.

Так, если рассматривать классы из нашего примера, то где бы у нас в программе ни появился какой-нибудь указатель типа `Pixel*` или `Circle*`, вызовы через такой указатель функций `Show` и `Hide` компилятор оформит как виртуальные, предполагая, что указатель во время исполнения может содержать адрес не только объекта базового класса, но и какого-нибудь его потомка, в том числе такого, о котором компилятору на момент компиляции ничего не известно; то же самое касается ссылок типа `Pixel&` и `Circle&`. Скажем, если мы пишем какую-нибудь функцию, имеющую параметр `p` типа `Pixel*`, то, решив показать переданный нам объект на экране, мы можем выполнить

```
p->Show();
```

и не волноваться о том, адрес объекта какого типа нам передали — ведь вызван в любом случае будет метод нужного класса.

10.6.6. Чисто виртуальные методы и абстрактные классы

Как отмечалось в сноске на стр. 67, наследование класса «окружность» от класса «точка» нарушает принципы объектно-ориентированного проектирования, поскольку окружность не является частным случаем точки. Попробуем исправить ситуацию. Для этого заметим, что и точка, и окружность — частные случаи *геометрических фигур*, причём можно считать, что каждая геометрическая фигура обладает цветом и имеет координаты *точки привязки*. Для обычной точки в качестве точки привязки выступает она сама, для окружности точкой привязки будет её центр. Для других фигур точку привязки можно выбрать разными способами; так, для какого-нибудь прямоугольника это может быть либо центр пересечения диагоналей, либо одна из вершин, и т. п.

Такое представление об абстрактном понятии геометрической фигуры позволяет нам указать единый для всех фигур алгоритм передвижения фигуры по экрану: стереть фигуру с экрана, изменить координаты точки привязки, отрисовать фигуру в новом месте. С этим алгоритмом мы уже знакомы, он был реализован в функции `Move` на стр. 67.

Теперь ясно, как нужно изменить архитектуру нашей библиотеки классов, чтобы привести её в соответствие с основными принципами объектно-ориентированного программирования. Понятия «точка» и «окружность» не являются частными случаями друг друга, но оба они являются частным случаем понятия «геометрическая фигура». Поэтому, если мы опишем класс для представления абстрактной геометрической фигуры и от него унаследуем оба класса `Pixel` и `Circle`, такая архитектура будет полностью удовлетворять требованиям теории объектно-ориентированного проектирования.

Прежде чем приступать к описанию класса, представляющего геометрическую фигуру, отметим ещё один важный момент. Описывая в предыдущем параграфе классы для точек и окружностей, мы не писали конкретных тел для методов `Show` и `Hide`, но предполагали при этом, что в рабочей программе тела этих методов будут описаны (с учётом конкретной платформы разработки, используемой графической библиотеки и т. п.). Но для абстрактной геометрической фигуры тела методов `Show` и `Hide` описать *невозможно*; действительно, как можно нарисовать на экране фигуру, не зная, как она выглядит?!

Несмотря на это, мы точно знаем, как написать функцию `Move` в предположении, что для классов-потомков будут правильно описаны методы `Show` и `Hide`. Иначе говоря, мы знаем, что все объекты классов-потомков данного класса должны уметь получать сообщения определённого типа, но мы не можем при описании базового класса описать какую бы то ни было реакцию на эти сообщения, поскольку такая реакция полностью зависит от типа нашего потомка. При этом для описания некоторых других (более общих) методов базового класса нам требуется знание о том, что реакция на соответствующие сообщения будет предусмотрена во всех наших потомках.

Специально для таких случаев в Си++ введены так называемые ***чисто виртуальные функции*** (англ. *pure virtual functions*). Описывая в классе чисто виртуальную функцию, программист информирует компилятор, что функция с таким профилем будет существовать во всех классах-потомках, что под неё нужно зарезервировать позицию в таблице виртуальных функций, но при этом сама функция (её тело) для базового класса описываться не будет, так что значение адреса этой функции в таблице виртуальных функций следует оставить нулевым. Синтаксис описания чисто виртуальной функции таков:

```
class A {
    // ...
    virtual void f() = 0;
    // ...
 };
```

Видя на месте тела функции специальную лексическую последовательность «`= 0;`», компилятор воспринимает функцию как чисто виртуальную.

Такой синтаксис трудно назвать удачным, особенно если учесть, что никакое число, кроме нуля, использоваться здесь не может; тем не менее синтаксис в языке Си++ именно такой. Вопрос о причинах этого следовало бы задать лично Бьёрну Страуструпу, но, скорее всего, он на него не ответит.

Класс, в котором есть хотя бы одна чисто виртуальная функция, называется ***абстрактным классом***. Полезно будет запомнить, что **компилятор не позволяет создавать объекты абстрактных классов**. Единственное назначение абстрактного класса — служить базисом для порождения других классов, в которых все чисто виртуальные функции будут конкретизированы. Если в порождённом классе не описывается хотя бы одна из функций, объявленных в базовом классе как чисто виртуальные, компилятор считает, что функция осталась чисто виртуальной; такой класс-потомок считается абстрактным, как и его предок.

Подытожим наши рассуждения. Чтобы соответствовать принципам объектно-ориентированного программирования, нам следует описать класс, представляющий абстрактную геометрическую фигуру, обладающую цветом и координатами точки привязки, но не обладающую конкретной формой; назовём этот класс `GraphObject`. Классы `Pixel` и `Circle` нужно переписать, унаследовав от `GraphObject`. В классе `GraphObject` методы `Show` и `Hide` мы объявим как чисто виртуальные, так что сам этот класс будет абстрактным:

```
class GraphObject {
protected:
    double x, y;
    int color;
public:
```

[21] На самом деле из этого правила есть одно важное исключение, которое мы рассмотрим позже в §10.6.13.

```
    GraphObject(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~GraphObject() {}
    virtual void Show() = 0;
    virtual void Hide() = 0;
    void Move(double nx, double ny);
};
```

Описание функции `Move` мы не приводим, поскольку оно слово в слово повторяет описание метода `Pixel::Move` на стр. 67. Напомним, что в теле функции `Move` вызываются функции `Show` и `Hide`. То, что тела для этих функций нами не заданы, не создаёт никаких проблем, поскольку для любого класса-потомка, объекты которого можно будет создавать, таблица виртуальных методов будет содержать адреса конкретных функций, описанных в этом классе-потомке.

Заголовки классов `Pixel` и `Circle` будут теперь выглядеть так:

```
class Pixel : public GraphObject {
public:
    Pixel(double x, double y, int col)
        : GraphObject(x, y, col) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
};
class Circle : public GraphObject {
    double radius;
public:
    Circle(double x, double y, int color, double rad)
        : GraphObject(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Для этих классов, в отличие от класса `GraphObject`, необходимо описать конкретные тела методов `Show` и `Hide`, иначе программа не пройдёт этап линковки.

10.6.7. Виртуальность в конструкторах и деструкторах

Вызов виртуальных функций в телах конструкторов и деструкторов сопряжён с одной достаточно нетривиальной особенностью, связанной с конструированием и последующим деструктированием в нашем объекте невидимого поля, содержащего адрес таблицы виртуальных функций. Естественно, это поле заполняет конструктор; но, как несложно догадаться, он делает это *после того, как остальные подобъекты нашего объекта уже сконструированы*, в том числе сконструирован и предок. Между тем конструктор предка тоже, естественно, содержит код, инициализирующий указатель на таблицу виртуальных методов, а поскольку про потомков он ничего не знает, в поле `vmtp` он занесёт адрес *своей* таблицы.

Получается, что, коль скоро в классе вообще имеются виртуальные методы, **во время выполнения тела конструктора вызываются те виртуальные функции, которые описаны для данного класса, вне зависимости от того, объект какого класса на самом деле конструируется**. Для симметрии аналогичный эффект присутствует также и в телах деструкторов; после завершения своего тела деструктор «деинициализирует» указатель на виртуальную таблицу — физически это означает, что в поле `vmtp` записывается адрес таблицы виртуальных методов предка.

В частности, из конструкторов и деструкторов вообще нельзя вызывать методы, описанные в данном объекте как чисто виртуальные: вызывать будет попросту некого. Не стоит, как мы видим, уповать и на то, что объект-потомок сможет как-то скорректировать действия нашего конструктора или деструктора путём переопределения вызываемых из них виртуальных методов. При возникновении любых сомнений на этот счёт лучше вообще воздержаться от обращения к виртуальным методам из конструкторов и деструкторов.

10.6.8. Наследование ради конструктора

В практическом программировании часто применяют один упрощённый случай наследования, при котором класс-потомок отличается от предка только набором конструкторов, то есть он не вводит ни новых методов, ни новых полей. Объекты такого класса создаются из соображений экономии объёма кода, чтобы не повторять одни и те же действия при конструировании однотипных объектов. Чтобы проиллюстрировать сказанное на примере, для начала мы введём ещё один класс-потомок класса `GraphObject`, представляющий ломаную линию, а затем опишем графический объект «квадрат» как частный случай ломаной линии.

Напомним, что каждая геометрическая фигура в нашей системе имеет точку привязки; ломаную при этом проще всего хранить в виде списка координатных пар, задающих смещение каждой вершины ломаной относительно точки привязки. Для организации такого списка мы в закрытой части класса опишем структуру, задающую элемент списка. Исходно будем создавать ломаную, не имеющую ни одной вершины, а для добавления новых вершин будем использовать метод, который назовём `AddVertex`. Это лучше, чем пытаться тем или иным способом передать координаты вершин в конструктор, поскольку мы не знаем заранее количество вершин, так что для передачи их в качестве параметра конструктора пришлось бы в том месте, где создаётся объект, формировать некую динамическую структуру данных (массив либо список), что потребовало бы дополнительных усилий. Напишем заголовок класса:

```
class PolygonalChain : public GraphObject {
    struct Vertex {
        double dx, dy;
        Vertex *next;
    };
    Vertex *first;
public:
    PolygonalChain(double x, double y, int color)
        : GraphObject(x, y, color), first(0) {}
    virtual ~PolygonalChain();
    void AddVertex(double adx, double ady);
    virtual void Show();
    virtual void Hide();
};
```

Функция `AddVertex` будет (для экономии усилий) добавлять новую вершину в начало списка, а не в конец; линия при этом будет изображаться на экране в обратном порядке, но это, естественно, ничего не изменит:

```
void PolygonalChain::AddVertex(double ax, double ay)
{
    Vertex *tmp = new Vertex;
    tmp->dx = ax;
    tmp->dy = ay;
    tmp->next = first;
    first = tmp;
}
```

Поскольку наш объект использует динамическую память, ему нужен деструктор. Напишем его:

```
PolygonalChain::~PolygonalChain()
{
    while(first) {
        Vertex *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

Как и раньше, мы воздержимся от написания тел функций `Show` и `Hide`, но будем предполагать, что это сделано.

Пусть теперь нам нужен класс для представления квадрата, стороны которого параллельны осям координат, а длина стороны задаётся параметром конструктора. Ясно, что такой квадрат представляет собой

частный случай ломаной, описываемой классом `PolygonalChain`. Если в качестве точки привязки выбрать левую нижнюю вершину квадрата, а длину стороны квадрата обозначить буквой a, то ломаная должна начаться в точке привязки (что соответствует вектору $(0, 0)$), пройти через точки $(a, 0)$, (a, a), $(0, a)$ и вернуться в точку $(0, 0)$; всего, таким образом, ломаная будет содержать пять вершин, причём первая и последняя будут совпадать, чтобы сделать ломаную замкнутой.

Чтобы не приходилось каждый раз для представления квадрата писать шесть строк кода (одну для описания объекта `PolygonalChain`, остальные для добавления вершин), можно описать класс (мы назовём его `Square`), который будет унаследован от `PolygonalChain`, а отличаться будет только конструктором:

```
class Square : public PolygonalChain {
public:
    Square(double x, double y, int color, double a)
        : PolygonalChain(x, y, color)
    {
        AddVertex(0, 0);
        AddVertex(0, a);
        AddVertex(a, a);
        AddVertex(a, 0);
        AddVertex(0, 0);
    }
};
```

Подчеркнём ещё раз, что больше ничего описывать для квадрата не нужно, всё остальное сделают методы базового класса.

10.6.9. Виртуальный деструктор

Ранее при обсуждении виртуальных функций на стр. 68 мы отметили, что в классе, имеющем хотя бы одну виртуальную функцию, деструктор тоже следует сделать (объявить) виртуальным, но не объяснили причины. Попробуем сделать это сейчас.

При активном использовании полиморфизма часто приходится применять `delete` к указателю, имеющему тип «указатель на базовый класс», притом что указывать он может и на объект потомка. В этой ситуации требуется, естественно, вызвать деструктор, соответствующий типу уничтожаемого объекта, а не указателя. Так, допустим, мы описали указатель на класс `GraphObject`:

```
GraphObject *ptr;
```

Вполне корректным будет такое присваивание:

```
ptr = new Square(27.3, 37.7, 0xff0000, 10.0);
```

Как видим, `ptr` имеет тип «указатель на `GraphObject`», но реально указывает на объект класса `Square`. Если теперь потребуется уничтожить этот объект, мы можем без всяких опасений выполнить

```
delete ptr;
```

Поскольку указатель имеет тип `GraphObject*`, а деструктор класса `GraphObject` виртуальный, компилятор произведёт вызов деструктора через таблицу виртуальных методов уничтожаемого объекта. В результате этого будет вызван неявный деструктор для класса `Square`, который вызовет деструктор класса `PolygonalChain`, а тот в свою очередь — деструктор класса `GraphObject`. Если бы мы не объявили деструктор класса `GraphObject` как виртуальный, компилятор произвёл бы жёсткий вызов деструктора по типу указателя, то есть был бы вызван только деструктор класса `GraphObject`. Между тем уничтожаемый объект класса `Square` порождает и использует список в динамической памяти, и если деструктор класса `PolygonalChain` не будет вызван, то эти элементы превратятся в «мусор», то есть будут по-прежнему занимать память, не принося никакой пользы.

Объявление деструктора как виртуального практически не приводит к расходу памяти: таблица виртуальных методов увеличивается на один слот, занимая несколько лишних байтов на каждый новый *класс* (не объект!). При этом сам факт наличия в классе виртуальных функций указывает на то, что при работе с объектами класса будет активно использоваться полиморфизм. При описании класса в большинстве случаев трудно предсказать, будут ли объекты потомков такого класса уничтожаться оператором `delete`, применяемым к указателю, имеющему тип «указатель на предка». Решив, что такое удаление нам не понадобится, мы рискуем в дальнейшем забыть об этом и получить трудную для обнаружения ошибку. Именно поэтому считается, что **деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует объявлять как виртуальный, не задумываясь о том, понадобится это в программе или нет; многие компиляторы выдают предупреждение, если этого не сделать**.

10.6.10. Ещё о полиморфизме

Приведём ещё один пример использования полиморфизма. Пусть мы создаём графическую сцену[22], состоящую из разных графических объектов — точек, окружностей, многоугольников и, возможно, каких-то других элементов, представляемых объектами классов, унаследованных от `GraphObject`. При этом на момент написания программы мы не знаем, сколько и каких именно объектов будет в сцене; так бывает, если описание сцены нужно получить из внешнего источника (например, прочитать из файла), либо если сцена генерируется во время исполнения программы (например, случайным образом, что часто используется во всевозможных скринсейверах).

Допустим, в некий момент выполнения программы всё же становится известно, сколько объектов будет содержать сцена. Это позволит использовать для хранения всей сцены динамически создаваемый массив указателей на объекты потомков `GraphObject`. Пусть, например, количество объектов сцены будет храниться в переменной `scene_length`, а указатель на сам массив мы назовём просто `scene`:

```
int scene_length;
GraphObject **scene;
```

Когда переменная `scene_length` тем или иным способом получит значение (например, оно будет прочитано из файла), можно будет завести массив:

```
scene = new GraphObject*[scene_length];
```

Теперь благодаря полиморфизму оказываются корректны, например, следующие присваивания (при условии, конечно, что `i` не превышает `scene_length`):

```
scene[i] = new Pixel(1.25, 15.75, 0xff0000);
// ...
scene[i] = new Circle(20.9, 7.25, 0x005500, 3.5);
// ...
scene[i] = new Square(55.0, 30.5, 0x008080, 10.0);
// ...
```

и тому подобные. У нас получился массив указателей типа `GraphObject*`, каждый из которых на самом деле указывает на некоторый объект *класса-потомка*. Нам может вовсе никогда больше не потребоваться знать, на объекты какого типа указывают конкретные указатели в нашем массиве. Вне зависимости от конкретных типов мы вполне можем перемещать объекты по экрану, гасить их и снова отрисовывать, ведь методы `Show`, `Hide` и `Move` доступны для класса `GraphObject`, а значит, могут быть вызваны по указателю типа `GraphObject*` без уточнения типа объекта. Точно таким же образом благодаря наличию виртуального деструктора можно уничтожить все объекты сцены, а потом и саму сцену:

```
for(int i = 0; i < scene_length; i++)
    delete scene[i];
delete[] scene;
```

[22] Напомним, что под сценой в компьютерной графике обычно понимается весь набор графических объектов, видимых одновременно.

Подобные ситуации часто возникают в более-менее сложных программах. Поскольку конкретные методы, которые нужно вызывать, становятся известны только во время исполнения программы, такой вид полиморфизма называется ***динамическим полиморфизмом***; он становится возможен благодаря механизму виртуальных функций и является в конечном итоге их предназначением.

10.6.11. Приватные и защищённые деструкторы

Убирая деструктор из публичного доступа, мы можем получить своеобразные эффекты, иногда довольно полезные на практике. Если деструктор описать в секции `private:`, это будет означать, что уничтожение объекта нашего класса возможно только в его методах и дружественных функциях; деструктор в секции `protected:` добавит к этому ещё потомков нашего класса.

Объект класса, имеющего приватный деструктор, за пределами его методов и «друзей» *нельзя будет создать в виде простой (локальной или глобальной) переменной*, ведь при этом компилятор должен будет вставить в код вызов деструктора (для локальных переменных — при завершении функции, для глобальных — при завершении выполнения программы), но мы ему это делать запретили. Этот эффект иногда используют, чтобы создать тип объекта, всегда размещаемого в динамической памяти; обычно такие объекты при тех или иных обстоятельствах удаляют себя сами — например, можно предусмотреть для этого в классе специальный метод, что-то вроде

```
class A {
    ~A() {} // no destruction, dectructor is private
public:
    // ...
    void Disappear() { delete this; }
};
```

Отметим, что **приватный деструктор полностью исключает возможность наследования от такого класса**, если только заранее не описать всех наследников в качестве друзей. Напротив, защищённый (`protected`) деструктор явным образом указывает на то, что наш класс задуман как заготовка для создания наследников и что использование его самого по себе не предполагается; как показывает практика, это действует надёжнее, чем соответствующая фраза в документации.

Если все наследники нашего класса тоже будут вводить свои деструкторы в защищённой части, мы получим целую полиморфную иерархию классов, объекты которых могут существовать исключительно в динамической памяти.

Привести пример, когда что-то подобное может потребоваться, не так просто — для этого нужно подробно описать довольно сложную предметную область. Делать этого мы не будем, ограничившись замечанием, что автору книги такие ситуации встречались, и неоднократно; полезно иметь в виду сам факт существования подобных возможностей, чтобы суметь воспользоваться ими, когда ваша практическая деятельность подбросит вам подходящую задачу.

10.6.12. Перегрузка функций и сокрытие имён

Допустим, мы описали класс `A`, от него унаследовали класс `B` и при этом *в обоих классах* есть поля или методы, названные одним и тем же идентификатором (например, `x`). Вообще-то так писать в большинстве случаев не надо (кроме случая переопределения виртуальной функции), но если вы всё-таки это написали, полезно будет помнить одно важное правило языка Си++: **введение поля или метода с именем `x` в порождённом классе *скрывает* любые поля или методы базовых классов, имеющие такое же имя**[23]. Если речь идёт в обоих случаях об имени поля или о функциях-методах с одинаковым профилем (т. е. одинаковым количеством и типами параметров), правило сокрытия оказывается достаточно очевидным. Очевидно оно и для случая, когда в базовом классе имеется функция с именем `x`, а в порождённом вводится поле `x`, или наоборот — ведь для имён полей в Си++ перегрузка не предусмотрена. Например:

```
class A { // ...
public:
    void f(int a, int b);
};
class B : public A {
    double f;  // метод f(int, int) теперь скрыт
};
```

Если теперь создать объект класса `B`, вызвать метод `f` мы с ходу не сможем:

```
B b;
b.f(2, 3);   // ОШИБКА!!! Метод f скрыт
```

Однако «скрыт» не значит «недоступен» — в классе `B` по-прежнему присутствует метод `f`, унаследованный от класса `A`, просто его вызов нужно выполнять с явным указанием области видимости:

```
b.A::f(2, 3);   // всё в порядке
```

Поначалу такая конструкция создаёт жутковатое впечатление, но это быстро пройдёт; в целом здесь всё вполне логично, ведь, как мы отмечали в §10.4.13, `A::f` — это не что иное, как *имя функции*.

Наиболее неочевидным проявлением описанного правила становится то, что появившаяся в порождённом классе функция-метод с тем же именем, что и метод базового класса, **скрывает метод базового класса, даже если они различаются профилем**. Перегрузка функций нас в этом случае не спасает:

```
class A {
    // ...
public:
    void f(int a, int b);
};
class B : public A {
public:
    double f(const char *str);  // метод f(int, int) скрыт
};

B b;
double t = b.f("abrakadabra");  // всё в порядке
b.f(2, 3);                      // ОШИБКА!!!
b.A::f(2, 3);                   // всё в порядке
```

Всё это можно выразить одним простым правилом: **перегрузка имён функций действует только в рамках одной области видимости**, если же имена введены в различных (пусть и пересекающихся) областях видимости, принципы перегрузки на них не распространяются.

10.6.13. Вызов в обход механизма виртуальности

Явное указание области видимости (имени класса), использовавшееся в предыдущем параграфе для обращения к именам, которые компилятор от нас скрыл, имеет в Си++ ещё один эффект, довольно неожиданный на первый взгляд: *при вызове виртуального метода с явным указанием имени класса отключается механизм виртуальности*; функция-метод вызывается из того класса, имя которого указано при её вызове, без оглядки на таблицу виртуальных методов. Рассмотрим для примера следующие классы:

```
class A {
public:
    virtual void f() { printf("first\n"); }
    void g() { f(); }    /* вызывается метод f в зависимости
                            от фактического типа объекта */
    void h() { A::f(); } /* всегда вызывается f из класса A */
```

[23] Мы не рассматриваем в нашей книге множественное наследование, но на всякий случай отметим, что если поля или методы с одинаковыми именами появились в двух базовых классах одного порождённого класса, то в таком порождённом классе сокрытию подвергнутся имена из *обоих* базовых классов.

```
};
class B : public A {
public:
    virtual void f() { printf("second\n"); }
};
```

Проиллюстрировать разницу между `f()` и `A::f()` поможет следующий фрагмент кода:

```
B b;
b.g();      /* печатается "second" */
b.h();      /* печатается "first" */
b.f();      /* печатается "second" */
b.A::f();   /* печатается "first" */
A *pa = &b;
pa->f();    /* печатается "second" */
pa->A::f(); /* печатается "first" */
```

Эта возможность требуется сравнительно редко, но иногда оказывается весьма полезной; наиболее частый вариант её использования — когда из новой версии виртуального метода, введённой классом-потомком, нужно вызвать старую версию, описанную для класса-предка.

10.6.14. Операции приведения типа

Встречаются ситуации, когда то или иное значение меняет свой тип. Чаще всего это происходит *неявно*, как при выполнении арифметических действий над числами разной разрядности или над целым и дробным, но иногда (например, при изменении типа указателя) приходится изменять тип явно с помощью специальной операции. В чистом Си для этого предусмотрена ***операция преобразования типа*** — унарная операция, обозначаемая именем типа, заключённым в круглые скобки, примерно так:

```
int x;
char *p = (char*)&x;
```

Здесь значение выражения `&x`, имеющее тип `int*`, приводится к типу `char*`. Операция приведения типа *опасна* в том смысле, что её применение позволяет при желании обойти любые ограничения, вводимые системой типизации, включая, например, запреты на запись в константные области памяти и даже защиту данных в классах. Необдуманное применение преобразования типов приводит к запутыванию программы и к ошибкам, которые трудно найти.

Для снижения негативного эффекта операции приведения типов, а также для поддержки полиморфного программирования в языке Си++ вводятся четыре дополнительные операции, предназначенные для преобразования типа выражения. Эти операции имеют достаточно нетривиальный синтаксис: сначала записывается ключевое слово, задающее операцию (`static_cast`, `dynamic_cast`, `const_cast` или `reinterpret_cast`), затем в угловых скобках ставится имя нового типа, после чего в круглых скобках записывается само выражение, тип которого нужно изменить, например:

```
Square *sp = static_cast<Square*>(scene[i]);
```

В отличие от операции приведения типов в языке Си, которая применялась для любых случаев «ручного» (явного) изменения типа, каждая из операций Си++ предназначена для своего случая. Так, операция `const_cast` позволяет снять или, наоборот, установить сколько угодно модификаторов `const` и `volatile`[24]; попытка сделать с её помощью любое другое изменение типа вызовет ошибку при компиляции:

```
int *p;
const int *q;
const char *s;
// ...
q = p;  // можно без преобразования
p = q;  // ОШИБКА! снятие const
p = const_cast<int*>(q); // правильно
p = const_cast<int*>(s); // ОШИБКА!
```

Отметим, что наличие в языке операции `const_cast` не отменяет опасности такого преобразования. Реальная потребность в обходе константной защиты возникает крайне редко; прежде чем применять преобразование, подумайте, всё ли вы правильно делаете, не забыли ли вы, например, пометить словом `const` функцию-метод, не изменяющую состояние объекта, и т. п. **Для применения операции `const_cast` нужны очень веские причины, и сакраментальное «без неё не работает» такой причиной считать нельзя.** В некоторых программистских коллективах на каждое применение `const_cast` нужно получить личное разрешение руководителя разработки.

Операция `static_cast` предназначена для работы с наследуемыми объектами и позволяет преобразовать указатель или ссылку в направлении, противоположном закону полиморфизма, т. е. от базового класса к порождённому. Попытка произвести любое другое преобразование вызовет ошибку. Приведём примеры:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C { /* ... */ };
// ...
    A *ap;
    B *bp;
    C *cp;
    // ...
    ap = bp; // можно без преобразования
    bp = ap; // ошибка!
    bp = static_cast<B*>(ap); // допустимо
    cp = static_cast<C*>(ap); // ошибка, A и C не родственны
```

Естественно, делать это следует только когда мы *действительно* уверены, что по данному адресу расположен объект именно того типа, к которому мы намерены преобразовывать; иное в большинстве случаев приведёт к ошибкам, причём часто к таким, на локализацию которых уходит много времени.

Операция `reinterpret_cast` позволяет произвести любое преобразование (чего угодно во что угодно), если только компилятор понимает, как это сделать (в частности, преобразовать объекты структур разных типов друг к другу не получится, поскольку непонятно, как такое преобразование производить), т. е. эта операция почти эквивалентна операции языка Си, которая обозначается именем типа, взятым в круглые скобки. Единственное отличие состоит в том, что `reinterpret_cast` не умеет снимать модификатор `const` (делать то, для чего предназначен `const_cast`). Рекомендуется в любом случае применять `const_cast`, `static_cast` и `reinterpret_cast`, а не операцию Си, поскольку такие преобразования требуют особого внимания, а выражение с использованием `_cast` лучше заметно в тексте программы, чем имя типа в скобках.

Несколько особое место занимает операция `dynamic_cast`. Три операции, которые мы рассмотрели выше, служат для управления системой контроля типов, т. е. для управления компилятором, и не порождают действий, выполняемых во время работы программы[25]. Операция `dynamic_cast`, напротив, предполагает проведение нетривиальной проверки *во время исполнения программы*. Подобно операции `static_cast`, операция `dynamic_cast` предназначена для преобразования адресов объектов в направлении, противоположном закону полиморфизма, т. е. от адреса предка к адресу потомка. В случае со `static_cast` ответственность за корректность такой операции возлагается на программиста: именно программист тем или иным способом должен убедиться, что преобразуемый адрес (будь то указатель или ссылка) указывает на объект нужного типа. Если же применить `dynamic_cast`, то она сама проведёт проверку и в случае, если преобразование некорректно (то есть по заданному адресу в памяти не находится объект нужного типа),

[24] С `volatile` мы встречались во втором томе при обсуждении сигналов, см. §5.3.14.

[25] Кроме преобразования между целыми числами и числами с плавающей точкой, что требует неких действий; иногда компилятору приходится изменить и численное значение указателя, но в нашей книге не рассматриваются механизмы, порождающие такую необходимость.

вернёт нулевой указатель. На момент написания программы в общем случае тип объекта неизвестен, так что проверка проводится во время исполнения, т. е. *динамически*, отсюда название операции.

Проверка типа производится на основании значения указателя на таблицу виртуальных функций; дело в том, что такая таблица уникальна для каждого класса, имеющего виртуальные методы, т. е. её адрес однозначно идентифицирует класс объекта. Как следствие, `dynamic_cast` может работать только с классами (или структурами), имеющими виртуальные функции. В некоторых источниках такие классы называют ***полиморфными***, что не совсем корректно: как мы видели, полиморфизм в определённом смысле работает и для классов, не имеющих виртуальных функций.

Отметим ещё один немаловажный момент. Обычно реализации `dynamic_cast` весьма неэффективны по времени исполнения — попросту говоря, работают очень медленно. Поэтому злоупотреблять этой операцией не следует. В действительности будет лучше вообще без неё обойтись.

Преобразование `dynamic_cast` умеет работать не только с указателями, но и со ссылками; при этом понятия «нулевой ссылки» в природе не существует, поэтому при отрицательном результате проверки такой `dynamic_cast` выбрасывает некое «стандартное исключение». Чтобы обработать его, придётся подключить заголовочный файл стандартной библиотеки, а дальше, как говорят, коготок увяз — всей птичке пропасть. В действительности нет *никаких* причин применять `dynamic_cast` к ссылкам: если у вас ссылка, а не указатель, примените к ней операцию взятия адреса, превратив её тем самым в обычный указатель, выполните `dynamic_cast`, проверьте результат на неравенство нулю и разыменуйте его обратно.

10.6.15. Иерархии исключений

При обсуждении преобразований типов выражений в обработчиках исключительных ситуаций (см. стр. 64) мы отметили, что одним из наиболее важных видов преобразования является преобразование по закону полиморфизма, однако подробное обсуждение этого отложили, поскольку на тот момент ещё не было введено понятие наследования.

Возвращаясь к этому вопросу, заметим, что третий и последний вид допустимых преобразований от типа выражения в операторе `throw` к типу, указанному в заголовке `catch` — это преобразование адреса (т. е. указателя или ссылки) объекта-потомка к соответствующему адресному типу объекта-предка. Если мы опишем два класса, унаследовав один от другого:

```
class A { /* ... */ };
class B : public A { /* ... */ };
```

— то обработчик вида

```
catch(const A& ex) { /* ... */ }
```

сможет ловить исключения *обоих* типов, т. е. результат как оператора «`throw A(...);`», так и «`throw B(...);`».

Это свойство используется для создания *иерархий исключительных ситуаций*. Например, мы можем поделить все ошибки, возникающие в какой-либо программе или библиотеке, на следующие категории:

- ошибки, возникающие по вине пользователя:
  - синтаксические ошибки при вводе (например, буквы там, где ожидается число);
  - неправильно указано имя файла;
  - неправильно введён пароль;
  - недопустимая комбинация требований (например, одновременное требование упорядочивания по возрастанию и по убыванию);
- ошибки, обусловленные средой выполнения:
  - отсутствие файлов, нужных для работы;
  - переполнение диска;
  - прочие ошибки операций ввода-вывода;
  - нехватка оперативной памяти;
  - ошибки при работе с сетью;
  - и т. п.
- ошибки в самой программе, требующие её исправления (например, переменная принимает значение, которое вроде бы принимать не должна).

Опишем теперь класс `Error`, соответствующий понятию «любая ошибка». В полном соответствии с принципами объектно-ориентированного программирования перейдём от общего к частному, унаследовав от класса `Error` подклассы `UserError`, `ExternalError` и `Bug` для обозначения соответственно пользовательских ошибок, внешних ошибок и ошибок в программе. От класса `UserError` унаследуем классы `IncorrectInput`, `WrongFileName`, `IncorrectPassword` и т. д. После этого обработчик вида

```
catch(const IncorrectPassword& ex) { /* ... */ }
```

будет обрабатывать только исключения, связанные с неправильным паролем, тогда как обработчик вида

```
catch(const UserError& ex) { /* ... */ }
```

будет реагировать на любые ошибки пользователя, что же касается обработчика

```
catch(const Error& ex) { /* ... */ }
```

то он сможет поймать вообще любое исключительное событие из нашей иерархии. Подчеркнём, что **такое преобразование работает только для адресов**, а не для объектов как таковых; именно поэтому мы в заголовках `catch` использовали ссылки.

10.7. Шаблоны

В программировании часто возникают ситуации, в которых простейшим и очевидным решением оказывается написание нескольких почти одинаковых (а иногда и совсем одинаковых) фрагментов кода. Известно, что в таких случаях пойти «очевидным» путём — идея крайне неудачная, ведь тексты программ приходится изменять, исправлять ошибки, добавлять новые возможности, и если некий фрагмент кода будет существовать более чем в одном экземпляре, вносить изменения придётся синхронно в каждый из экземпляров. Практика показывает, что рано или поздно мы про какой-нибудь из экземпляров забудем, исправив все, кроме него; впрочем, даже если бы не это, механически дублировать одни и те же правки в нескольких местах программы оказывается делом утомительным и неприятным.

Если фрагменты, которые хочется написать для быстрого решения возникшей задачи, различаются только некоторыми *значениями* (или не различаются вовсе), бороться с дублированием кода легко: достаточно оформить такой фрагмент в виде функции, и вместо нескольких почти одинаковых кусков кода в нашей программе появятся вызовы этой функции; различаться они будут значениями параметров, причём заметить отличия при чтении такого кода будет, разумеется, гораздо проще, чем если бы приходилось каждый раз сверять большие фрагменты. Аналогичным образом обстоят дела при различии по именам переменных: мы можем в таком случае написать функцию, в которую будет передаваться адрес соответствующей переменной.

Но что делать, если нам потребовались почти одинаковые фрагменты, различия между которыми не сводятся к *значениям выражений*? Самый простой и часто встречающийся пример такого рода — различие *типов* переменных и выражений; как поступить, если нам потребовалось выполнить одни и те же действия, но в одном случае — над переменными типа `int`, а в другом — над переменными типа `double`? В языке Си в таких случаях приходится прибегать к определению макросов, причём часто — многострочных. Каждый программист, написавший хотя бы один многострочный макрос на Си, знает, какое это утомительное

и неблагодарное занятие. Если в определение макроса вкралась ошибка, найти её будет непросто, причём даже не сразу становится понятно, что ошибка кроется где-то в макросе, поскольку компилятор в сообщении об ошибке указывает не на тело макроса, а на то место, где макрос вызван. Вообще макропроцессор языка Си, из соображений совместимости включённый и в Си++, представляет собой средство опасное в применении и крайне неудобное, что многократно отмечают многие авторы, включая Страуструпа. Однако вред от дублирования кода оказывается ещё больше, так что при работе на Си в ряде случаев иного выхода не остаётся.

В языке Си++ из большинства таких положений можно выйти, используя ***шаблоны***, которые делятся на *шаблоны функций* и *шаблоны классов*. Общая идея тех и других состоит в том, что мы пишем как бы заготовку для функции или класса, которая при конкретизации некоторых параметров может превратиться в настоящую функцию или, соответственно, класс. В качестве параметров шаблона чаще всего выступают имена типов выражений, но это могут быть и значения целого типа, а в некоторых специфических случаях и адресные выражения. Разумеется, один шаблон можно использовать для создания нескольких разных функций или классов, отличающихся друг от друга только значениями некоторых параметров. Важно понимать, что **шаблон — это ещё не код, это лишь заготовка для кода**. Вполне допустимо восприятие шаблонов как этаких «умных макросов» — как минимум в том смысле, что это фрагменты текста программы, из которых неким преобразованием (с подстановкой параметров) получаются другие фрагменты текста программы, которые уже компилируются как обычно. Надо сказать, что Страуструп тщательно отрицает связь шаблонов с макросами — но, видимо, он под макросами понимает только то странное недоделанное нечто, которое мы имеем в чистом Си, тогда как в действительности макропроцессирование как явление намного сложнее.

Можно считать, что **шаблоны представляют собой ещё один вид полиморфизма** — так называемый ***параметрический полиморфизм***. Поскольку этот вид полиморфизма реализуется на стадии компиляции, его следует рассматривать как частный случай статического полиморфизма, и, как и другие случаи статического полиморфизма, к ООП он никакого отношения не имеет, что, конечно же, не делает его менее важным.

Кроме того, шаблоны можно (и нужно) считать проявлением ***метапрограммирования*** (см. §9.4.3), ведь сам шаблон — ещё не окончательный код, он лишь описание того, как этот окончательный код должен выглядеть. Как мы увидим чуть позже, возможности шаблонов намного шире, нежели тривиальные подстановки одного вместо другого, и в некоторых случаях окончательный код будет выглядеть совсем не так, как выглядел шаблон. Но — обо всём по порядку.

10.7.1. Шаблоны функций

Рассмотрим для примера функцию, сортирующую массив целых чисел методом «пузырька»[26]:

```
void sort_int(int *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
        for(int i=len-2; i>=start; i--)
            if(array[i+1] < array[i]) {
                int tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
}
```

Пусть теперь нам потребовалась такая же функция для сортировки массива из чисел типа `double`. Использовать имеющуюся функцию мы, понятное дело, не сможем: числа с плавающей точкой совершенно иначе сравниваются и имеют другой размер. Если же мы всё-таки напишем функцию `sort_double`, она будет отличаться от `sort_int` всего в двух местах: в типе параметра `array` и в типе временной переменной `tmp`, больше нигде. Поскольку дублировать код таким образом совершенно недопустимо, приходится как-то выкручиваться. В языке Си для этого пришлось бы создать с помощью директивы `#define` многострочный макрос с одним параметром, задающим как раз тип элементов сортируемого массива и, соответственно, тип переменной `tmp`.

В Си++ проблемы, подобные вышеописанной, можно решить с помощью ***шаблонов***. Как уже говорилось, шаблон — это своего рода *заготовка* для функции; сам по себе шаблон функцией не является, но может быть в неё превращён, если указать значения параметров. В данном случае параметром шаблона будет тип элементов массива. Опишем этот шаблон:

```
template <class T>
void sort(T *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
        for(int i=len-2; i>=start; i--)
            if(array[i+1] < array[i]) {
                T tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
}
```

Ключевое слово `template` сообщает компилятору, что далее последует шаблон. Затем в угловых скобках перечисляются ***параметры шаблона***, причём слово `class` означает на самом деле *произвольный тип*, а не только класс. В нашем примере шаблон имеет один параметр (с именем `T`), в качестве которого ожидается указание некоего типа. Далее следует тело функции, в котором идентификатор `T` используется в качестве типа. Ясно, что такое тело невозможно откомпилировать в объектный код, поскольку неизвестно, какой тип обозначается именем `T`, как выполнять индексирование в массиве из таких элементов (ведь размер элемента тоже неизвестен), как выполнять присваивание и сравнение. Но компилировать написанное мы и не собираемся — ведь мы же договорились, что пишем не функцию, а лишь заготовку для неё. Эта заготовка (шаблон) имеет имя `sort`.

Чтобы теперь заставить компилятор построить функцию на основе шаблона, достаточно указать конкретный тип, который будет использоваться вместо `T`. Это делается тоже с помощью угловых скобок; выражение `sort<int>` обозначает функцию для сортировки целочисленных массивов, полученную из шаблона `sort` с использованием типа `int` в качестве значения параметра `T`:

```
int a[30];
// ...
sort<int>(a, 30);
```

Таким же точно образом `sort<double>` обозначает функцию для сортировки массивов из чисел типа `double`. Более того, шаблон годится для сортировки массивов из элементов произвольного типа, нужно только, чтобы для этого типа был доступен конструктор копирования (поскольку переменная `tmp` создаётся с его помощью), были определены операция `<`, используемая в нашем шаблоне для сравнения, и операция присваивания.

Получение функции из шаблона называется ***инстанциацией*** шаблона. Компилятор инстанциирует каждую функцию только один раз,

[26] Конечно, этот метод неэффективен для массивов заметного размера, но для примера он вполне подойдёт, и к тому же на массивах из десятка элементов ничего эффективнее простого «пузырька» пока что не придумано.

т. е. если в нашем модуле трижды встретится вызов функции `sort<int>`, код для неё будет сгенерирован лишь единожды.

Интересно, что в ряде случаев инстанциированную функцию можно вызвать, не указывая параметры шаблона; так, вызов

```
sort(a, 30);
```

будет вполне корректным. По типу параметра `a` компилятор «догадается», что имеется в виду именно `sort<int>`. Это называется ***автоматическим выводом аргументов шаблона***; забегая вперёд, отметим, что для шаблонов классов такой возможности нет.

10.7.2. Шаблоны классов

Чтобы понять, зачем нужны шаблоны классов, вспомним пример, который мы приводили в §10.4.25. Напомним, что мы рассматривали *разреженный массив целых чисел*. При этом мы написали класс `SparseArrayInt` и подчинённый ему класс `Interm`. Если нам потребуется теперь разреженный массив чисел другого типа или, скажем, разреженный массив символьных строк, то код класса `SparseArrayInt` придётся дублировать с минимальными изменениями, что, как мы уже говорили, недопустимо. Существенно правильнее будет превратить существующий код для целочисленного массива в шаблон массива произвольного типа.

Конечно, такой переход от конкретного типа `int` к типу, которого мы заранее не знаем, общую картину несколько усложнит. Так, в методах класса `SparseArrayInt` мы для упрощения сделали «общим значением» очевидный в этой роли `0`, но для *неизвестного* типа этот номер не пройдёт; в самом деле, не всякий тип предусматривает значение `0`, и не ко всякому типу это значение можно преобразовать, а ограничиваться только обработкой типов, такое значение (или хотя бы такое преобразование) допускающих, было бы несколько странно. Поэтому классы, инстанциируемые из нашего шаблона, будут получать «общее значение» через параметр конструктора и сохранять в специальном поле объекта, которое мы назовём `defval`.

Второй нетривиальный момент касается набора операций над элементами массива как таковыми (в отличие от их значений). Описывая класс `SparseArrayInt`, мы воспользовались тем, что для типа `int` существует ограниченный и вполне обозримый набор операций, выполняемых над *переменной как таковой*: это обычное присваивание, десять операций присваивания, совмещённого с арифметическим действием (вроде `+=`) и четыре операции инкремента и декремента. Пятнадцать операций — это не так много, так что для объекта «класса-посредника», возвращаемого операцией индексирования, можно описать их все. Совсем другое дело, когда о конкретном наборе операций мы можем только догадываться, поскольку вообще-то не знаем тип, работу с которым нам предстоит этаким способом имитировать. Мы вернёмся к этому вопросу позже, а пока за неимением лучшего ограничимся одной только операцией простого присваивания.

Ну и ещё одна деталь: если про тип `int` мы точно знаем, что его передача по значению и лишнее копирование практически ничего не стоит, то для произвольного (заранее неизвестного) типа мы о чём-то можем только догадываться. Поэтому на всякий случай наши методы будут получать значения этого типа по константным ссылкам.

Учтя всё сказанное, перепишем заголовок класса, приведённый на стр. 57, в виде шаблона:

```
template<class T>
class SparseArray {
    struct Item {
        int index;
        T value;
        Item *next;
    };
    Item *first;
    T defval;
public:
    SparseArray(const T &dv) : first(0), defval(dv) {}
    ~SparseArray();
    class Interm {
        friend class SparseArray<T>;
        SparseArray<T> *master;
        int index;
        Interm(SparseArray<T> *a_master, int ind)
            : master(a_master), index(ind) {}
        T& Provide();
        void Remove();
    public:
        operator T();
        T operator=(const T &x);
    };
    friend class Interm;

    Interm operator[](int idx)
        { return Interm(this, idx); }
    T operator[](int idx) const;

    int NonzeroCount() const;

private:
    SparseArray(const SparseArray<T>&) {}
    void operator=(const SparseArray<T>&) {}
};
```

Обратите внимание, что везде к слову `SparseArray` мы добавляем параметр `<T>`, за исключением имени класса и имён конструкторов и деструкторов. Дело тут в том, что относительно самого класса (т. е. шаблона) компилятор и так знает, что описывается шаблон, и то же самое можно сказать про конструкторы и деструкторы; когда же речь идёт о типах параметров в функциях, о типах указателей, и, наконец, о конкретном дружественном классе — то теоретически в качестве таковых могут выступать любые классы, в том числе созданные из этого же шаблона, но с параметром, отличным от `T`. Поэтому тип приходится указывать полностью. Функции-методы нашего шаблона класса тоже придётся описывать как шаблоны. Например, описание деструктора примет следующий вид:

```
template <class T>
SparseArray<T>::~SparseArray()
{
    while(first) {
        Item *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

а присваивание для класса `Interm` нужно будет описать так:

```
template <class T>
T SparseArray<T>::Interm::operator=(T x)
{
    if(x == master->defval)
        Remove();
    else
        Provide() = x;
    return x;
}
```

При раскрытии области видимости нам тоже, как видим, приходится явно указывать, какой *класс* (а не шаблон класса) мы имеем в виду, то есть писать `SparseArray<T>`, а не просто `SparseArray`. Это и понятно: соответствующий метод будет присутствовать в каждом классе, построенном по нашему шаблону, но ведь это будут *разные* методы, хотя и построенные по одному и тому же *шаблону* метода. **Вообще, везде, где по смыслу предполагается имя типа, мы должны при использовании шаблона указать значения для параметров шаблона, чтобы получить тип; имя шаблона класса без угловых скобок и параметров используется только в трёх случаях: в начале описания шаблона (когда, собственно говоря, *задаёт-***

***ся* имя шаблона), при описании конструктора и при описании деструктора**.

Описания остальных методов мы не приводим; при желании читатель найдёт полное описание шаблона `SparseArray` вместе с небольшими тестовыми примерами в архиве примеров к книге в файле `sparse_t.cpp`.

Вернёмся теперь к проблемам, которые возникают в силу *неизвестности* типа `T`. Мы отказались от описания «всех» возможных операций над элементом массива (как таковым, в отличие от его значения); в классе `SparseArrayInt` мы могли так сделать, поскольку знали, что имеем дело с обычными целочисленными переменными, но для типа наперёд неизвестного такой подход, увы, не годится.

В действительности мы вполне могли бы описать для класса `Interm` и эти операции, и любые другие, поскольку **методы шаблона класса инстанциируются только при их использовании**; иначе говоря, инстанциирование шаблона класса не приводит к появлению в программе сразу всех описанных методов. Мы могли бы, например, снабдить шаблон операцией `<<=` (присваивание, совмещённое с побитовым сдвигом влево); для этого в публичную часть заголовка класса `Interm` нужно добавить строку вроде

```
T operator<<=(int n);
```

(на всякий случай напомним, что правый операнд операции сдвига задаёт *количество битов*, так что он по смыслу целочисленный; хотя, конечно, компилятор не станет нас заставлять соблюдать исходную семантику операций — ни этой, ни любой другой). Полное описание метода может быть, например, таким:

```
template <class T>
T SparseArray<T>::Interm::operator<<=(int n)
{
    T& location = Provide();
    location <<= n;
    T res = location;
    if(res == master->defval)
        Remove();
    return res;
}
```

Такое тело метода предполагает, что переменные типа `T` поддерживают операцию `<<=` — именно эта операция применена во второй строчке (`location <<= n`), но **даже если мы в качестве параметра шаблона `T` укажем тип, соответствующей операции не имеющий, никакой ошибки не произойдёт до тех пор, пока мы не попытаемся реально задействовать операцию в нашем коде**. Например, мы вполне можем описать переменную типа `SparseArray<Complex>` и работать с разреженным массивом комплексных чисел; у компилятора не возникнет возражений, если только мы не попытаемся комплексное число побитово сдвинуть.

Это позволяет при большом желании вернуться к идее описания для `Interm` «всех» возможных операций над «элементом массива как таковым», с тем чтобы потом просто не использовать те из них, которые конкретным типом элементов не предусмотрены. Вот только проблемы это толком не решит. В роли *неизвестного* типа может выступать класс или структура со своими полями и методами, и это, мягко говоря, *повышает* количество операций, возможных (и подразумеваемых) над переменной как таковой; слово «повышает» мы не случайно выделили, оно тут может использоваться скорее в переносном смысле, поскольку речь в действительности идёт о разнице между пятнадцатью известными операциями и, на минуточку, *бесконечностью*. В частности, если в роли `T` будет выступать тривиальная структура `str_complex` без всяких методов, с которой мы начали §10.3.1, то от переменной типа `SparseArray<str_complex>` кто-то мог бы бы ожидать поддержки чего-то вроде

```
double w = array[idx].re;
```

— но такого мы обеспечить не можем: в самом деле, операция выборки — вот эта вот точка — в Си++ не перегружается. Точно так же для «массива», описанного как `SparseArray<Complex>`, мы не сможем добиться возможности

```
double x = array[idx].GetRe();
```

Заставить подобные конструкции работать можно только путём возврата ссылки на реально существующую переменную соответствующего типа, но мы ещё в §10.4.25 выяснили, что для разреженных массивов этот вариант не подходит.

Забавно, что в наших декорациях оба эти присваивания будут прекрасно работать, если объект массива константный — например, если слово `array` в действительности не имя объекта, а имя константной ссылки на него (например, нам его так передали через параметр функции). Для типа вроде `Complex` это может даже до определённой степени показаться решением проблемы, но это лишь потому, что в классе `Complex` — в том виде, как он описан в главе 10.4 — почти все методы константные, изменить объект можно только присваиванием. Для объектов, имеющих модифицирующие методы, проблема окажется даже хуже: если имя `array` окажется константной ссылкой или (что несколько экзотично и в обычной жизни не встречается) именем константного объекта, то компилятор молча проглотит что-нибудь вроде

```
array[idx].SetMyProperty(my_value);
```

— и после этого мы долго будем искать, где оказались неправы; в отладчике всё будет выглядеть, будто строчка кода вроде бы выполняется, и даже метод вызывается, что-то там делает, вот только элемент «массива» всю эта движуху нагло игнорирует и вообще никак не меняется. Дело тут в том, что Си++ прилежно запрещает передавать временные и анонимные объекты по неконстантным ссылкам (см. §10.4.11), но в то же время почему-то разрешает для таких объектов вызовы неконстантных методов. Если за запретом передачи временных объектов по неконстантной ссылке мы признали определённую логику, даже, надо сказать, временами весьма удобную, то здесь мы вынуждены заявить, что отсутствие аналогичного запрета для вызова методов всю эту логику вдребезги расшибает.

Автору книги пару раз довелось наступить на заботливо разложенные Бьёрном свет нашим Страуструпом грабли, связанные с наличием в одном классе константной и неконстантной версии операции индексирования; если такой псевдомассив передавать в некую функцию по обычной ссылке, внутри функции вызывать какой-то модифицирующий метод, а потом в какой-то момент поменять тип параметра, добавив к нему `const`, программа по-прежнему будет проходить компиляцию, вот только корректно работать перестанет, и это вполне можно не заметить.

В действительности эта проблема довольно просто снимается: **если в вашем классе есть две версии операции индексирования, различающиеся своей константностью, и неконстантная возвращает ссылку на объект, имеющий методы, то константная версия должна возвращать константную ссылку, а не простое значение, как это вроде бы кажется логичным**. Иначе говоря, вот такого

```
MyElement &operator[](int idx);
MyElement operator[](int idx) const;
```

в вашем классе быть не должно, если только этот `MyElement` не представляет собой какой-нибудь простой числовой тип или указатель. Правильно действовать вот так:

```
MyElement &operator[](int idx);
const MyElement &operator[](int idx) const;
```

— даже если это во многих случаях неудобно. В действительности именно так стоило поступить и в нашем шаблоне по имени `SparseArray`; придумать, что возвращать для случая несуществующего элемента, предложим читателю самостоятельно.

Возвращаясь к вопросу об операциях, которые стоит (или не стоит) описывать в классе `SparseArray<T>::Interm` с тем, чтобы их можно было применять непосредственно к элементам нашего псевдомассива, мы вынужденно согласимся, что объять необъятное, по-видимому, не получится. Несмотря на это, рискнём дать для подобных случаев одну рекомендацию: на случай, если в роли типа `T` будет выступать указатель на класс или структуру, стоит предусмотреть операцию `->` (см. §10.4.22). Информации о том, как это сделать, дано уже достаточно, так что оставим это читателю в качестве упражнения.

Отметим ещё один момент, достаточно характерный для шаблонов: *далеко не всякий тип может послужить нашему шаблону параметром*, даже если мы не станем пытаться явно применить к его объектам несуществующие операции. В частности, шаблон `SparseArray` написан в предположении, что для объектов типа `T` доступны возможность опи-

сывать переменные этого типа без указания параметров конструктора, а также присваивание, копирование и операция сравнения (==). Для встроенных типов и указателей все эти условия никогда не нарушаются, но, как мы знаем, с абстрактными типами данных, какие можно ввести в Си++ (а если попросту — с объектами) всё далеко не так радужно: мы уже знаем, как для класса или структуры запретить любые из перечисленных возможностей или даже все сразу. Ну а без любой из них `SparseArray` для такого типа работать уже не будет.

В целом шаблоны с объектами типов, заданных параметром, *хоть что-нибудь, но делают*, а Си++ позволяет для объекта отдельно взятого типа запретить едва ли не что угодно, так что никакого «решения проблемы» тут предложить в любом случае нельзя. Впрочем, проблемы как таковой тут и нет. Просто нужно помнить, что любой шаблон своей реализацией накладывает ограничения на типы, используемые в роли параметров, и желательно такие ограничения отражать в документации.

10.7.3. Специализация шаблонов

Язык Си++ позволяет задать свой (отличный от общего) вид шаблона для частных случаев его параметров — попросту говоря, правило вида «в таком случае генерировать вот так, во всех остальных случаях — в соответствии с общим шаблоном».

Допустим, нам захотелось использовать шаблон функции сортировки, приведённый на стр. 75, для сортировки массива указателей на строки (указателей типа `char*`), чтобы расположить соответствующие строки в лексикографическом (алфавитном) порядке. Проблема в том, что шаблон в той его версии, которая нами рассматривалась, использует для сравнения элементов операцию «<», а эта операция хотя и определена для указателей, но никакого отношения к алфавитному порядку строк не имеет. Решение, однако, оказывается достаточно простым. Для начала заменим знак «меньше» на вызов функции, которую назовём, например, `sort_less`:

```
template <class T>
void sort(T *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
        for(int i=len-2; i>=start; i--)
            if(sort_less(array[i+1], array[i])) {
                T tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
}
```

Саму функцию `sort_less` опишем тоже с помощью шаблона:

```
template <class T>
bool sort_less(T a, T b)
{
    return a < b;
}
```

В результате для всех типов, для которых имеется операция «меньше», шаблон `sort` будет продолжать работать так же, как работал до переделки; функцию `sort_less` компилятор будет каждый раз генерировать автоматически как обычное сравнение. Нам осталось только предусмотреть особый случай для сравнения элементов типа `char*`, и тут мы как раз и прибегнем к ***явной специализации***.

Описание случая явной специализации начинается, как обычно, со слова `template`, но угловые скобки после него оставляются пустыми, чтобы показать, что это, с одной стороны, шаблон, но, с другой стороны, *этот* шаблон пишется для частного случая, не зависящего от дополнительных параметров. Далее записывается имя шаблонной функции, причём в некоторых случаях требуется вместе с именем указать все параметры шаблона. В нашем случае это не обязательно, поскольку единственный параметр шаблона выводится из типа параметров функции, так что параметр шаблона в имени функции можно опустить; но в случае, когда параметр шаблона с типами параметров функции не совпадает, указать его придётся обязательно. Впрочем, для наглядности мы его всё равно укажем. Всё вместе выглядит так:

```
template<>
bool sort_less<const char*>(const char *a, const char *b)
{
    return strcmp(a, b) < 0;
}
```

Для компилятора это означает примерно следующее: «Будь любезен, шаблон `sort_less` для случая `sort_less<const char*>` обрабатывай в соответствии с вот этим текстом шаблона, а не с каким-либо иным».

Шаблон класса тоже можно подвергнуть специализации. Например, если нам потребуется разреженный массив элементов типа `bool`, мы можем, конечно, воспользоваться шаблоном из §10.7.2, но, как легко заметить, такая реализация окажется несколько странной: ведь тип `bool` имеет всего два значения, а для разреженного массива это значит, что если элемент вообще хранится в объекте, то этот элемент имеет значение `true` (поскольку если бы он имел значение `false`, он бы в объекте не хранился: массив-то разреженный). Следовательно, элементы списка `Item` для этого случая логично бы сделать состоящими из двух, а не из трёх полей — хранить номер элемента и указатель на следующий элемент, а значение не хранить, поскольку оно и так известно. Больше того, множество целых чисел нам может показаться удобнее хранить в массиве, а не в списке, и т. д. Механизм специализаций позволяет создать отдельное описание шаблона `SparseArray` для случая `T == bool`:

```
template <>
class SparseArray<bool> {
    // ... реализация булевского разреженного массива ...
};
```

Для шаблонов *классов* язык Си++ предусматривает ***частичную специализацию***, при которой специализирующий вариант шаблона сам по себе тоже зависит от параметров. Такой специализатор начинается со слова `template`, после которого следует *непустой* список параметров шаблона; при этом параметры в заголовке шаблона, вообще говоря, могут не совпадать (или совпадать не полностью) с параметрами описываемого шаблона; фактические параметры шаблона указываются в угловых скобках после его имени. В простейшем случае о частичной специализации речь идёт, если задаётся конкретное значение одного или нескольких — но *не всех* — параметров шаблона. Так, если у нас имеется шаблон `Cls`, зависящий от двух параметров:

```
template <class A, class B>
class Cls {
    /* ... */
};
```

то можно задать специализированный вариант, например, для случая, когда параметр `B` есть тип `int`:

```
template <class X>
class Cls<X, int> {
    /* ... */
};
```

Более сложный случай частичной специализации возникает, когда в заголовке шаблона указывается некий тип (`class T`), а в описываемом типе используется указатель или ссылка на `T`, например:

```
template <class Z>
class Foo {
    /* общая реализация шаблона Foo */
};

template <class T>
class Foo<T*> {
    /* специальная реализация Foo для указателей */
};
```

В этом случае мы инструктируем компилятор, что шаблон `Foo` следует инстанциировать специальным способом, если в качестве его параметра задан тип-указатель, и обычным способом — если заданный параметр указателем не является.

Отметим, что **для шаблонов функций** (в отличие от шаблонов классов) **частичная специализация запрещена**, поскольку в сочетании с перегрузкой приводит к нежелательным последствиям. Ещё один немаловажный момент состоит в том, что в тексте программы специализированный (полностью или частично) вариант шаблона всегда должен находиться *после* общего.

10.7.4. Пример: свёртка последовательностей

Рассматривая в §9.2.3 свёртку последовательностей, мы вынуждены были признать, что примеры на языке Си выглядят не слишком убедительно. Изучив Си++ и получив в своё распоряжение шаблоны, мы можем привести более интересные примеры применения свёртки.

Для начала попытаемся обобщить понятие последовательности, насколько это возможно. Собственно говоря, условий придётся наложить только два. Во-первых, значения, входящие в последовательность, должны быть одного типа — ведь процесс свёртки подразумевает вызов одной и той же функции для каждого из членов последовательности, а Си++ пока ещё типизированный язык.

Во-вторых, последовательность должна допускать существование некой информации, идентифицирующей *текущую позицию* — так, чтобы, имея эту информацию, можно было получить текущий элемент, а также сдвинуться на следующую позицию в последовательности (т. е. получить информацию, идентифицирующую следующую позицию), и чтобы некое специальное значение текущей позиции указывало, что последовательность кончилась. Так, для обычного односвязного списка в роли такой «информации о текущей позиции» может выступать обыкновенный указатель на текущее звено списка, при этом текущий элемент и идентификатор следующей позиции мы получим с помощью операции выборки поля (например, `->val` и `->next`), а на исчерпание списка укажет нулевое значение указателя.

С массивом будет несколько сложнее: текущую позицию можно было бы идентифицировать указателем на текущий элемент, используя для получения элемента и следующей позиции операции разыменования и прибавления единицы, но узнать, что массив кончился, мы при этом не сможем. Следовательно, для массива придётся хранить ещё и количество оставшихся элементов.

Ничто не мешает рассмотреть последовательности совершенно иной природы — например, значения, читаемые из файла, или, напротив, значения, вычисляемые по мере их использования (такие как псевдослучайные числа), лишь бы эти элементы были одного типа и по последовательности можно было двигаться (в одном направлении). Всё это логично приводит нас к работе с последовательностью через некоторый *абстрактный тип данных*, идентифицирующий текущую позицию. Поскольку мы работаем на Си++, будем предполагать, что для этого у нас имеется некий объект (неважно, класс или структура), снабжённый методами

```
T Get() const;
Pos Next() const;
bool Empty() const;
```

где `T` — тип элемента последовательности, `Pos` — тип объекта текущей позиции (собственно того объекта, к которому относятся эти три метода). Обратим внимание, что метод `Next` возвращает значение следующей позиции *по значению*; иное в общем случае может оказаться недостижимым, ведь эта «следующая позиция» может вычисляться внутри самого метода `Next`, а локально вычисленное значение никак иначе вернуть не получится. Как следствие, нам придётся предположить, что объекты типа `Pos` корректно поддаются копированию; это предположение заодно несколько упростит последующую реализацию.

Пусть в роли последовательности выступает односвязный список, причём звено этого списка представляет собой открытую структуру, указатель на следующее звено называется `next`, а значение, помещённое в список, хранится в поле с именем `data`. Чтобы написать класс, удовлетворяющий сформулированным условиям для идентификации текущей позиции, нам потребуется знать ещё тип поля `data`, а также тип самого звена списка, но если эти два типа знать, то в остальном классы для разных списков окажутся совершенно одинаковыми. Воспользуемся шаблоном, имеющим два параметра-типа: `item` будет обозначать тип звена, `elem` — тип хранимой в списке информации[27]:

```
template <class item, class elem>
class ListPosition {
    item *p;
public:
    ListPosition(item *lst) : p(lst) {}
    ~ListPosition() {}
    ListPosition<item, elem> Next() const
        { return ListPosition(p->next); }
    bool Empty() const { return !p; }
    const elem &Get() const { return p->data; }
};
```

Для массива достаточно знать только тип элемента, в остальном всё будет одинаково для любых массивов: как уже говорилось, придётся хранить адрес текущего элемента и количество оставшихся элементов (исходно равное длине массива). Снова воспользуемся шаблоном класса, но на этот раз параметр будет только один — тип элемента последовательности:

```
template <class elem>
class ArrayPosition {
    elem *p;
    int rest_size;
public:
    ArrayPosition(elem *a, int len) : p(a), rest_size(len) {}
    ~ArrayPosition() {}
    ArrayPosition<elem> Next() const
        { return ArrayPosition(p+1, rest_size-1); }
    bool Empty() const { return rest_size < 1; }
    const elem &Get() const { return *p; }
};
```

Теперь самое интересное — нам предстоит написать сами функции редуцирования. Естественно, это будут (строго говоря) не функции — шаблон становится функцией только после инстанциирования. Текст шаблонов получается на удивление лаконичным:

```
template <class Pos, class Elem, class Res>
Res reduce_right(Res (*func)(Elem, Res), Pos pos, Res init)
{
    return pos.Empty() ? init :
        func(pos.Get(), reduce_right(func, pos.Next(), init));
}

template <class Pos, class Elem, class Res>
Res reduce_left(Res (*func)(Elem, Res), Pos pos, Res init)
{
    return pos.Empty() ? init :
        reduce_left(func, pos.Next(), func(pos.Get(), init));
}
```

[27] Здесь и далее — фрагменты из файла `reduce.cpp`.

Шаблоны имеют три параметра: `Pos` — тип класса, символизирующего текущую позицию и удовлетворяющего нашей спецификации, то есть имеющего методы `Get`, `Next` и `Empty`; `Elem` — тип элемента последовательности; `Res` — тип результата. Напомним, что при редукции последовательностей последние два типа могут совпадать, и даже достаточно часто совпадают — но в общем случае могут и не совпадать, и наши шаблоны это позволяют. Стоит обратить внимание, что в наших шаблонах как для левой, так и для правой редукции функция, по которой производится редуцирование, принимает первым параметром очередной элемент последовательности, а результат предыдущего шага редукции принимает вторым параметром.

Теперь можно, написав соответствующие функции, попробовать редукцию в деле. Начнём с самых простых вариантов: суммы и произведения элементов последовательности. Соответствующие функции, чтобы долго не возиться, сделаем шаблонными:

```
template <class t>
t sum(t x, t y)
{
    return x + y;
}

template <class t>
t prod(t x, t y)
{
    return x * y;
}
```

Для начала попробуем поработать с массивом — введём, например, такой простенький массив:

```
    int array[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Посчитать сумму и произведение его элементов можно с помощью как правой, так и левой редукции:

```
int res1 =
    reduce_left(sum<int>, ArrayPosition<int>(array, 10), 0);
int res2 =
    reduce_right(sum<int>, ArrayPosition<int>(array, 10), 0);
int res3 =
    reduce_left(prod<int>, ArrayPosition<int>(array, 10), 1);
int res4 =
    reduce_right(prod<int>, ArrayPosition<int>(array, 10), 1);
```

В результате переменные `res1` и `res2` получат значение `55`, а `res3` и `res4` — значение `3628800`.

Рассмотрим пример посложнее: с помощью редукции построим *список* из элементов заданного массива. Список будет состоять из звеньев такого типа:

```
struct int_item {
    int data;
    struct int_item *next;
};
```

Для создания элемента списка предусмотрим такую функцию:

```
struct int_item *make_il(int d, struct int_item *rest)
{
    struct int_item *p = new int_item;
    p->data = d;
    p->next = rest;
    return p;
}
```

и по ней проведём правую редукцию массива:

```
    struct int_item *list =
        reduce_right(make_il, ArrayPosition<int>(array, 10),
                     (int_item*)0);
```

Здесь стоит обратить внимание на то, что тип последнего аргумента нам пришлось явным образом привести к `int_item*`, иначе компилятор не смог бы вывести третий параметр шаблона `reduce_right`.

Результатом каждого шага редукции здесь становится очередное звено списка, оба поля которого уже заполнены: `data` — значением из очередного элемента массива, `next` — адресом «хвоста» списка, построенного в ходе предыдущих шагов редукции. Напомним, что правая редукция означает применение функции к элементам последовательности справа налево.

Итак, указатель `list` у нас теперь указывает на начало списка, содержащего числа от `1` до `10`. Это позволяет нам попробовать наши функции редукции на списке; для начала посчитаем произведение с помощью левой редукции (на сумму и правую редукцию время тратить не будем, они делаются совершенно аналогично):

```
int res5 =
    reduce_left(prod<int>, ListPosition<int_item, int>(list), 1);
```

Переменная `res5` получит уже знакомое нам значение `3628800`; примечательно здесь, пожалуй, то, что нам не пришлось изобретать новую функцию, мы воспользовались той же функцией `prod<int>`, что и для массивов. В действительности нашим функциям, по которым мы проводим редукцию, вообще безразлично, какова природа последовательности.

Теперь позволим себе небольшую вольность и сделаем то, чего функциональщики обычно не делают — применим редукцию к функции с побочным эффектом. Функция будет такая:

```
struct int_item *print_il(int d, struct int_item *rest)
{
    printf("%d ", d);
    return 0;
}
```

Отметим, что такой тип возвращаемого значения выбран здесь исключительно по той причине, что он должен совпадать с типом второго параметра; функция, как видим, всегда возвращает `0` — за неимением лучшей идеи. Теперь левая редукция позволит распечатать элементы списка в прямом порядке, а правая — в обратном:

```
    reduce_left(print_il, ListPosition<int_item, int>(list),
                (int_item*)0);
    printf("\n");

    reduce_right(print_il, ListPosition<int_item, int>(list),
                (int_item*)0);
    printf("\n");
```

Возможности редукции на этом не исчерпываются. Довольно очевидно, как применить её для создания копии списка, при этом правая редукция создаст обычную копию, а левая (с использованием той же функции) — перевёрнутую; можно при копировании списка выбросить из него некоторые элементы, можно подвергнуть элементы какому-нибудь преобразованию и так далее.

Читатель может справедливо заметить, что перечисленные задачи легко решаются без всякой редукции, причём в большинстве случаев решение получится и короче, и эффективнее. Согласившись с этим, мы всё же ответим, что в этом параграфе продемонстрировали, с одной стороны, технику редукции последовательностей, которую обычно считают эксклюзивной принадлежностью функциональных языков, а с другой — выразительную мощь механизма шаблонов Си++; и то, и другое стоит потраченного времени, даже если на практике решения, подобные приведённым выше, применяться не будут. В конце концов, наш основной предмет обсуждения сейчас — парадигмы программирования.

10.7.5. Константы в роли параметров шаблона

Параметрами шаблонов могут быть не только типы, но и обычные константные выражения, чаще всего целочисленные. Многие авторы иллюстрируют эту возможность на примере задания границ массивов; так, мы могли бы в нашем шаблоне для сортировки сделать размер массива параметром *шаблона*, а не функции:


```
template <class T, int len>
void sort(T *array)
{
    /* реализация сортировки */
}
```

— вот только не совсем понятно, зачем это делать. Если, скажем, у нас имеются два целочисленных массива разного размера (например, 10 элементов и 15) и мы применим для их сортировки такой вот шаблон, то компилятор сгенерирует *две разные* функции: `sort<int,10>` и `sort<int,15>`, и эти функции будут представлять собой абсолютно одинаковые фрагменты исполняемого кода, отличающиеся только одной константой, тогда как если применять шаблон в том виде, в котором мы его писали ранее, функция будет *одна* (`sort<int>`).

Существенно интереснее (хотя и намного сложнее) будет другой пример. Зададимся целью создать объект, реализующий N-мерные массивы для произвольных значений N, которые к тому же будут автоматически изменять свои размеры по любому из измерений, подобно тому, как изменял свою длину массив `IntArray`, рассмотренный нами в §10.4.20. Поскольку нам могут понадобиться N-мерные массивы элементов разных типов, реализуем нашу задумку в виде шаблона класса с тремя параметрами: первый параметр будет задавать тип элементов массива, второй параметр (имеющий тип, совпадающий с типом элемента) укажет, какое исходное значение присваивать элементам массива при их создании, и, наконец, третий параметр — выражение типа `int` — будет задавать *количество измерений* массива. Шаблон будет описывать операцию индексирования, которая возвращает ссылку на объект, заданный таким же шаблоном с теми же фактическими параметрами, только с уменьшенным на единицу количеством измерений. Ясно, что такой вариант не подходит для одномерного массива (не может же он, в самом деле, возвращать ноль-мерный массив, ведь такого не бывает), но здесь нам поможет частичная специализация: для случая количества измерений, равного одному, мы опишем специальный случай нашего шаблона, в котором операция индексирования будет возвращать ссылку на простой элемент (тип которого задан первым параметром шаблона).

С реализацией одномерного случая всё более-менее понятно, достаточно взять уже знакомый нам `IntArray` и переделать его в шаблон. Что касается всех многомерных случаев, то самый простой способ их реализации — взять всё тот же динамически расширяемый массив, элементами которого на сей раз будут выступать *указатели* на объекты, представляющие массив на единицу меньшей размерности: двумерный массив будет реализован как массив указателей на объекты одномерных массивов, трёхмерный — как массив указателей на двумерные и т. д. Для экономии памяти указатели исходно будут нулевыми; соответствующие объекты $(N-1)$-мерных массивов будут создаваться только при первом обращении к ним.

Итак, в основе обоих случаев нашего массива (как одномерного, так и двумерного) оказывается одномерный массив, автоматически расширяющийся при необходимости. Дважды реализовывать эту сущность совершенно ни к чему, тем более что в нашем распоряжении имеется механизм шаблонов. Начнём с реализации динамически расширяющегося массива с произвольным типом элементов:

```
template <class T>
class Array {
    T *p;
    T init;
    unsigned int size;
public:
    Array(T in) : p(0), init(in), size(0) {}
    ~Array() { if(p) delete[] p; }
    T& operator[](unsigned int idx) {
        if(idx >= size)
            Resize(idx);
        return p[idx];
    }
    const T& operator[](unsigned int idx) const {
        return idx >= size ? init : p[idx];
    }
    int Size() const { return size; }
private:
    void Resize(unsigned int required_index) {
        unsigned int new_size = size==0 ? 8 : size;
        while(new_size <= required_index)
            new_size *= 2;
        T *new_array = new T[new_size];
        for(unsigned int i = 0; i < new_size; i++)
            new_array[i] = i < size ? p[i] : init;
        if(p)
            delete[] p;
        p = new_array;
        size = new_size;
    }
       // запретим копирование и присваивание
    void operator=(const Array<T>& ref) {}
    Array(const Array<T>& ref) {}
};
```

Подробно комментировать этот шаблон мы не будем, поскольку реализация почти дословно повторяет реализацию класса `IntArray`, остановимся только на отличиях. Во-первых, мы добавили здесь поле `init`, значение которого, задаваемое в конструкторе, присваивается новым (не существовавшим ранее) элементам массива при его расширении. Во-вторых, размер массива исходно принимается нулевой, а не 16, как в `IntArray`. Кроме того, константная версия операции индексирования сделана возвращающей константную ссылку, а не значение, как в `IntArray` (см. замечание на стр. 77). Ну и, конечно, класс преобразован в шаблон, что тоже существенно. Имея этот шаблон, мы можем легко реализовать общий случай нашего многомерного массива. Новый шаблон мы назовём `MultiMatrix`; напомним, что он должен работать для любого количества измерений, бóльшего единицы, а одномерный случай мы потом опишем путём специализации.

```
template <class T, T init_val, int dim>
class MultiMatrix {
    Array<MultiMatrix<T, init_val, dim-1>*> arr;
public:
    MultiMatrix() : arr(0) {}
    ~MultiMatrix() {
        for(int i=0; i < arr.Size(); i++)
            if(arr[i])
                delete arr[i];
    }
    MultiMatrix<T, init_val, dim-1>&
    operator[](unsigned int idx) {
        if(!arr[idx])
            arr[idx] = new MultiMatrix<T, init_val, dim-1>;
        return *arr[idx];
    }
    const MultiMatrix<T, init_val, dim-1>&
    operator[](unsigned int idx) const {
        static MultiMatrix<T, init_val, dim-1> dummy;
        return arr[idx] ? *arr[idx] : dummy;
    }
};
```

Мы воспользовались здесь шаблоном `Array` для построения массива *указателей* на элементы типа «$(N-1)$-мерный массив»; сам этот тип задаётся через сам же шаблон `MultiMatrix`, причём с теми же параметрами, что и у исходного шаблона, отличается лишь третий параметр, задающий количество измерений. Получается, что описание шаблона `MultiMatrix` в определённом смысле рекурсивно.

Объект массива указателей мы назвали `arr`. Напомним, что классы, построенные по шаблону `Array`, принимают в качестве параметра конструктора начальное значение, исходно присваиваемое элементам массива; в данном случае этот параметр — константа `0`, что означает нулевой указатель. Исходно все элементы массива нулевые, и только при первом обращении к соответствующему элементу создаётся объект $(N-1)$-мерного массива (см. оператор `if` в теле операции индексирования). Стоит обратить внимание, что нам здесь пришлось описать нетривиальный деструктор. Дело в том, что объект класса, инстанциированного из `Array`, ничего не знает о природе элементов, которые хранит; свой внутренний массив, в котором хранятся указатели, объект `arr` уничтожит в своём собственном деструкторе, но об объектах, *на которые* указывают указатели, хранимые в `arr`, нам приходится заботиться самостоятельно.

Довольно странной может показаться на первый взгляд реализация константной версии операции индексирования; но возвращать в качестве ссылки разыменованный нулевой указатель — совсем никуда не годится, это заведомо приведёт к ошибкам. В то же время объект, описанный под именем `dummy`, будет (на всех уровнях) представлять собой массив, не содержащий ни одного элемента — ссылку на ровно такой же объект возвращает и неконстантная версия операции, только в её случае этот объект только что создан и может быть изменён; объект, который возвращает константная версия операции, изменить всё равно нельзя (если только не обойти защиту, предоставленную модификатором `const`, но тогда это уже будут не наши проблемы).

Опишем теперь базисный случай — одномерный массив, то есть специализированный вариант шаблона `MultiMatrix` для случая, когда третий параметр шаблона равен единице. Это можно сделать гораздо проще, поскольку новый класс фактически представляет собой обёртку вокруг соответствующего объекта `Array` (он, как и раньше, называется `arr`). Сам объект класса `Array` можно сделать приватным полем; обёртка введёт операции индексирования, реализованные через такие же операцию объекта-массива, и задаст аргумент его конструктору, больше от неё ничего не требуется:

```
template <class T, T init_val>
class MultiMatrix<T, init_val, 1> {
    Array<T> arr;
public:
    MultiMatrix() : arr(init_val) {}
    T& operator[](unsigned int idx) {
        return arr[idx];
    }
    const T& operator[](unsigned int idx) const {
        return arr[idx];
    }
};
```

Операторы `return` в обеих версиях операции индексирования выглядят абсолютно одинаково, но первая вызовет неконстантный вариант операции индексирования из объекта `arr`, а вторая — константный; в самом деле, поля любого объекта считаются константными (не подлежащими изменению) в константных методах этого объекта, и поля `arr` при обращении к нему из константной версии операции индексирования это тоже касается.

Можно поступить ещё лучше — *унаследовать* наш частный случай от `Array<T>`, тогда и операции индексирования описывать не придётся:

```
template <class T, T init_val>
class MultiMatrix<T, init_val, 1> : public Array<T> {
public:
    MultiMatrix() : Array<T>(init_val) {}
};
```

К сожалению, такой упрощённый вариант подходит только для нашего частного случая — того, при котором шаблон `MultiMatrix` превращается в простой одномерный массив, содержащий элементы типа `T`. Дело в том, что во всех остальных случаях — т. е. когда параметр `dim` превышает `1` — массив, получаемый из шаблона `Array`, хранит *указатели* на объекты `MultiMatrix` с `dim`, уменьшенным на единицу, тогда как операции индексирования должны на эти же объекты возвращать *ссылку*, для чего к выражению `arr[idx]` там применяется разыменование (возвращается `*arr[idx]`). Попытка написать что-то вроде

```
template <class T, T iv, int dim>
class MultiMatrix : public Array<MultiMatrix<T, iv, dim-1> > {
public:
    MultiMatrix() {}
};
```

провалится в том смысле, что нам всё равно придётся описать здесь свои собственные операции индексирования, только вместо компактного и понятного имени поля `arr` при вызове операций индексирования из базового класса придётся этот класс упоминать явно — а это, на минуточку, `Array<MultiMatrix<T, init_val, dim-1> >`, вот прямо так, целиком.

Может показаться логичной идея хранить в массивах не указатели, а прямо «живьём» объекты тех же `MultiMatrix`'ов с соответствующим `dim`, но эта идея тоже не проходит. Что такой объект сам по себе, даже когда не содержит элементов, всё же по размеру изрядно больше указателя — это ещё полбеды. Что для создания массивов этих объектов потребуется конструктор по умолчанию — тоже не проблема. Хуже, что в методе `Resize` элементы массивов *присваиваются*, так что при таком решении придётся операцию присваивания, которую мы намеренно запретили, разрешить обратно, и не просто разрешить, а корректно реализовать; следствием этого станет *копирование* всех подчинённых структур данных при каждом изменении размера массива, что уже вообще никуда не годится.

Полученный шаблон можно проверить, например, с помощью такой функции `main`:

```
int main()
{
    MultiMatrix<int, -1, 5> mm;
    mm[3][4][5][2][7] = 193;
    mm[2][2][2][2][2] = 251;
    printf("%d %d %d %d\n",
        mm[3][4][5][2][7], mm[2][2][2][2][2],
        mm[0][1][2][3][4], mm[1][2][3][2][1]);
    return 0;
}
```

Программа напечатает строку «`193 251 -1 -1`». Текст этого примера читатель найдёт в файле `multimatrix.cpp` из архива примеров к книге.

Параметром шаблона может быть константа любого целого типа, известная на момент компиляции; но целыми числами возможности параметров шаблонов не исчерпываются. В роли параметра шаблона могут выступать адресные выражения (указатели), но здесь действует целый ряд ограничений. Фактическим значением адресного параметра шаблона может быть только адрес глобальной переменной или функции[28], причём только в форме `&var`, `&f` или просто `f`, где `var` — имя глобальной переменной, `f` — имя функции. Недопустимо использовать сложные адресные выражения, нельзя использовать адреса локальных переменных (что и понятно, ведь их невозможно определить во время компиляции, когда происходит инстанциация шаблонов), нельзя использовать строковые литералы (строки в двойных кавычках; это тоже можно понять, ведь у двух одинаковых строковых литералов совершенно не обязан совпадать адрес, а если они появились в разных единицах трансляции, то их адреса наверняка будут разными). Впрочем, в реальной жизни очень редко применяется что-либо, кроме типов; даже целочисленные параметры шаблонов — это скорее экзотика.

Рассмотрев `MultiMatrix` во всех подробностях, в заключение отметим, что у нас снова получился ***контейнер***, причём в этот раз ещё и ***обобщённый*** в том смысле, что одна (шаблонная) реализация способна работать с широким спектром типов хранимых значений (слово «обобщённый» в данном контексте соответствует английскому термину *generic*). Мы уже говорили это на стр. 54, но на всякий случай повторим ещё раз: **никогда не применяйте контейнеры на практике**, в особенности вот такие «обобщённые». Если вам кажется, что какое-то подобное сооружение способно сэкономить ваше время и трудозатраты, то это вам только кажется, и совершенно неважно, сами вы напишете очередной контейнер или его кто-то за вас уже написал: в долгосрочной перспективе и то, и другое выльется в совершенно несуразные потери.

[28] Ещё можно использовать так называемый указатель на член класса, но мы эту сущность не рассматриваем.

10.8. Снова о парадигмах

Мы рассмотрели все возможности Си++, которые планировали, и сейчас самое время привести в порядок полученные знания и понять, куда мы пришли.

10.8.1. Спектр парадигм в Си++

В §9.3.1 мы перечислили семь возможных уровней совместимости между языком программирования и заданной парадигмой. Попробуем понять, как с различными парадигмами соотносится Си++.

Пожалуй, можно с некоторой натяжкой сказать, что Си++ никаких парадигм не *навязывает* — во всяком случае в том смысле, в котором это слово использовалось в §9.3.1. В самом деле, в Си++ можно обойтись даже без императивного программирования, хотя вряд ли будет комфортно. Есть только один момент: Си++ — язык, предполагающий компиляцию, так что если рассматривать компилируемую модель исполнения программы как парадигму (а мы ближе к концу книги такое рассмотрение проведём), то эту парадигму можно считать навязанной, иное в Си++ невозможно.

Возможно, Си++ ничего не навязывает, но *понуждает* он к применению целого ряда парадигм, и прежде всего — императивного программирования. Без присваивания в Си++ обойтись с формальной точки зрения возможно, но в реальности это превратит процесс программирования в ад.

Интереснее всего будет выглядеть список парадигм, применение которых Си++ *поощряет*, то есть без них можно обходиться, но если начать их применять, то это немедленно даст ощутимый выигрыш. Прежде всего это, конечно, объектно-ориентированное программирование и абстрактные типы данных; им мы посвятим отдельный параграф. Кроме того, к поощряемым парадигмам здесь относятся обработка исключений и ***обобщённое программирование*** (*generic programming*), представленное механизмом шаблонов.

К поддерживаемым, но не поощряемым в Си++ можно отнести разве что макропроцессирование как таковое (если не считать шаблоны макросами).

Перечислить парадигмы, *допускаемые* языком Си++, пожалуй, невозможно — просто потому, что в природе не встречается списков парадигм, которые могли бы всерьёз претендовать на полноту. Какую бы парадигму мы ни рассматривали, скорее всего, при некотором старании её применение в Си++ окажется возможно, в особенности если такое применение поддержать какой-нибудь специально для этого придуманной библиотекой.

Что касается двух оставшихся уровней — то парадигм, применению которых Си++ бы всерьёз *препятствовал* или тем более начисто *запрещал*, автору с ходу придумать не удалось. Впрочем, вряд ли кто-то рискнёт всерьёз утверждать, что таких нет.

10.8.2. ООП и АТД

Как уже говорилось, эти две парадигмы люди часто и безосновательно путают. Принципиальные различия между ними мы уже обсуждали (см. стр. 16); попытаемся понять, где границы поддержки одного и другого в языке Си++.

В принципе всё, что в Си++ можно делать с классами и структурами такого, что не было предусмотрено в чистом Си, относится либо к ООП, либо к АТД, либо и к тому, и к другому. Самый очевидный пример варианта «и к тому, и к тому» — механизм защиты, представленный ключевыми словами `public`, `private`, `protected` и `friend`; он же выступает главной причиной путаницы. Также к поддержке обеих парадигм можно отнести конструкторы и деструкторы (сам факт их существования). В то же время многообразие видов конструкторов — конструкторы по умолчанию, конструкторы преобразования и копирования — не имеет никакого отношения к объектно-ориентированному программированию, это сугубо инструмент контроля за поведением экземпляров абстрактного типа данных (но при этом инструмент, надо сказать, мощнейший и очень удобный). То же самое можно сказать и про переопределение символов операций (функции и методы со словом `operator` в названии); наличие этого механизма делает абстрактные типы данных полноценными, выводит их на один уровень с АТД, введёнными самим языком, так что возможность применения инфиксных операций к пользовательским данным очень важна (и совершенно напрасно некоторые программисты эту возможность недооценивают), но, опять-таки, никакого отношения к объектно-ориентированному программированию переопределение операций не имеет.

Что же, в таком случае, остаётся на долю ООП? Прежде всего, как ни странно, *методы*, то есть сама возможность сделать функцию составной частью типа данных. Для абстрактных типов данных наличие методов не обязательно, вполне хватило бы директивы `friend` для обозначения функций, которым разрешён доступ к внутреннему устройству типа; точно так же внешними функциями (не методами) могли бы быть конструкторы и деструкторы. Но если при построении абстрактного типа данных можно обойтись без методов, то воспринимать нечто как объект в смысле ООП, не имея методов, можно лишь при наличии уже сформировавшихся устойчивых навыков объектно-ориентированного мышления. Автор этих строк, имея за плечами два десятка лет преподавания, всё-таки не взялся бы *учить* кого-нибудь объектно-ориентированному программированию на примере языка, не поддерживающего методы.

Впрочем, методы сами по себе — это, как было сказано на стр. 16, ещё не совсем ООП; во всей полноте эта парадигма раскрывается лишь с введением наследования — и, естественно, виртуальных функций, влекущих за собой динамический полиморфизм. Для объектно-ориентированного программирования всё, что связано с наследованием — возможности ключевые, тогда как к АТД наследование вообще никаким образом не относится.

10.8.3. Полиморфизм без мистики

Как отмечалось ранее (см. комментарий на стр. 65), в литературе можно встретить различные трактовки термина «***полиморфизм***». Достаточно часто полиморфизм жёстко связывают с *типами*, а не операциями, как это делали мы. В принципе это дело вкуса; в любом случае, как бы ни формулировалось «определение» полиморфизма, в поясняющих примерах тут же возникают операции — функции, методы, арифметические символы — поскольку без них полиморфизм «проявить» невозможно. Если всё-таки говорить об операциях, то полиморфизм можно понимать в том смысле, что *одна и та же операция корректно работает с объектами*[29] *различных типов*, но можно понимать и иначе: что одно и то же *обозначение операции* соответствует *различным* операциям, среди которых выбор производится в зависимости от типов операндов.

Надо признать, что второй вариант выглядит логичнее. В самом деле, если у нас есть символ «+», обозначающий сложение целых чисел, сложение чисел с плавающей точкой, сложение комплексных чисел, сложение каких-нибудь (естественно, введённых пользователем) векторов и матриц и, в довершение картины, конкатенацию текстовых строк — то это *одна* операция, работающая по-разному для разных типов, или всё-таки *много разных операций*, обозначенных одним символом? Весь наш опыт требует считать, что это именно *разные операции*, они ведь даже могут быть реализованы в разных местах: сложение чисел (как целых, так и «плавучих») встроено в язык, конкатенация строк могла прийти из какой-нибудь библиотеки (в том числе «стандартной», если не удержаться и всё-таки начать её использовать), а для векторов с матрицами мы операцию сложения, скорее всего, ввели сами.

Так-то оно так, но есть одна проблема. Если полиморфизм состоит в способности транслятора к выбору (в зависимости от типов данных) одной из многих операций, имеющих одинаковое обозначение, то получится, что свойство применимости всех операций, доступных для ба-

[29] В данном случае слово «объект» не подразумевает объектно-ориентированного программирования; в английской литературе в таких случаях используется термин *entity*.

зового класса, к объекту класса-потомка (проявляющееся, как мы знаем, без всякой виртуальности, просто за счёт сохранения расположения полей) — полиморфизмом не является, да и вообще полиморфизмом перестаёт быть явление, которое мы называли ***полиморфизмом адресов***, состоящее в том, что указатель или ссылку на объект-потомок можно использовать везде, где требуется указатель или ссылка на объект-предок. С этим можно было бы просто смириться, но пользоваться такой терминологией будет неудобно, мы ведь потеряем заодно термин «закон полиморфизма», означающий допустимость неявного преобразования адресов и ссылок от типа потомка к типу предка, а этот термин мы довольно активно использовали.

Решение для этой терминологической коллизии можно предложить самое простое: *ну и чёрт с ней*. Никто не мешает рассматривать оба варианта понимания полиморфизма как допустимые, благо из контекста всегда можно понять, о чём идёт речь. В конце концов, никого ведь не смущают два совершенно разных понимания *ввода-вывода* и две — опять же, абсолютно разные — ипостаси *файловой системы* (см. т. 2, §§5.2.1 и 8.4.1).

Увлёкшись рассмотрением полиморфизма в разнообразных новомодных «динамических» (читай — интерпретируемых) языках, многие авторы забывают про фундаментальный фактор классификации его проявлений; но при работе на статически компилируемом языке, каким является Си++, мы себе такой забывчивости позволить не можем. Проявления полиморфизма следует относить к совершенно разным видам в зависимости от того, *когда* (на каком этапе жизненного цикла программы) происходит выбор между разными версиями одной операции (или разными операциями, обозначенными одним символом). Когда такой выбор компилятор может сделать сам во время компиляции, говорят о ***статическом полиморфизме***, если же во время компиляции выбор сделать невозможно и приходится его откладывать до времени исполнения программы, речь идёт о полиморфизме ***динамическом***.

Динамический полиморфизм в Си++ представлен только одним инструментом — виртуальными методами; именно при их вызове происходит выбор конкретной реализации функции в зависимости от действительного типа объекта, находящегося в памяти по заданному адресу. Со всеми остальными проявлениями полиморфизма компилятор справляется сам, не привлекая средства времени исполнения.

В частности, к статическому полиморфизму относится перегрузка функций, важным частным случаем которой в Си++ выступает переопределение символов операций для пользовательских типов. Этот вид полиморфизма часто называют ***полиморфизмом ad hoc***. Латинское словосочетание *ad hoc* обычно используется для обозначения решений, предназначенных для одной конкретной проблемы, одного частного случая, временных решений и т. п.; в применении к полиморфизму использование этого словосочетания может быть оправдано тем, что для одного конкретного обозначения функции или операции явным образом описывается её реализация для каждой возможной комбинации операндов, и такое описание не допускает обобщений: если для какой-то комбинации типов операция не описана, то она к этой комбинации типов неприменима.

Второй важный тип статического полиморфизма представлен в Си++ шаблонами функций; обычно его называют ***параметрическим полиморфизмом***. Такое название основано на том, что некий *тип*, определяющий, какая операция (если угодно, какая реализация операции) будет применяться, задаётся *параметром*. Это тот случай, когда полиморфизм действительно хочется связать прежде всего с типом, а не с операцией; впрочем, почему бы и нет. Кстати, если сравнить полиморфизм ad hoc с полиморфизмом параметрическим, смысл словосочетания *ad hoc* может стать яснее: очевидно, что при параметрическом полиморфизме множество типов, к которым применима операция (или, если угодно, её шаблон, из которого потом получаются реализации), остаётся открытым, никто не перечисляет явно все возможные случаи её применения.

К статическому полиморфизму относится, несомненно, и введённый нами полиморфизм адресов вместе с преобразованием адресов по закону полиморфизма — если, конечно, вообще рассматривать это как полиморфизм.

10.8.4. Наследование как сужение множества

> Всякая селёдка — рыба, но не всякая рыба — селёдка.
>
> *А. Некрасов. Приключения капитана Врунгеля.*

Как уже говорилось, при описании объектно-ориентированного программирования можно использовать различные варианты терминологии. Так, термин «вызов метода объекта» эквивалентен термину «отправка сообщения объекту», просто эти термины относятся к разным терминологическим системам: о вызовах методов мы говорим при изучении практического применения ООП, тогда как передача сообщения — термин, относящийся к теории ООП и к тем языкам программирования, которые полностью соответствуют этой теории (к таким языкам относится, например, Smalltalk).

Класс с теоретической точки зрения представляет собой *множество* объектов, удовлетворяющих определённым условиям. В этом смысле порождённый (наследуемый, или дочерний) класс представляет собой *подмножество*. В самом деле, согласно закону полиморфизма объект порождённого класса может быть использован в качестве объекта базового класса, то есть, попросту говоря, является одновременно и объектом порождённого, и объектом базового класса; в то же время объект базового класса вовсе не обязан быть объектом класса порождённого.

Можно прийти к тем же выводам и иначе. Наследование представляет собой *уточнение* свойств объекта, или, иначе говоря, переход от общего случая к частному. Ясно, что множество частных случаев есть подмножество множества случаев общих. Получается, что в терминах множеств наследование описывается отношением включения ($A \supseteq B$). Естественным следствием такого рассмотрения является термин ***подкласс*** (англ. *subclass*) для обозначения порождённого класса и термин ***надкласс*** или ***суперкласс*** (англ. *superclass*) — для обозначения базового.

Такая терминология создаёт определённую путаницу. Дело тут в том, что объект порождённого класса (*подкласса*) мало того, что памяти занимает заведомо не меньше (а при добавлении новых полей — совершенно точно больше), нежели объект класса базового (*суперкласса*), но ещё и *содержит в себе объект базового класса*, т. е. объект суперкласса оказывается *подобъектом* объекта подкласса. Получается, что мы рассматриваем одновременно два отношения вложенности, и они мало того что разные, они оказываются *направлены противоположно*: объект базового класса вложен в объект порождённого класса (чисто технически), а вот сами классы (уже как множества объектов) вложены, наоборот, порождённый в базовый.

Вся эта путаница обусловлена применением двух разных терминологических систем в одном месте. Когда речь идёт о суперклассах и подклассах — это значит, что используется теоретико-множественная терминология. Когда же речь заходит об используемой памяти и подобъектах — очевидно, что разговор идёт в терминах реализаторской (прагматичной) точки зрения. Мы уже встречались с этим дуализмом в §10.6.3 (см. замечание на стр. 66). Обе терминологические системы активно используются и имеют право на существование; вы на практике можете столкнуться как с одним вариантом терминологии, так и с другим, а в некоторых случаях — и с их смешением, как в вышеприведённом примере. Поэтому желательно понимать, что означают термины обеих систем.

10.9. Особенности оформления кода на Си++

10.9.1. Соглашения об именах

Первоначально в Си++ действовали те же соглашения об именах, которые привычны программистам на чистом Си: макросы именовались

большими буквами, всё остальное — маленькими. Следы той эпохи всё ещё сохранились в именовании типов и прочих сущностей стандартной библиотеки: все стандартные классы, шаблоны и т. п. поименованы с использованием букв нижнего регистра. Со временем, однако, традиции несколько изменились. Читатель наверняка обратил внимание, что, введя в §10.3.5 слово `class`, мы начали в тексте примеров применять в идентификаторах «смешанный регистр». Символы подчёркивания при этом оказываются не нужны, поскольку слова в составе длинного идентификатора отделяются друг от друга как раз большими буквами. Такие идентификаторы в программах на Си++ обычно используют для имён, связанных с нововведениями Си++ — классами и их методами (функциями-членами).

Те сущности, которые не затрагивают отличий Си++ от чистого Си — обычные переменные, функции, которые не являются членами классов и т. п. — по-прежнему обозначают, как и в чистом Си, идентификаторами, состоящими из маленьких букв и подчёркиваний, в некоторых случаях добавляют цифры, ну а макросы — ровно по тем же причинам, что и в Си — именуют полностью большими буквами, чтобы их было хорошо видно. Надо сказать, что макросы в Си++ сильно не в почёте; единственное их использование, от которого невозможно отойти — это защита от повторного включения заголовочных файлов и вообще условная компиляция в широком смысле (см. т. 2, §§4.6.6, 4.7.4), во всех же остальных случаях можно (и нужно!) использовать шаблоны, константы, inline-функции и прочие возможности, не завязанные на макропроцессор. Даже для обозначения нулевого указателя, как уже говорилось в самом начале, рекомендуется использовать арифметический `0`, а не макрос `NULL`, как в чистом Си.

Остаётся, впрочем, вопрос, *как именно* использовать смешанный регистр, то есть какие конкретно буквы делать заглавными, а какие оставлять строчными. Так, в известной книге Гради Буча [9] в именах *классов* каждое слово начинается с большой буквы, а дальше идут маленькие, например `MyGoodClass`, `StrangeThing`, `Complex` и т. п., тогда как для именования *методов* (функций-членов) используется чуть более сложная нотация: имя метода должно состоять не менее чем из двух слов, причём первое слово пишут полностью маленькими буквами, а каждое следующее — с большой буквы, примерно так: `closeFile`, `isReady`, `doSomethingUseful`, `gameScore`, `getXCoord`, `setColor`... Поля классов и структур (члены-данные) по-прежнему называют маленькими буквами с использованием подчёркивания для разделения слов, тогда как в именах классов и методов подчёркивание не используется вообще.

Естественно, такой стиль — не единственный возможный, к тому же можно с ходу назвать определённые недостатки, связанные с ним: конструкторы и деструкторы класса тоже являются методами, но называться они обязаны так же, как и класс, в результате получается, что не все методы называются единообразно. Кроме того, не всегда удобно нижнее ограничение на количество слов в имени: часто возникает желание назвать метод *одним словом*, но сделать этого нельзя, поскольку хотя бы одна заглавная буква в имени должна присутствовать. Поэтому часто можно встретить программы на Си++, где соглашение «каждое слово с большой буквы» используется как для классов, так и для методов; в наших примерах мы именно так и поступали.

10.9.2. Форматирование заголовков классов

При оформлении классов (а также структур, имеющих закрытую часть, если у вас такие предусмотрены) мы сталкиваемся с дополнительной сложностью из-за директив `public:`, `private:` и `protected:`. Вариантов с ними ровно два: либо сдвигать их на дополнительный размер отступа, либо не сдвигать. Вообще говоря, оба следующих фрагмента вполне корректны и допустимы:

```
class MyInteger {
        int x;
    public:
        MyInteger(int ax = 0);
        operator int() const;
};
```

```
class MyInteger {
    int x;
public:
    MyInteger(int ax = 0);
    operator int() const;
};
```

Тем не менее, первый вариант обычно встречается только в программах начинающих, а программисты более опытные используют второй вариант, не предполагающий дополнительного сдвига. В качестве одной из проблем первого варианта можно назвать то, что описания, идущие в начале заголовка класса, т. е. сразу после открывающей фигурной скобки, оказываются сдвинуты на *два* размера отступа. Начинающие иногда пытаются бороться с этим примерно так:

```
class MyInteger {
    int x;
    public:
        MyInteger(int ax = 0);
        operator int() const;
};
```

— но вот это уже совсем никуда не годится, потому что описания членов класса, которые имеют, очевидно, *одинаковый* ранг вложенности (ведь они вложены в класс и более ни во что), оказываются сдвинуты на *разный* размер сдвига.

10.9.3. Форматирование заголовка конструктора

Заголовок конструктора отличается от заголовка обычной функции наличием списка инициализаторов; если списка инициализаторов нет, то никаких особенностей конструктор по сравнению с другими функциями не имеет. Но вот если этот список есть, возникает, во-первых, вопрос, куда его поместить, и, во-вторых, что делать, если он не влез на одну строку.

Прежде всего отметим, что обычно список инициализаторов сносят на отдельную строку (кроме случая, когда конструктор описывается непосредственно в заголовке класса, но об этом позже), причём чаще всего — вместе с двоеточием. Эстетичнее смотрится вариант со сдвигом этой строки на размер отступа, например:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s), int_field(x)
{
    // here the body comes
}
```

Но можно, в принципе, написать и вот так:

```
MyGoodClass::MyGoodClass(int x, const char *s)
: MyBaseClass(s), int_field(x)
{
    // here the body comes
}
```

Иногда встречается и вот такое форматирование:

```
MyGoodClass::MyGoodClass(int x, const char *s) :
    MyBaseClass(s),
    int_field(x)
{
```

Надо отметить, что с размещением двоеточия возникают наиболее неоднозначные вариации. Можно встретить и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s),
    int_field(x)
{
```

и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
:
    MyBaseClass(s),
    int_field(x)
{
```

и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    :
    MyBaseClass(s),
    int_field(x)
{
```

Сразу хотелось бы предостеречь читателя от одного варианта, который хотя и допустим (кроме случая использования табуляции для отступов), но всё же не слишком правилен, потому что очередная строка оказывается сдвинута на непонятно какой уровень отступа:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s),
      int_field(x)
{
```

Довольно удачным оказывается решение, странновато выглядящее на первый взгляд, когда запятую ставят не *после* инициализатора, а *перед следующим*, что позволяет представить двоеточие и запятые как своего рода *символ начала инициализатора* и расставить их в столбик:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s)
    , int_field(x)
    , another_field(s)
{
```

Отдельного рассмотрения заслуживает случай, когда решено инициализаторы записывать в строчку, но при этом список инициализаторов не умещается на одной строке. Для этого случая можно порекомендовать разорвать строку, по необходимости в нескольких местах; выбрать конкретные места разрыва можно либо из соображений равномерности, либо исходя из некой «группировочной» логики (например, в первой строке записать инициализаторы базовых классов, а инициализаторы полей снести на следующую строку). Нужно только помнить, что инициализаторы должны перечисляться в том же порядке, в котором в классе объявлены соответствующие поля. Расположить такие строки, а также знак двоеточия можно, например, так:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s), int_field(x), first(0), last(0),
    sequence_counter(1), the_collection(0),
    helper(new MyHelperClass(*this))
{
```

10.9.4. Тела функций в заголовке класса

Тело функции-члена класса можно вынести за пределы заголовка класса, но можно написать и в самом заголовке; компилятор попытается (но не обязан) обработать такие функции как инлайновые.

Прежде всего отметим, что пользоваться этой возможностью следует очень осторожно. Важнейшая роль описания класса состоит в том, чтобы при взгляде на него можно было понять, как следует работать с объектами такого класса, то есть каков его набор методов и как ими следует пользоваться. Для этого желательно (если не сказать необходимо), чтобы описание класса оставалось компактным, таким, чтобы охватить его можно было одним взглядом. Ясно, что наличие тел функций-членов отнюдь не способствует лаконичности описания класса. Кроме того, если класс не является внутренним для отдельно взятого модуля, то его заголовок приходится выносить в заголовочный файл, который с помощью директивы `#include` (то есть текстуально) будет включаться в другие модули; как следствие, объектный код всех функций-членов, тела которых оставлены в заголовке класса, будет присутствовать в каждом из таких модулей, что далеко не лучшим образом сказывается как на размере исполняемого файла, так и на скорости компиляции проекта. Всё это позволяет сформулировать довольно простое правило: **в заголовке класса могут присутствовать тела функций-членов, весь код которых состоит из одной строки, реже — из двух, в исключительных случаях — из трёх строк**; описания функций-членов бóльшего объёма следует выносить за пределы описания класса.

Остаётся вопрос, как оформить тело функции-члена, если его всё же решено оставить в заголовке класса. Лаконичность описания класса как целого — это самостоятельная цель, ради достижения которой имеет смысл пожертвовать некоторыми другими принципами. Так, если тело функции состоит из одной строки, то часто такое тело записывают на одной строке с её заголовком (особенно часто такое встречается для так называемых аксессоров):

```
class Picture {
    // ...
    int color;
public:
    // ...
    int GetColor() const { return color; }
    void SetColor(int col) { color = col; }
    // ...
};
```

Более того, если функция-член состоит из более чем одного оператора, но все эти операторы способны уместиться на одной строке, программисты часто идут и на это, хотя в обычных условиях так писать ни в коем случае не следует:

```
class FileStream {
    // ...
    void SetFd(int f) { if (fd != -1) close(fd); fd = f; }
    // ...
};
```

Если тело никак не желает помещаться в одну строку с заголовком, его можно целиком снести на следующую строку, по-прежнему записав одной строчкой. В этом случае открывающую фигурную скобку обычно сдвигают на размер отступа, хотя это, строго говоря, не обязательно:

```
class Table {
    // ...
    const char *GetNameById(int id) const
        { return ProvideRecordById(id)->GetName(); }
    // ...
};
```

Если всё-таки вы решили, что тело вашей функции будет лучше смотреться, будучи расписанным на несколько строк, то остаётся ещё возможность сэкономить одну строку, разместив открывающую фигурную скобку на одной строке с заголовком функции (это тоже специфика ситуации тела функции в заголовке класса; в иных случаях так не делают):

```
class FileStream {
    // ...
    void SetFd(int f) {
        if (fd != -1)
           close(fd);
        fd = f;
    }
    // ...
};
```

Чего точно не следует делать, так это приносить лаконичность заголовка класса в жертву стилистическим канонам. Если тело метода занимает несколько строк и «комкать» его не хочется, и тем более если оно при этом становится непонятным, будет намного правильнее всё-таки убрать такое тело из заголовка класса. При этом вы всё ещё можете объяснить компилятору, что ваш метод желательно сделать инлайновым, поставив слово `inline` перед его описанием, как перед описанием обычной inline-функции.

10.10. Пример: TCP-сервер

Объектно-ориентированное программирование можно успешно применить практически к любой сколько-нибудь сложной задаче, но есть одно своеобразное «слово-триггер», появление которого ясно обозначает, что пришло время для ООП. Это слово — «*событие*»; если кто-то заговорил о событиях при обсуждении будущей программы, то, скорее всего, здесь стоит попробовать объекты.

Основных областей, эксплуатирующих события, две: имитационное моделирование вроде упоминавшейся в §10.6.1 модели дорожного дви-

жения и хорошо знакомое нам по второму тому (по TCP-серверам) ***событийно-управляемое программирование***; этот вариант мы и выберем для демонстрации ООП, как говорится, в действии. В качестве такой демонстрации мы напишем простенький TCP-сервер, реализующий *чат*: пользователям, подключившимся к серверу с помощью программы `telnet`, будет сначала предлагаться ввести своё имя, а затем все строки, которые пользователь передаёт серверу, будут пересылаться всем пользователям, подключённым к серверу, с указанием имени того, кто эту строку прислал.

Чтобы сократить объём примера, мы не станем реализовывывать никакие команды вроде «изменить своё имя», «узнать, кто сейчас в чате», «выйти из чата» и т. п., так что наш чат окажется совсем неудобным в использовании. Нам сейчас важнее показать общий подход к построению похожих программ с использованием ООП, ограничившись по возможности меньшим количеством программного кода. Совершенно не сложно сделать чат удобнее для пользователей, предусмотрев, например, строки, начинающиеся с символа «/», которые сервер никому не пересылает, а рассматривает как команды; если угодно, сделайте это сами.

Программу мы построим по событийно-управляемому принципу с использованием системного вызова `select` (см. т. 2, §6.4.4), но, в отличие от примеров, которые мы писали во втором томе на чистом Си, здесь за построение главного цикла (в том числе за организацию ***выборки событий***) будет отвечать созданный специально для этого объект. «Получателями» событий, обнаруженных в ходе выборки, будут другие объекты; для их создания мы тоже опишем специальный класс.

В нашем примере мы ограничимся выборкой и обработкой только двух видов событий — готовности дескриптора к чтению и к записи, причём воспользуемся в итоге только готовностью к чтению. Для обработки обоих видов событий будет использоваться один и тот же класс, в объекте которого содержится файловый дескриптор.

Если бы нам потребовалось обрабатывать другие события, которые можно обнаружить с помощью вызова `select` — истечение заданного временно́го интервала (или наступление указанного момента времени) и/или поступление сигнала — нам для этого пришлось бы ввести другие классы, а поскольку эти два вида событий существенно отличаются по своей природе от событий готовности дескрипторов, взаимодействие объекта, отвечающего за главный цикл, с соответствующими объектами, обрабатывающими события, выглядело бы совсем не так, как взаимодействие с обработчиком готовности дескриптора. Относительно простым примером объектно-ориентированной библиотеки, обрабатывающей все возможные события, обнаруживаемые `select`'ом, может служить написанная автором книги библиотека Sue, домашняя страничка которой расположена по адресу `http://www.croco.net/software/sue/`. Объём её исходного кода можно считать мизерным, если сравнивать с некоторыми другими библиотеками классов, но для примера, приводимого в печатной книге, он всё-таки слишком велик.

Наш чат мы реализуем в два слоя. Первый слой будет состоять из класса `FdHandler`, объекты которого содержат файловый дескриптор и обрабатывают события его готовности, и класса `EventSelector`, единственный объект которого хранит указатели на все активные объекты класса `FdHandler`, а специально предназначенный для этого метод `Run` содержит главный цикл.

Второй слой будет состоять из классов, отражающих понятия «сервера» и «сеанса работы». В самом первом приближении сервер — это слушающий сокет, а сеанс работы — сокет, связанный с клиентом (полученный из вызова `accept`); обе сущности заинтересованы в событиях готовности файлового дескриптора, так что оба класса второго слоя будут наследниками класса `FdHandler`. Конечно, и сеанс, и сервер — это несколько больше, чем просто файловый дескриптор; в частности, для нашей задачи — чат-сервера — объекты сеансов должны взаимодействовать друг с другом, передавая реплики каждого пользователя всем остальным клиентам, подключённым к серверу. Всё это будет реализовано в методах классов сервера и сеансов, но об этом позже.

Начнём с описания класса `FdHandler`. Как обычно в таких случаях, постараемся забыть про разнообразные аспекты стоящей перед нами задачи во всём её великолепии и сосредоточимся на тех частных проблемах, с которыми должен справляться этот *отдельно взятый* класс. Предназначение его объектов — реагировать на события, связанные с файловым дескриптором. Ясно, что такая «реакция ради реакции» никому не нужна, то есть в действительности реагировать на события будет объект какого-то другого класса, решающий ту или иную реально вставшую задачу; по-видимому, такой класс будет *унаследован* от `FdHandler`.

О событиях объекты нашего класса будет оповещать некто (мы знаем, что это будет объект класса `EventSelector`, но пока не будем это знание применять), умеющий о таких событиях узнавать. Ему для этого потребуется знать, за какой файловый дескриптор отвечает наш объект и какие события ему интересны — готовность к чтению, готовность к записи или оба. Очевидно, что с дескриптором также будут работать и методы класса-наследника, не просто же так этот дескриптор существует и к тому же порождает события — ему соответствует поток ввода-вывода, из которого нужно читать данные, в который нужно записывать данные и т. д. Как следствие, в объекте класса `FdHandler` должен по меньшей мере храниться этот самый дескриптор, и его значение должно быть доступно как для других объектов (хотя бы для того, который делает выборку событий), так и для методов наследника. В то же время *изменять* этот дескриптор, что называется, *на лету* никому позволять не следует, поскольку вряд ли к такому изменению будет готов объект, реализующий главный цикл (представьте себе, что в главном цикле обнаружено событие, связанное с дескриптором `5`, а объект, который только что вроде бы за него отвечал, уже содержит дескриптор `17`, и события на пятом дескрипторе его больше не волнуют; корректно выйти из такой ситуации будет непросто). Поэтому передавать объекту дескриптор мы будем через параметр конструктора, а для доступа к нему предусмотрим метод, возвращающий значение дескриптора, но метода для его изменения у нас не будет.

Остаётся вопрос, считать ли дескриптор *собственностью* объекта. От этого зависит, следует ли закрывать этот дескриптор из деструктора объекта при его ликвидации. Если задействовать фантазию, можно представить себе и такие способы использования объекта, при которых правильнее будет считать объект единоличным хозяином дескриптора, и такие, при которых, наоборот, поток ввода-вывода объекту предоставлен во временное пользование и мнить себя хозяином дескриптора ему точно не следует. Мы предусмотрим *оба* варианта, введя для этого специальное булевское поле, показывающее, владеет объект дескриптором или нет.

Заинтересованность объекта в каждом из двух видов готовности дескриптора — в общем случае сущность динамическая. Готовность к чтению чаще всего или не интересна вообще (если поток открыт только на вывод), или нужна всегда; но готовность к записи, если она вообще проверяется, интересна лишь тогда, когда в распоряжении объекта имеются данные, подлежащие отправке, но пока не отправленные. Чтобы не создавать лишних рамок, предусмотрим для индикации заинтересованности объекта в готовности к записи и чтению две виртуальные функции, которые так и назовём `WantRead` и `WantWrite`. По умолчанию первая будет возвращать истину (поскольку чаще всего, если дескриптор вообще передаётся `select`'у, готовность к чтению для него отслеживать требуется), а вторая за неимением лучшего будет возвращать ложь; класс-потомок сможет переопределить эти функции в соответствии со своими потребностями, на то они и виртуальные.

Что касается собственно извещения о наступившем событии, то для него мы предусмотрим функцию `Handle` с двумя логическими параметрами, соответствующими готовности к чтению и записи; функция будет чисто виртуальной, поскольку мы никак не можем знать, что будет делать с дескриптором тот, кому реально требуется информация о наступлении готовности. Это, как мы помним, сделает класс абстрактным — но так и должно быть, ведь реальной функциональности в нём нет, только заготовка. Заголовок нашего класса получается таким:

```
class FdHandler {
    int fd;
    bool own_fd;
public:
    FdHandler(int a_fd, bool own)
        : fd(a_fd), own_fd(own) {}
    virtual ~FdHandler();
    virtual void Handle(bool r, bool w) = 0;
    int GetFd() const { return fd; }
    virtual bool WantRead() const { return true; }
    virtual bool WantWrite() const { return false; }
};
```

Как видим, единственный метод, для которого в заголовке отсутствует тело — это деструктор; остальные методы настолько тривиальны, что выносить их тела за пределы заголовка не хочется. Впрочем, деструктор тоже получается довольно простой, вот он:

```
FdHandler::~FdHandler()
{
    if(own_fd)
        close(fd);
}
```

Займёмся теперь вторым классом нижнего слоя нашей реализации — классом `EventSelector`. В нём должны быть методы для добавления и удаления объектов типа `FdHandler` (точнее — указателей на них); эти методы мы так и назовём `Add` и `Remove`. Хранить адреса зарегистрированных объектов будем в массиве, индексируемом по номерам дескрипторов, чтобы не тратить время на поиск нужного объекта; память под этот массив изначально вообще не будем выделять, сделаем это только при регистрации первого объекта. Конечно, придётся помнить, на какое количество элементов выделена память, чтобы при необходимости увеличить размер массива. Кроме того, чтобы каждый раз не вычислять максимальный дескриптор (что требуется при обращении к вызову `select`), будем хранить это значение в поле объекта; заодно это позволит не просматривать весь массив, когда его размер существенно больше нужного. Значения полей, предназначенных для хранения длины массива и максимального дескриптора, будем считать невалидными, если массива нет (т. е. когда указатель на его начало всё ещё нулевой).

В событийно-управляемых программах часто возникает ситуация, когда изнутри обработчика события нужно прекратить выполнение всей программы. Можно, конечно, вызвать `exit`, но намного корректнее в такой ситуации просто сообщить главному циклу, что ему пора завершаться. Для этого предусмотрим флажок `quit_flag`, который каждый раз перед запуском главного цикла будет сбрасываться; метод `BreakLoop` будет этот флаг взводить, а цикл в этом случае будет завершаться. Таким образом каждый, кому известен адрес объекта `EventSelector`, сможет заставить уже запущенный главный цикл остановиться.

Заголовок класса у нас получается такой:

```
class EventSelector {
    FdHandler **fd_array;
    int fd_array_len;
    int max_fd;
    bool quit_flag;
public:
    EventSelector() : fd_array(0), quit_flag(false) {}
    ~EventSelector();
    void Add(FdHandler *h);
    bool Remove(FdHandler *h);
    void BreakLoop() { quit_flag = true; }
    void Run();
};
```

Напишем тела оставшихся методов. Проще всего с деструктором — нужно только удалить массив указателей, если он есть:

```
EventSelector::~EventSelector()
{
    if(fd_array)
        delete[] fd_array;
}
```

В методе `Add` нам придётся обработать два особых случая: когда массива ещё нет и когда он есть, но в нём нет элемента с нужным номером. При создании массива, если дескриптор не превышает `15`, будем отводить память под 16 элементов; поскольку эти числа ни на что не влияют за пределами данного конкретного метода, не будем заморачиваться именованными константами (в самом деле, если мы решим принять какой-то другой начальный размер, за пределами тела метода `Add` в коде ничего менять не придётся). Текст метода получается таким:

```
void EventSelector::Add(FdHandler *h)
{
    int i;
    int fd = h->GetFd();
    if(!fd_array) {
        fd_array_len = fd > 15 ? fd + 1 : 16;
        fd_array = new FdHandler*[fd_array_len];
        for(i = 0; i < fd_array_len; i++)
            fd_array[i] = 0;
        max_fd = -1;
    }
    if(fd_array_len <= fd) {
        FdHandler **tmp = new FdHandler*[fd+1];
        for(i = 0; i <= fd; i++)
            tmp[i] = i < fd_array_len ? fd_array[i] : 0;
        fd_array_len = fd + 1;
        delete[] fd_array;
        fd_array = tmp;
    }
    if(fd > max_fd)
        max_fd = fd;
    fd_array[fd] = h;
}
```

Метод `Remove` оказывается несколько проще. Единственный нетривиальный момент в нём — необходимость поддержания корректного значения поля `max_fd`; если был удалён дескриптор, имевший максимальное значение, то нужно найти, какой дескриптор будет максимальным после такого удаления, для чего массив следует просмотреть справа налево до первого непустого элемента. Код будет таким:

```
bool EventSelector::Remove(FdHandler *h)
{
    int fd = h->GetFd();
    if(fd >= fd_array_len || fd_array[fd] != h)
        return false;
    fd_array[fd] = 0;
    if(fd == max_fd) {
        while(max_fd >= 0 && !fd_array[max_fd])
            max_fd--;
    }
    return true;
}
```

Остался самый длинный из наших методов — `Run`. Подробно комментировать его мы не будем, построению главного цикла на основе вызова `select` было уделено достаточно внимания во втором томе (см. §6.4.4). Просто покажем, что у нас получилось:

```
void EventSelector::Run()
{
    quit_flag = false;
    do {
        int i;
        fd_set rds, wrs;
        FD_ZERO(&rds);
        FD_ZERO(&wrs);
        for(i = 0; i <= max_fd; i++) {
            if(fd_array[i]) {
                if(fd_array[i]->WantRead())
                    FD_SET(i, &rds);
                if(fd_array[i]->WantWrite())
                    FD_SET(i, &wrs);
            }
        }
        int res = select(max_fd+1, &rds, &wrs, 0, 0);
        if(res < 0) {
            if(errno == EINTR)
                continue;
            else
                break;
        }
        if(res > 0) {
            for(i = 0; i <= max_fd; i++) {
                if(!fd_array[i])
                    continue;
                bool r = FD_ISSET(i, &rds);
                bool w = FD_ISSET(i, &wrs);
                if(r || w)
                    fd_array[i]->Handle(r, w);
            }
        }
    } while(!quit_flag);
}
```

Нижний слой мы завершили, можно приступать к реализации верхнего. Здесь нам потребуется два класса — `ChatServer` и `ChatSession`. Объект класса `ChatServer` будет обслуживать слушающий сокет и при наступлении на нём готовности к чтению, приняв соединение вызовом `accept`, создавать объект класса `ChatSession`. Чтобы сделать возможной рассылку сообщения всем подключённым клиентам, `ChatServer` будет хранить список всех порождённых им (и всё ещё существующих) объектов типа `ChatSession`. Удалять эти объекты тоже будут только методы класса `ChatServer`; в частности, получив на своём дескрипторе ситуацию «конец файла», объект сеанса будет обращаться к своему объекту сервера с просьбой удалить его.

Вообще можно заметить, что *всю* работу с объектами сеансов (объектами класса `ChatSession`) будут вести только методы сервера. Именно сервер и порождает, и удаляет сеансы, и сообщает им, что нужно что-то отправить клиенту, а больше, собственно говоря, с объектами сеансов делать ничего и не надо. Конечно, объект класса `EventSelector` будет сообщать объектам сеансов о готовности их дескрипторов к чтению, но с точки зрения `EventSelector` это будут объекты класса `FdHandler`, он других не знает. Как следствие, можно всё содержимое класса `ChatSession` сделать приватным, а класс `ChatServer` объявить его другом.

Нетривиальное решение с применением защиты напрашивается и для класса сервера. Поскольку он унаследован от `FdHandler`, для создания его объекта нужно *уже* иметь готовый файловый дескриптор. Между тем процедуру подготовки сервера к работе, которая включает обращения к вызовам `socket`, `bind` и `listen`, хотелось бы всё-таки доверить классу сервера. Благодаря статическим методам это возможно: конструктор класса мы сделаем приватным, а подготовку и последующее создание объекта будет проводить статическая функция `Start`; если сервер запустить не удалось, она будет возвращать нулевой адрес, если же всё прошло успешно — адрес созданного объекта класса `ChatServer`.

Объект сеанса будет хранить (в виде строки) имя, которым пришедший клиент представился. Для этого в классе предусмотрим поле `name`, изначально содержащее нулевой адрес; при получении строки от клиента объект будет считать эту строку именем, если в поле `name` всё ещё ноль, в противном случае строка будет рассматриваться как реплика в чате и рассылаться всем клиентам. После получения имени объект будет отсылать своему клиенту приветственное сообщение, а всем остальным — сообщение о том, что в чат вошёл новый участник; при выходе участника из чата всем (кроме, естественно, ушедшего участника) будет рассылаться сообщение о его уходе.

Для рассылки сообщения всем клиентам (или всем, кроме одного) в классе сервера будет предусмотрен метод `SendAll` с двумя параметрами: первый — строка, которую нужно разослать, второй — адрес объекта сеанса, клиенту которого эту строку посылать не надо (если этот параметр оставить нулевым, сообщение рассылается всем). Для отправки сообщения одному клиенту класс сеанса будет снабжён методом `Send`.

Сразу же предусмотрим две константы, задающие максимально допустимую длину строки (строки большей длины будут игнорироваться) и размер очереди непринятых запросов на соединение (второй параметр вызова `listen`):

```
enum {
    max_line_length = 1023,
    qlen_for_listen = 16
};
```

Чтобы метод `Handle` в классе сеанса не распухал, вынесем часть его функциональности в подпрограммы, которые назовём `ReadAndIgnore` (выполнить очередное чтение, прочитанное игнорировать до получения первого символа перевода строки), `ReadAndCheck` (выполнить очередное чтение и обработать полученные строки, если они есть), `CheckLines` (проверить, не пришла ли целиком строка, обработать все такие строки по одной, оставив в буфере только начало незаконченной строки, если такое есть) и `ProcessLine` (обработать одну строку). С учётом этого заголовки классов верхнего слоя получаются такими:

```
class ChatSession : FdHandler {
    friend class ChatServer;
    char buffer[max_line_length+1];
    int buf_used;
    bool ignoring;
    char *name;
    ChatServer *the_master;

    ChatSession(ChatServer *a_master, int fd);
    ~ChatSession();
    void Send(const char *msg);
    virtual void Handle(bool r, bool w);
    void ReadAndIgnore();
    void ReadAndCheck();
    void CheckLines();
    void ProcessLine(const char *str);
};

class ChatServer : public FdHandler {
    EventSelector *the_selector;
    struct item {
        ChatSession *s;
        item *next;
    };
    item *first;
    ChatServer(EventSelector *sel, int fd);
public:
    ~ChatServer();
    static ChatServer *Start(EventSelector *sel, int port);
    void RemoveSession(ChatSession *s);
    void SendAll(const char *msg, ChatSession *except = 0);
private:
    virtual void Handle(bool r, bool w);
};
```

Начнём с реализации методов сервера. Функция `Start` работает без объекта — она его создаёт, если всё в порядке. Последовательность действий по запуску сервера не комментируем, она подробно рассмотрена ранее (см. т. 2, §6.3.4). Текст функции вот:

```
ChatServer *ChatServer::Start(EventSelector *sel, int port)
{
    int ls, opt, res;
    struct sockaddr_in addr;
    ls = socket(AF_INET, SOCK_STREAM, 0);
    if(ls == -1)
        return 0;
    opt = 1;
    setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(port);
    res = bind(ls, (struct sockaddr*) &addr, sizeof(addr));
    if(res == -1) {
        close(ls);
        return 0;
    }
    listen(ls, qlen_for_listen);
    return new ChatServer(sel, ls);
}
```

С конструктором и деструктором всё сравнительно просто:

```
ChatServer::ChatServer(EventSelector *sel, int fd)
    : FdHandler(fd, true), the_selector(sel), first(0)
{
    the_selector->Add(this);
}
ChatServer::~ChatServer()
{
    while(first) {
        item *tmp = first;
        first = first->next;
        the_selector->Remove(tmp->s);
        delete tmp->s;
        delete tmp;
    }
    the_selector->Remove(this);
}
```

При получении сообщения о готовности слушающего сокета нужно принять соединение вызовом `accept`, создать и зарегистрировать объект сеанса для этого соединения. Всё это делается в функции `Handle`:

```
void ChatServer::Handle(bool r, bool w)
{
    if(!r)       // must not happen
        return;
    int sd;
    struct sockaddr_in addr;
    socklen_t len = sizeof(addr);
    sd = accept(GetFd(), (struct sockaddr*) &addr, &len);
    if(sd == -1)
        return;
    item *p = new item;
    p->next = first;
    p->s = new ChatSession(this, sd);
    first = p;
    the_selector->Add(p->s);
}
```

В нашей упрощённой реализации адрес пришедшего клиента, записанный в параметр `addr`, никак не используется, но в реальном сервере его стоило бы записать в файл журнализации и, возможно, запомнить в объекте сеанса, чтобы, к примеру, администратор сервера мог посмотреть, из какой сети пришёл тот или иной участник чата.

Остаются ещё методы `RemoveSession` и `SendAll`, их реализация в целом очевидна:

```
void ChatServer::RemoveSession(ChatSession *s)
{
    the_selector->Remove(s);
    item **p;
    for(p = &first; *p; p = &((*p)->next)) {
        if((*p)->s == s) {
            item *tmp = *p;
            *p = tmp->next;
            delete tmp->s;
            delete tmp;
            return;
        }
    }
}

void ChatServer::SendAll(const char *msg, ChatSession *except)
{
    item *p;
    for(p = first; p; p = p->next)
        if(p->s != except)
            p->s->Send(msg);
}
```

Реализуем теперь методы объекта сеанса. Самый простой здесь, видимо, метод `Send`:

```
void ChatSession::Send(const char *msg)
{
    write(GetFd(), msg, strlen(msg));
}
```

Конструктор класса воспользуется этим методом, чтобы спросить у пользователя его имя:

```
ChatSession::ChatSession(ChatServer *a_master, int fd)
    : FdHandler(fd, true), buf_used(0), ignoring(false),
    name(0), the_master(a_master)
{
    Send("Your name please: ");
}
```

Всё, что нужно удалить в деструкторе — это строку имени пользователя (если, конечно, она успела сформироваться, что происходит не всегда, ведь пользователь мог прервать сеанс, так и не представившись). Обратите внимание, что в конструкторе мы передали конструктору базового класса вторым параметром значение `true`, так что объект `FdHandler` считает себя владельцем дескриптора — и, следовательно, закроет его из своего деструктора, нам сейчас об этом можно не беспокоиться. Деструктор получается таким:

```
ChatSession::~ChatSession()
{
    if(name)
        delete[] name;
}
```

Остаётся реализовать реакцию на готовность сокета к чтению. Здесь вроде бы нет ничего сложного, но код получается громоздкий. Чтобы знать, сколько памяти выделять под каждое посылаемое сообщение, опишем используемые строки в виде константных массивов:

```
static const char welcome_msg[] =
    "Welcome to the chat, you are known as ";
static const char entered_msg[] = " has entered the chat";
static const char left_msg[] = " has left the chat";
```

Выше мы уже писали, за что отвечает каждая из получившихся подпрограмм, так что здесь просто приведём их код:

```
void ChatSession::Handle(bool r, bool w)
{
    if(!r)   // should never happen
        return;
    if(buf_used >= (int)sizeof(buffer)) {
        buf_used = 0;
        ignoring = true;
    }
    if(ignoring)
        ReadAndIgnore();
    else
        ReadAndCheck();
}
void ChatSession::ReadAndIgnore()
{
    int rc = read(GetFd(), buffer, sizeof(buffer));
    if(rc < 1) {
        the_master->RemoveSession(this);
        return;
    }
    int i;
    for(i = 0; i < rc; i++) {
        if(buffer[i] == '\n') {   // stop ignoring!
            int rest = rc - i - 1;
            if(rest > 0)
                memmove(buffer, buffer + i + 1, rest);
            buf_used = rest;
            ignoring = false;
            CheckLines();
            return;
        }
    }
}
void ChatSession::ReadAndCheck()
{
    int rc =
        read(GetFd(), buffer+buf_used, sizeof(buffer)-buf_used);
    if(rc < 1) {
        if(name) {
            int len = strlen(name);
            char *lmsg = new char[len + sizeof(left_msg) + 2];
            sprintf(lmsg, "%s%s\n", name, left_msg);
            the_master->SendAll(lmsg, this);
            delete[] lmsg;
        }
        the_master->RemoveSession(this);
        return;
    }
    buf_used += rc;
    CheckLines();
}
void ChatSession::CheckLines()
{
```

```
        if(buf_used <= 0)
            return;
        int i;
        for(i = 0; i < buf_used; i++) {
            if(buffer[i] == '\n') {
                buffer[i] = 0;
                if(i > 0 && buffer[i-1] == '\r')
                    buffer[i-1] = 0;
                ProcessLine(buffer);
                int rest = buf_used - i - 1;
                memmove(buffer, buffer + i + 1, rest);
                buf_used = rest;
                CheckLines();
                return;
            }
        }
    }
    void ChatSession::ProcessLine(const char *str)
    {
        int len = strlen(str);
        if(!name) {
            name = new char[len+1];
            strcpy(name, str);
            char *wmsg = new char[len + sizeof(welcome_msg) + 2];
            sprintf(wmsg, "%s%s\n", welcome_msg, name);
            Send(wmsg);
            delete[] wmsg;
            char *emsg = new char[len + sizeof(entered_msg) + 2];
            sprintf(emsg, "%s%s\n", name, entered_msg);
            the_master->SendAll(emsg, this);
            delete[] emsg;
            return;
        }
        int nl = strlen(name);
        char *msg = new char[nl + len + 5];
        sprintf(msg, "<%s> %s\n", name, str);
        the_master->SendAll(msg);
        delete[] msg;
    }
```

После этого нам остаётся только написать функцию `main`, которая получается довольно компактной. Для экономии места мы не будем обрабатывать параметры командной строки, единственную «неизвестную величину» — номер порта — приколотим к коду, как шутят в таких случаях программисты, ржавыми гвоздями:

```
    static int port = 7777;
    int main()
    {
        EventSelector *selector = new EventSelector;
        ChatServer *serv = ChatServer::Start(selector, port);
        if(!serv) {
            perror("server");
            return 1;
        }
        selector->Run();
        return 0;
    }
```

Полностью эта программа приведена в архиве примеров в поддиректории `cpp_chat`; там она разбита на три модуля и снабжена файлом для `make`.

10.11. О графических интерфейсах пользователя

В нынешних условиях конечные пользователи обычно ожидают от любой программы наличия графического интерфейса пользователя (*graphical user interface, GUI*), а командную строку как пользовательский интерфейс воспринимать категорически отказываются; таков печальный итог пропагандистской деятельности софтверных гигантов. Конечно, «интуитивная понятность» графических интерфейсов — не более чем миф, их, как и любые способы работы с компьютером, приходится *изучать*, а работа с ними, как мы знаем, неэффективна, не поддаётся автоматизации и в целом контрпродуктивна; но мы вряд ли сможем объяснить это всему миру, прочно *подсевшему* на GUI, и даже большинству специалистов в области IT — которым, казалось бы, всё должно быть ясно и без объяснений.

Так или иначе, программист, если он чего-то хочет добиться, просто обязан уметь снабжать свои программы графическим интерфейсом, когда возникает такая необходимость. В действительности создавать программы с графическим интерфейсом совсем не сложно, но мы сознательно откладывали разговор о GUI до подходящего момента. Можно считать, что этот момент настал: мы теперь знакомы с событийно-ориентированной моделью программной архитектуры, знаем язык Си++ и, что ещё важнее, умеем использовать объектно-ориентированное программирование.

Между прочим, если заявить, что GUI и ООП «созданы друг для друга», мы окажемся недалеки от истины, причём в самом буквальном смысле слова. В качестве первого в истории объектно-ориентированного языка часто называют Smalltalk (забывая при этом про язык Simula, из которого Smalltalk заимствовал многие идеи, включая наследование); но основной целью проекта Smalltalk был не новый язык программирования, а *первый в истории оконный графический интерфейс пользователя*. Именно в этом проекте появились — практически в современном их виде — окна с заголовками, которые можно перемещать по экрану и менять их размеры, а также привычные ныне элементы управления — ***виджеты*** (англ. *widgets*) вроде кнопок, «галочек» (*checkbuttons*), «переключателей» (*radiobuttons*) и т. п.

Теоретически возможно построить пользовательский интерфейс со всеми его элементами, обращаясь напрямую к графической подсистеме, в нашем случае X Window (см. т. 1, §1.2.18, т. 2, §5.1.6). В самом деле, никто вроде бы не мешает установить соединение с X-сервером, попросить его нарисовать окно, в окне изобразить все нужные кнопочки, надписи и прочие элементы интерфейса, затем организовать обработку сообщений о действиях пользователя, таких как перемещение мыши и нажатия на её кнопки, нажатия клавиш на клавиатуре и прочее; при этом, например, чтобы создать у пользователя впечатление «нажатия кнопки», придётся сообразить, что изображение кнопки в нажатом положении отличается от исходного расположением тени вдоль границы изображения, так что, когда пользователь щёлкнет мышкой по нарисованной кнопке, нужно будет эту тень соответствующим образом перерисовать, и аналогичным образом поступать в нужные моменты со всеми остальными элементами интерфейса. Кроме того, довольно интересные приключения поджидают нас при отображении надписей — здесь нам придётся озаботиться выбором шрифта, его размера и прочих свойств; к счастью, X Window хотя бы умеет отображать надпись заданным шрифтом, иначе пришлось бы самим читать файлы шрифтов, извлекать оттуда изображения символов и соображать, как наложить их на имеющуюся картинку. Пожалуй, в таком режиме даже самое простенькое диалоговое окно отняло бы несколько дней напряжённого программирования.

К счастью, вся эта кропотливая работа практически одинакова для всех программ, поддерживающих GUI, и, разумеется, за нас её уже давно сделали; нам остаётся только выбрать подходящую ***библиотеку виджетов***. Такие библиотеки сами устанавливают связь с X-сервером (или общаются с другой графической подсистемой; практически все популярные библиотеки виджетов переносятся между X Window, MS Windows и Mac OS X, многие также допускают работу под Android и на других платформах), сами как надо рисуют элементы управления (те самые виджеты), сами реализуют реакцию на бóльшую часть действий пользователя — нам остаётся лишь указать, какие виджеты мы будем использовать и как их расположить; дальше мы самостоятельно запрограммируем реакцию на наиболее существенные события (например, нет нужды реагировать на каждое изменение текста в поле ввода или установку и сбрасывание галочек, достаточно выполнить некую процедуру, когда пользователь нажмёт наконец на Самую Главную Кнопку) — и программа готова.

10.11.1. Знакомимся с библиотекой FLTK

Библиотек виджетов доступно довольно много, но по тем или иным причинам большинство из них нам на роль учебного пособия не подходят. Впрочем, научившись обращаться с одной такой библиотекой, вы, скорее всего, без особых проблем сможете перейти на любую другую. Библиотека, которую мы рассмотрим, называется FLTK (*Fast Light Toolkit*); это библиотека классов Си++, позволяющая работать со всеми основными видами виджетов и создавать полноценные программы с графическим интерфейсом, но при этом не использующая ничего из тех новомодных аспектов Си++, от рассмотрения которых мы отказались.

На момент подготовки второго издания книги в начале 2021 года актуальной для FLTK оставалась стабильная версия 1.3.5. Интересно, что попытки создания более новых версий — 2.0.* и 3.0.* — кончились в итоге ничем: проект создания FLTK 2.0 тихо загнулся в 2008 году, попытка объединить возможности первой и второй версий — FLTK 3.0 — продолжалась с 2010 до 2012 года и тоже благополучно заглохла. При этом стабильная версия 1.3.* и экспериментальная 1.4.* продолжают активно развиваться: версия 1.3.5, на которую мы будем ориентироваться, вышла в марте 2019 года.

Библиотека входит в большинство дистрибутивов Linux и может быть установлена из репозиториев. В дистрибутивах семейства Debian для установки файлов, необходимых для программирования с использованием FLTK, нужно поставить пакет `libfltk1.3-dev`. Вполне возможно, что при этом будет установлена не самая последняя версия, например, 1.3.4; это не страшно, все наши примеры будут с ней отлично работать. При желании можно скачать исходные тексты библиотеки с её официального сайта `www.fltk.org` и собрать бинарные файлы, следуя инструкции (в простейшем случае достаточно раскрыть архив и дать команду `make` в получившейся директории). В дальнейшем тексте мы предполагаем, что библиотека FLTK установлена в системе, так что компилятор знает, где брать заголовочные файлы и файлы библиотек; если вы решили библиотеку в системные директории не устанавливать — например, оставить её в своей домашней директории — при запуске компилятора придётся объяснить ему, где искать заголовочники и библиотеки, с помощью флагов `-I` и `-L` (см. т. 2, приложение 1; напомним, что `gcc` и `g++` — это физически одна и та же программа).

Чтобы преодолеть барьер вхождения, мы сейчас напишем очень простую программу с использованием FLTK, соберём её и запустим, но для этого нам нужно обсудить основы терминологии. Итак, прежде всего, ***окно*** — это прямоугольная область экрана, описываемая неким объектом графической подсистемы (в нашем случае — системы X Window). Различают ***окна верхнего уровня*** и ***дочерние***. Окно верхнего уровня (*top level window*) выглядит на экране как самостоятельная сущность, обычно снабжается заголовком, рамкой и всевозможными элементами оформления, позволяющими передвигать окно, менять его размер, «вытаскивать» его поверх других окон или, наоборот, «утапливать», чтобы другие окна стали видны; как мы знаем, за все эти элементы оформления в X Window отвечает ***оконный менеджер***. Внутри окна верхнего уровня могут располагаться дочерние окна (*child windows* или просто *children*), причём они тоже могут содержать другие окна (формально говоря, на любую глубину). Надо сказать, что оконные менеджеры работают только с окнами верхнего уровня, дочерние окна они полностью игнорируют, так что их внешний вид полностью определяется породившей их программой. В принципе программа может открыть сколько угодно окон верхнего уровня, и довольно часто программы так и поступают; в частности, ***диалоговые окна*** чаще всего оформляются именно как окна верхнего уровня, хотя и не всегда. Понять, является ли появившееся диалоговое окно окном верхнего уровня, очень просто: если оно снабжено такой же рамкой и прочими элементами оформления, как и другие окна, то, значит, его оформлением занимается оконный менеджер, а он работает, как мы уже отметили, только с окнами верхнего уровня.

Второй важнейший для нас термин — ***виджет*** (*widget*). Как часто бывает в таких случаях, пытаться дать определение этого термина бесполезно — всё равно ничего адекватного не получится. На интуитивном уровне виджет — это некий обособленный элемент графического интерфейса; к числу виджетов относятся всевозможные кнопки, «галочки», надписи, поля ввода текста и т. п. Виджеты могут быть вложены друг в друга; практически любая система предусматривает в том или ином виде виджет «группа», который специально предназначен, чтобы содержать другие виджеты, но бывают и другие случаи — например, более сложный виджет может использовать более простые виджеты как свои составные части. Многие библиотеки виджетов предпочитают каждый виджет оформлять как окно, что позволяет изрядную часть работы по их отрисовке, удалению с экрана, перемещению и т. п. переложить на графическую подсистему. FLTK применяет противоположный подход: окно здесь считается *частным случаем* виджета, при этом в большинстве программ применяются только окна верхнего уровня. Библиотека позволяет создавать дочерние окна, но требуется это редко — например, чтобы использовать возможности библиотеки OpenGL, для которой нужно отдельное окно. Что касается обычных виджетов, то всё, что связано с их внешним видом, позиционированием, наложением друг на друга и т. п., FLTK помнит и обрабатывает самостоятельно; с точки зрения графической подсистемы виджеты FLTK остаются просто фрагментами изображения внутри одного окна, т. е. **виджеты FLTK не являются окнами**.

Программа, которой мы воспользуемся для примера, будет работать очень просто: нарисует окно со словом «`hello`» в заголовке и надписью «`Hello, World!`» в самом окне, после чего дождётся нажатия клавиши Escape и завершится. Чтобы реализовать это с помощью FLTK, нам потребуются объекты двух классов, описанных в библиотеке. Объект класса `Fl_Window` будет отвечать за главное окно (за его размеры и текст заголовка), а для отображения текста в самом окне нам придётся создать соответствующий виджет — объект класса `Fl_Box`. Кроме того, для запуска главного цикла, который библиотека FLTK предпочитает организовывать сама[30], нам нужно будет обратиться к статическому методу класса `Fl`, который объединяет бóльшую часть глобальных сущностей (переменных и функций), вводимых библиотекой.

Каждый из классов `Fl`, `Fl_Window` и `Fl_Box` описан в отдельном заголовочном файле, так что нам придётся подключить их все. Затем мы опишем функцию `main`, в теле которой создадим объекты для главного окна и вложенного в него виджета, прикажем библиотеке показать (вывести на экран) главное окно и запустим главный цикл. Полностью текст программы будет таким:

```
// fltk/hello.cpp
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv)
{
    Fl_Window *win = new Fl_Window(300, 100, "hello");
    Fl_Box *b = new Fl_Box(0, 0, 300, 100, "Hello, World!");
    b->labelsize(36);
    win->end();
    win->show();
    return Fl::run();
}
```

Здесь уместны некоторые пояснения. Параметры конструктора класса `Fl_Window` — размер окна по горизонтали, по вертикали и текст для заголовка окна (последний параметр не обязателен). Класс `Fl_Window` является в FLTK наследником класса `Fl_Group`; в нашем примере это важно, поскольку объекты `Fl_Group` обладают интересным свойством: начиная с момента создания такого объекта все другие создаваемые объекты-виджеты автоматически становятся его подчинёнными (включаются в группу), и продолжается это до тех пор, пока не будет вызван метод `end`. Таким образом, виджет типа `Fl_Box`, который мы создаём следующей строкой нашей программы, будет считаться вложенным в главное окно, что нам, собственно, и требуется.

Параметры конструктора класса `Fl_Box` — координаты верхнего левого угла виджета *относительно окна, в которое входит виджет* (отметим, что здесь имеется в виду именно окно, а не, например, такой виджет типа `Fl_Group`, который окном не является), размеры виджета и текст надписи, ради которой мы, собственно, этот `Box` и создаём. Как видим, наш виджет будет заполнять собой всё пространство главного окна.

В следующей строке для виджета с надписью вызывается метод `labelsize`, который устанавливает размер шрифта в 36 пикселей, чтобы надпись получилась покрупнее.

Далее мы сообщаем объекту главного окна, что больше виджетов в него добавлять не собираемся (метод `end`) и приказываем вывести окно на экран (метод `show`). Именно этот метод установит соединение с X-сервером, поскольку оно ещё не установлено. Последняя строка программы запускает главный цикл обработки событий, вызывая статический метод `run` класса `Fl`; возвращаемое этим методом значение равно нулю, если всё в порядке, и отличается от нуля, если что-то пошло не так; это позволяет использовать результат метода `run` в качестве кода завершения процесса, что мы и делаем, поместив его вызов в тело оператора `return`.

Чтобы откомпилировать и собрать нашу программу, потребуется примерно такая команда:

[30] Позже мы узнаем, как отобрать главный цикл у FLTK и организовать его самим, извещая библиотеку о тех событиях, которые её касаются.

Рис. 10.4. «Hello, World» на FLTK

Рис. 10.5. Окно программы `speak`

```
g++ -Wall -g hello.cpp -lfltk -o hello
```

Запустив программу, как обычно, командой «`./hello`», мы увидим окно, выглядящее как показано[31] на рис. 10.4. Чтобы завершить программу, можно нажать `Ctrl-C` в терминале, где мы её запустили (попросту говоря, *прибить* её), либо нажать Escape в *её собственном* окне. Такую реакцию на Escape библиотека FLTK предусматривает по умолчанию; при желании это можно отменить.

Окно нашей программы обладает одним несколько неожиданным свойством: его можно перемещать по экрану, а вот изменить его размер не получится. Дело тут в том, что единственный виджет в нашем окне имеет жёстко заданный размер и не предусматривает его изменения. К вопросу об изменении размеров окон мы ещё вернёмся.

Программа в таком виде, в котором мы её воспроизвели, имеет один весьма серьёзный недочёт в плане качества кода: константы `100`, `300` и `36` в ней записаны явным образом, без введения имён. Потом мы обязательно исправимся, а сейчас нас больше интересовала наглядность короткого примера.

10.11.2. Простые кнопки и реакция на них

Программа, рассмотренная в предыдущем параграфе, фактически ничего не делала — единственным действием пользователя, на которое она реагировала, было нажатие Escape (для завершения работы), да к тому же эту реакцию на Escape предусмотрели не мы, а авторы FLTK. Единственным виджетом, который мы задействовали, был `Fl_Box`, который относится к ***пассивным*** виджетам или ***виджетам вывода*** (*output widgets*) — таким, с которыми пользователь обычно не может ничего сделать, которые сами по себе не обрабатывают никаких событий. На самом деле их можно заставить это делать — например, если пользователь вам чем-то не угодил и вы поставили себе цель загадать ему ребус; но в нормальных условиях виджеты вывода только отображают ту или иную информацию, которая может оставаться неизменной, а может меняться в ходе работы программы.

В отличие от виджетов вывода, ***виджеты ввода***, иногда называемые также ***активными***[32], предполагают, что пользователь будет с ними что-то делать — тыкать в них мышкой, нажимать клавиши на клавиатуре, на что виджет должен как-то реагировать. Изучение таких виджетов и их взаимодействия мы начнём с самого, пожалуй, примитивного из них — с простой кнопки, нарисованной на экране и реагирующей на клик мыши, если при этом мышиный курсор находился над изображением кнопки.

Между прочим, автор лично видел людей, не понимающих, что «кнопка» на экране на самом деле вовсе не кнопка, что она *нарисована* — в особенности когда речь идёт о сенсорных экранах, где на кнопку нажимают не щелчком мыши, а прямо пальцем. Конечно, читателя нашей книги в подобном подозревать было бы по меньшей мере странно, но речь не об этом. Проектируя свой графический интерфейс, помните, что восприятие пользователя может кардинально отличаться от вашего собственного, притом зачастую трудно представить, *насколько*.

В библиотеке FLTK кнопка представлена классом `Fl_Button`. Как ни странно, параметры конструктора у него точно такие же, как и у знакомого нам по предыдущему параграфу `Fl_Box`: четыре числа, задающие координаты левого верхнего угла, ширину и высоту, и строка — текст, который следует показать. Вообще в FLTK таковы параметры *едва ли не всех* конструкторов классов, представляющих виджеты; исключения вроде `Fl_Window` редки и погоды не делают. Различается разве что способ интерпретации строки, передаваемой пятым параметром.

Как мы понимаем, кнопка — это не просто изображение на экране; её предназначение — *что-то делать, когда пользователь по ней щёлкнет*. Чтобы пользователь окончательно забыл, что имеет дело с *нарисованной* кнопкой, а не настоящей, она при щелчке мышью меняет своё изображение так, чтобы выглядеть нажатой; с этим библиотека справляется сама, без нашего участия. Но ведь и наша программа тоже должна что-то сделать, когда пользователь своими действиями активирует кнопку!

В библиотеке FLTK принят довольно неожиданный подход к заданию реакции на события, в число которых входит и нажатие кнопки: для этого используются функции обратного вызова, больше известные под названием ***callback-функции***, которые мы обсуждали в §9.4.1. Если припомнить, какую терминологию мы там ввели, то в роли подсистемы-сервера у нас окажется библиотека FLTK, а в роли клиента — головная программа, написанная нами. Callback-функция должна иметь такой вид:

```
void callback_func(Fl_Widget *w, void *user)
{
    // ...
}
```

Отметим, что класс `Fl_Widget` в библиотеке FLTK является базовым для всех классов, описывающих виджеты, так что, каков бы ни был ваш виджет, его адрес можно будет в такую функцию передать первым параметром (помните про преобразование адресов по закону полиморфизма!)

Создав и настроив объект кнопки (объект класса `Fl_Button`), мы сообщаем ему адрес callback-функции и значение для её параметра `user` (последнее позволяет использовать одну функцию для обработки разных событий). Когда пользователь активирует кнопку на экране, объект `Fl_Button` вызовет нашу callback-функцию, передав ей первым параметром адрес объекта-кнопки, а вторым — то значение, которое мы указали при регистрации callback'а.

Для иллюстрации рассмотрим сугубо искусственную программу, которая по нажатию кнопки будет выдавать некие строки в свой стандартный поток вывода — то есть мы увидим эти строки в том терминале, из которого запустили программу. Обычно так не делают; предполагается, что программа, имеющая графический интерфейс, может быть запущена так, что пользователь не будет видеть её стандартные потоки (например, через меню или иконочные файловые менеджеры), и поэтому всю информацию, предназначенную для пользователя, она должна выдавать через свой графический интерфейс. Сделать это нетрудно, но здесь и сейчас нам важнее проиллюстрировать технику обработки нажатий на нарисованные кнопки *на как можно более короткой программе*, а вывод в стандартный поток (вместо использования для этого специального виджета) позволит сэкономить десяток строк длины примера и добрую страницу пояснений.

Программа, которую мы напишем, отобразит в своём окне три кнопки с надписями «`Say hello`», «`Say goodbye`» и «`Quit`» (см. рис. 10.5). По нажатию на первые две из них программа будет выдавать в стандартный поток вывода соответственно фразы «`Hello, world!`» и «`Goodbye, world!`», а по нажатию на третью — завершаться. Для этого нам потребуются две callback-функции: одна будет печатать сообщение, адрес которого передан через параметр `user`, а вторая — завершать программу, игнорируя свои параметры. Выглядеть они будут так:

```
static void say_callback(Fl_Widget *, void *user)
{
```

[31] Помните, что в X Window за рисование рамки и заголовка окна отвечает не сама оконная программа, а оконный менеджер. Автор книги использует fvwm2, который можно было назвать довольно популярным — где-нибудь этак в середине девяностых; приведённые на рисунках скриншоты сделаны именно под ним. Поскольку вы, скорее всего, используете другой оконный менеджер, внешний вид оформления окошек у вас получится не такой, как на рисунках. Совпадать будет только вид внутренней области окна.

[32] Строго говоря, «активный виджет» — это термин, предполагающий не просто *возможность* для виджета обрабатывать события, но и то, что данный конкретный виджет на экране в настоящий момент для этого доступен. Виджеты вывода активными не бывают никогда, но и виджет ввода *не обязательно* является активным — его можно деактивировать, и он тогда никакие события обрабатывать не будет, пока его снова не активируют. Такие «деактивированные» виджеты, по-английски чаще называемые словом *disabled*, обычно отрисовываются каким-нибудь серым цветом или просто более тусклым, чем обычно, чтобы показать пользователю, что прямо сейчас данный виджет для взаимодействия недоступен.

```
    printf("%s\n", (const char*)user);
}
static void exit_callback(Fl_Widget *, void *)
{
    exit(0);
}
```

Можно, впрочем, завершать программу менее радикальным способом. Функция `Fl::run`, реализующая главный цикл, работает до тех пор, пока хотя бы одно окно, созданное нашей программой, остаётся в состоянии видимого, когда же видимые окна кончились, она возвращает управление. Окно у нас всего одно, так что достаточно его закрыть — и дело сделано. Адрес главного окна можно было бы передать в `exit_callback` через второй параметр, но можно сделать ещё проще. Первый параметр здесь есть всегда, и в данном случае через него будет передан адрес объекта кнопки `Quit`, а мы точно знаем, что эта кнопка расположена непосредственно в главном окне, то есть главное окно является её *родительским* виджетом, так что `exit_callback` может выглядеть так:

```
static void exit_callback(Fl_Widget *w, void *)
{
    w->parent()->hide();
}
```

У такого решения есть один серьёзный недостаток: оно завязано на тот факт, что кнопка выхода не вложена ни во что, кроме главного окна. Это может не всегда быть так — например, ради более эстетичного вида окна или с целью тонкой настройки процесса смены размера окон мы можем эту кнопку вложить в группу (виджет `Fl_Group`) или куда-то ещё, причём решение об этом мы можем принять уже после того, как функция `exit_callback` будет написана, и, конечно же, забудем её изменить. Правильнее будет другое решение: двигаться от родителя к родителю до тех пор, пока не найдём виджет, родителя не имеющий (для него `parent` вернёт нулевой указатель) — это и будет главное окно. В таком варианте наша функция примет следующий вид:

```
static void exit_callback(Fl_Widget *w, void *)
{
    Fl_Widget *p;
    do {
        p = w->parent();
        if(p)
            w = p;
    } while(p);
    w->hide();
}
```

Имея обе callback-функции, мы можем приступить к формированию нашего диалогового окна. Обычно в таких случаях трудно заранее предугадать, каковы должны быть размеры всех наших виджетов, чтобы полученное окно смотрелось если не красиво, то хотя бы не слишком уродливо; между тем именно этого — указания расположения и размеров с точностью до пикселя — от нас требуют конструкторы виджетов FLTK. Как правило, прежде чем окно обретёт окончательный вид, приходится внести несколько коррекций в выбранные размеры. При этом числа, передаваемые конструкторам, зависят друг от друга; например, мы сейчас собираемся нарисовать три кнопки одну над другой, так что вертикальное расположение каждой из них, кроме первой, будет зависеть от высоты предыдущих кнопок; плюс к тому сами кнопки, по-видимому, стоит сделать одинакового размера, но угадать, какого именно, тоже с первого раза не получится. Поэтому мы применим старый добрый принцип: **никакие числа не должны встречаться в программе в явном виде, всем константам следует дать имена**. Констант у нас будет четыре: ширина кнопки, высота кнопки, расстояние между кнопками, а также между кнопкой и краем окна (вообще говоря, эти два расстояния не обязаны совпадать, но для упрощения картины будем считать их одинаковыми), и размер шрифта для надписей на кнопках. Получаем следующее:

```
enum {
    spacing = 5,        // промежутки
    button_w = 200,     // ширина кнопки
    button_h = 40,      // высота кнопки
    font_size = 20      // размер шрифта
};
```

Здесь показаны окончательные размеры, полученные методом проб и ошибок за три или четыре попытки. Чтобы создать все три объекта кнопок одним циклом, опишем надписи на кнопках в виде массива:

```
static const char *msg[] = {
    "Say hello", "Say goodbye", "Quit"
};
```

Теперь можно приступать к написанию функции `main`. Начнём её с создания объекта главного окна, но предварительно посчитаем, какого оно должно быть размера. По горизонтали у нас получается одна кнопка и два промежутка — слева и справа от кнопки; по вертикали — три кнопки и четыре промежутка. Имеем следующее:

```
int main(int argc, char **argv)
{
    int win_w = button_w + spacing * 2;
    int win_h = button_h * 3 + spacing * 4;
    Fl_Window *win = new Fl_Window(win_w, win_h, "buttons demo");
```

Теперь пора создавать объекты кнопок. Поскольку потом нужно будет ещё установить им callback-функции, адреса создаваемых объектов сохраним в массиве из трёх указателей; при создании объектов горизонтальное положение, длина и ширина будут каждый раз те же самые, сообщения мы возьмём из массива `msg`, а положение по вертикали для текущей кнопки будем хранить в переменной `y` и каждый раз после создания кнопки увеличивать:

```
    Fl_Button *b[3];
    int i;
    int y = spacing;
    for(i = 0; i < 3; i++) {
        b[i] = new Fl_Button(spacing, y,
                             button_w, button_h, msg[i]);
        b[i]->labelsize(font_size);
        y += button_h + spacing;
    }
```

Поскольку больше дочерних виджетов не предполагается, сообщим главному окну, что с формированием его содержимого мы закончили:

```
    win->end();
```

Установим реакцию на нажатие каждой из кнопок:

```
    b[0]->callback(say_callback, (void*)"Hello, world!");
    b[1]->callback(say_callback, (void*)"Goodbye, world!");
    b[2]->callback(exit_callback, 0);
```

Здесь в каждой из строчек первый параметр метода `callback`, как можно догадаться, задаёт адрес функции, которую следует вызывать при активации кнопки, а второй — то значение, которое в вызываемую функцию будет передаваться вторым параметром (параметром `user`).

Все виджеты готовы, остаётся только вывести главное окно на экран и запустить цикл обработки событий:

```
    win->show();
    return Fl::run();
}
```

Полностью этот пример читатель найдёт в файле `fltk/speak.cpp`.

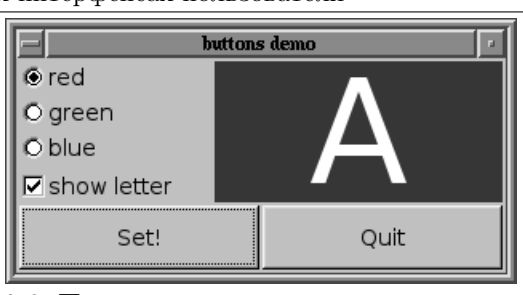

Рис. 10.6. Программа с радиокнопками и галочкой

10.11.3. Другие виды кнопок

Помимо обычных кнопок, FLTK поддерживает целый ряд виджетов, классы которых унаследованы от `Fl_Button`, то есть, строго говоря, тоже являются кнопками, но *другими*. Всё их многообразие мы рассматривать не будем, ограничимся двумя видами виджетов, которые часто применяются в GUI вне зависимости от того, какая для этого используется библиотека.

Первый из них по-английски называется *checkbox* и представляет собой подписанный квадратик, в который можно поставить галочку. Этот вид виджетов в библиотеке FLTK реализован классом `Fl_Check_Button`. Работает checkbox очень просто: если щёлкнуть по нему, галочка ставится, если щёлкнуть ещё раз — галочка снимается (исчезает).

Второй вид кнопок, которые мы рассмотрим в этом параграфе, обычно называют *radio buttons*. Название происходит от старых радиоприёмников, на которых обычно присутствовал целый ряд кнопок для выбора используемого диапазона, причём при нажатии одной из кнопок она залипала, а если до этого была нажата другая — она отскакивала, так что в любой момент оставалась нажатой только одна из кнопок. В графических пользовательских интерфейсах такие «радиокнопки» применяются для выбора из нескольких фиксированных вариантов и выглядят обычно как расположенные в столбик кружочки, в одном из которых может стоять отметка, а справа (реже — слева) от кружочков помещается текст, описывающий каждый из вариантов выбора. Если щёлкнуть по одной из «радиокнопок», входящих в единую группу, то она становится выбранной (пометка с этого момента помещается в её кружочке), а остальные радиокнопки той же группы теряют статус выбранных.

Радиокнопка привычного нам вида в FLTK реализована классом `Fl_Radio_Round_Button`. Отметим, что в FLTK есть также виджет `Fl_Radio_Button`, но делает он не совсем то: его объекты выглядят как простые кнопки без всяких кружочков, просто эти кнопки объединяются в группу, в которой нажатая кнопка залипает, а остальные при этом отскакивают. Вроде бы это тот же самый выбор одного варианта из нескольких, просто выглядит непривычно.

Очевидно, что отдельные объекты радиокнопок нужно как-то объединять в группы, и каждый из объектов должен знать, какие ещё объекты входят в ту же группу, чтобы сообщить им, что пользователь остановил свой выбор на нём, так что остальным надлежит перейти в невыбранное состояние. В FLTK принято довольно простое решение, изрядно экономящее силы программиста: радиокнопки автоматически объединяются в группу, если у них один родительский виджет. В нашем случае в роли такого виджета будет выступать главное окно; если бы нам нужно было создать несколько групп радиокнопок, достаточно было бы описать соответствующее количество виджетов типа `Fl_Group`, а сами кнопки располагать уже в них.

Наша демонстрационная программа задействует все три вида кнопок, которые мы включили в рассмотрение: обычные кнопки, радиокнопки и галочку (см. рис. 10.6). Пространство окна, свободное от кнопок, мы займём виджетом типа `Fl_Box`, который будет менять цвет своего фона в зависимости от того, какая из трёх радиокнопок выбрана; кроме того, этот виджет будет показывать и скрывать большую букву «`A`» в зависимости от того, установлена или сброшена галочка. В нижней части окна мы расположим две кнопки; нажатие на левую будет приводить виджет `Fl_Box` в соответствие с состоянием галочки и радиокнопок, а правая кнопка будет завершать работу программы.

С реакцией на нажатие правой кнопки всё понятно, точно такую же кнопку мы использовали в предыдущем примере, поэтому просто скопируем из него функцию `exit_callback` (см. стр. 93). Сложнее будет организовать реакцию на левую кнопку. Программа при этом должна изменить настройки *бокса* (виджета типа `Fl_Box`) — его цвет фона и отображаемый в нём текст — в соответствии с тем, что́ пользователь сделал с галочкой и радиокнопками, но для этого нужно иметь доступ и к объектам кнопок, и к объекту бокса. В принципе можно было бы сделать все эти объекты (точнее, указатели на них) глобальными переменными, но мы прекрасно знаем, что глобальные переменные — сущность вредная и лучше не привыкать использовать их по пустякам. Поэтому мы поступим иначе: опишем структуру, в которой будут предусмотрены указатели на все интересующие нас виджеты, переменную типа этой структуры создадим в функции `main`, а в callback-функцию нашу структуру передадим через параметр `user`. Сама структура будет выглядеть так:

```
struct controls {
    Fl_Radio_Round_Button *rb[3];
    Fl_Check_Button *cb;
    Fl_Box *box;
};
```

Прежде чем двигаться дальше, создадим два константных массива, каждый из трёх элементов. В первом разместим коды цветов, в которые следует покрасить бокс при выборе радиокнопки с соответствующим номером, во втором — текст для самих радиокнопок. Поскольку они очевидным образом связаны между собой, расположим их в тексте программы рядом, чтобы при изменении любого из них не забыть про второй.

```
static const int colors[] = { FL_RED, FL_GREEN, FL_BLUE };
static const char *const colnames[] =
                                { "red", "green", "blue" };
```

Чтобы написать подходящую callback-функцию, нужно знать, что у всех объектов-виджетов, наследуемых от `Fl_Button`, есть метод `value`, который возвращает истину (единицу), если кнопка нажата (радиокнопка выбрана, галочка поставлена), и ложь (ноль) в противном случае. Для изменений внешнего вида бокса воспользуемся методами `color` и `label`, которые устанавливают соответственно цвет фона и текст надписи; если галочка сброшена, в качестве текста будем использовать пустую строку. Остальные параметры бокса — цвет и размер текста, вид самого бокса — мы установим при его создании в функции `main`. Текст callback-функции получается таким:

```
static void set_callback(Fl_Widget *w, void *user)
{
    controls *c = (controls*)user;
    int i;
    for(i = 0; i < 3; i++) {
        if(c->rb[i]->value()) {
            c->box->color(colors[i]);
            break;
        }
    }
    c->box->label(c->cb->value() ? "A" : "");
}
```

Отметим, что указатель `c` мы здесь ввели исключительно для краткости записи; его значение (физически, то есть содержимое области памяти, занятой этим указателем) тождественно значению параметра `user`, `c` отличается от `user` только типом. Скорее всего, в машинном коде указателя `c` уже не будет, оптимизатор его ликвидирует.

Прежде чем писать функцию `main`, предусмотрим константы для всех размеров, от которых зависит положение виджетов и размер главного окна. Констант в этот раз получается шесть: ширина и высота для радиокнопок и чекбокса, ширина и высота для обычных кнопок, расположенных внизу окна, размер промежутка между кнопками (и между краем окна и кнопкой) и размер шрифта для буквы, отображаемой в боксе справа при установленной галочке:

```
enum {
    spacing = 2,
    button_w = 150,
    button_h = 40,
    option_w = 120,
    option_h = 20,
    letter_size = 90
};
```

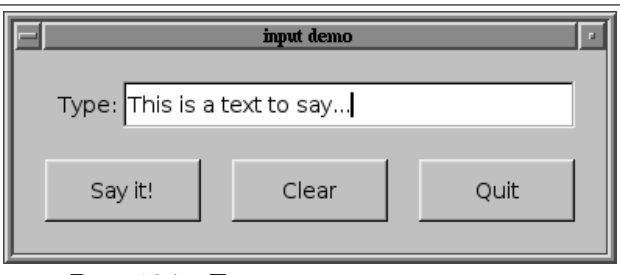

Рис. 10.7. Демонстрация поля ввода

Функцию `main` начнём с вычисления размеров главного окна и собственно создания объекта главного окна:

```
int main()
{
    int win_w = button_w * 2 + spacing * 3;
    int win_h = option_h * 4 + button_h + spacing * 7;
    Fl_Window *win = new Fl_Window(win_w, win_h, "buttons demo");
```

Теперь создадим в динамической памяти структуру `controls`:

```
    controls *ctrl = new controls;
```

и начнём наполнять наше окно виджетами. С радиокнопкой и галочками всё более-менее понятно, отметим только, что галочку мы отодвинем по вертикали от радиокнопок на двойное расстояние. Код получится таким:

```
    int i;
    for(i = 0; i < 3; i++) {
        int y = spacing * (i+1) + option_h * i;
        ctrl->rb[i] = new Fl_Radio_Round_Button(spacing, y,
                            option_w, option_h, colnames[i]);
    }
    ctrl->cb =
        new Fl_Check_Button(spacing, 5*spacing + 3*option_h,
                            option_w, option_h, "show letter");
```

Несколько сложнее обстоят дела с боксом. Его размеры, в отличие от размеров других виджетов, не задаются константами — они полностью определяются тем, сколько места занимают другие виджеты. По вертикали наш бокс будет занимать столько же, сколько занимают радиокнопки и галочка вместе с промежутками между ними, а по горизонтали — столько, насколько радиокнопки меньше, чем две обычные кнопки и промежуток. После создания бокса мы уже знакомым нам методом `labelsize` установим размер текста, затем методом `labelcolor` сделаем этот текст белым (этот цвет лучше всего смотрится на всех трёх наших вариантах цвета фона). Кроме того, мы воспользуемся методом `box`, чтобы установить стиль обрамления[33], а также текст, расположение, параметры шрифта и некоторые другие параметры для *метки*. Если этого не сделать, бокс останется *прозрачным*, так что видно будет только его текст (если таковой есть), а самого бокса видно не будет, то есть все наши усилия по установке цвета фона пропадут впустую. В качестве типа мы выберем `FL_FLAT_BOX` — это простой прямоугольник. Полностью код для создания бокса и его первичной настройки будет таким:

```
    int box_x = 2 * spacing + option_w;
    int box_w = 2 * button_w + spacing - option_w - spacing;
    int box_h = 4 * option_h + 4 * spacing;
    ctrl->box = new Fl_Box(box_x, spacing, box_w, box_h);
    ctrl->box->labelsize(letter_size);
    ctrl->box->labelcolor(FL_WHITE);
    ctrl->box->box(FL_FLAT_BOX);
```

Остаются ещё кнопки, но с этим проще, ведь с кнопками мы уже работали:

```
    int buttons_y = 6 * spacing + 4 * option_h;
    Fl_Button *set_b =
        new Fl_Button(spacing,
                      buttons_y, button_w, button_h, "Set!");
    set_b->callback(set_callback, (void*)ctrl);

    Fl_Button *close_b =
        new Fl_Button(2*spacing + button_w,
                      buttons_y, button_w, button_h, "Quit");
    close_b->callback(exit_callback, 0);
```

Всё, все виджеты готовы, остаётся сообщить об этом главному окну, вывести его на экран и запустить главный цикл:

```
    win->end();
    win->show();
    return Fl::run();
}
```

Полный код этого примера находится в файле `fltk/options.cpp`.

В последние годы среди разработчиков пользовательских интерфейсов, в особенности веб-приложений, стало модно обходиться без кнопки, по которой выбранные пользователем варианты приводятся в исполнение (в нашем случае в этой роли выступает кнопка `Set!`). Вместо этого все изменения происходят сразу, как только пользователь хоть что-то изменит в диалоге. Мы тоже могли бы так сделать, это очень просто: кнопку `Set!` убрать, а callback-функцию (собственно говоря, ту же самую, которая у нас уже есть) назначить всем трём радиокнопкам и кнопке-галочке, благо метод `callback` у них у всех есть и делает то же самое, что и для обычной кнопки. Так вот, *не делайте этого*, и если кто-нибудь скажет вам, что так якобы удобнее для пользователя — не обращайте внимания на таких советчиков.

10.11.4. Виджеты для ввода текста

Для того, чтобы дать возможность пользователю вводить текст, FLTK предоставляет три основных типа виджетов: однострочное поле ввода (класс `Fl_Input`), многострочное поле (`Fl_Multiline_Input`) и текстовый редактор (`Fl_Text_Editor`).

Начнём с однострочного поля как самого простого. Конструктор класса `Fl_Input`, как и конструкторы большинства виджетов, принимает пять параметров — координаты, размеры и текст метки, причём последний параметр можно не указывать. Если его всё-таки указать, то само поле ввода создаётся в соответствии с заданными координатами, а слева к нему добавляется ещё и текст метки, что может оказаться довольно неожиданным. Чтобы всё выглядело «как нужно», для метки приходится оставлять запас; к тому же далеко не всегда её положение слева от поля ввода удачно вписывается в общую концепцию построения диалогового окна. В принципе положение метки можно изменить; как этим управляют, мы расскажем в §10.11.5. Тем не менее часто бывает проще оставить `Fl_Input` без метки, а все нужные надписи вывести с помощью других виджетов, хотя бы уже знакомого нам `Fl_Box`.

Класс `Fl_Input` снабжён методом `value` без параметров, который возвращает значение типа `const char*` — адрес строки, соответствующей текущему тексту, набранному в поле ввода. Метод с таким же именем, но с параметром типа `const char*` (и ничего не возвращающий), используется для *изменения* текущего текста.

Сделаем для начала простейшую демонстрационную программу, которая нарисует окно с одним полем ввода и тремя кнопками; по нажатию первой из них строка из поля ввода будет выдаваться в стандартный поток вывода, нажатие второй кнопки очистит поле ввода, третья кнопка, как обычно, будет завершать работу (см. рис. 10.7). Для отработки нажатий первой и второй кнопки потребуется доступ к объекту виджета поля ввода, так что его (точнее, его адрес) мы и передадим в callback-функции через параметр `user`.

Прежде чем писать текст callback-функций, сделаем ещё одно замечание. Если наша программа использует несколько активных виджетов, в каждый момент только один из них обладает так называемым ***фокусом ввода***, то есть является получателем ввода с клавиатуры. Надо сказать, что клавиатурные события нужны не только полям ввода текста; в частности, обыкновенные кнопки тоже умеют реагировать на клавиатурный ввод — например, нажать на кнопку (а также поставить или снять галочку, выбрать радиокнопку) можно клавишей пробела, при условии, конечно, что именно эта кнопка в настоящий момент владеет фокусом ввода. Вы можете поэкспериментировать с фокусом ввода, запустив любой из наших предыдущих примеров; виджет, обладающий фокусом ввода, отмечается едва заметной пунктирной рамочкой, а переместить фокус можно нажатием на клавишу Tab (при этом комбинация Shift-Tab перемещает фокус в обратном направлении).

При активации любого из виджетов мышкой — в том числе при исполненном с помощью мышки нажатии на нарисованную кнопку — фокус обычно остаётся у того виджета, который был активирован. Во всех примерах, рас-

[33] В оригинале — просто *box type*; перевести это как «тип бокса» формально можно, но не хочется.

смотренных до сих пор, мы не обращали на это внимания, поскольку вообще не собирались ничего вводить с клавиатуры, но как только у нас появляется поле для ввода текста, потеря этим полем фокуса ввода способна пользователя изрядно дезориентировать, а затем и разозлить: вот только что вроде бы нормально можно было вводить текст с клавиатуры, а теперь почему-то ничего не работает. Конечно, исправить ситуацию очень просто, достаточно щёлкнуть мышкой в области поля ввода, и фокус снова окажется там — но, хотите верьте, хотите нет, не каждый пользователь до этого сразу додумается. Поэтому мы поступим вежливо: в ходе обработки нажатия на кнопку будем принудительно возвращать фокус виджету `Fl_Input`. Делается это очень просто — вызовом метода `take_focus` (без параметров). В итоге callback-функции получатся такими:

```
static void say_callback(Fl_Widget *w, void *user)
{
    printf("%s\n", ((Fl_Input*)user)->value());
    ((Fl_Input*)user)->take_focus();
}
static void clear_callback(Fl_Widget *w, void *user)
{
    ((Fl_Input*)user)->value("");
    ((Fl_Input*)user)->take_focus();
}
```

Для построения диалогового окна, показанного на рисунке, нам потребовались следующие константы:

```
enum {
    spacing = 20,
    input_h = 30,      // высота поля ввода
    label_w = 50,      // ширина метки поля ввода
    button_w = 100,
    button_h = 40
};
```

Текст функции `main` целиком мы приводить не будем, отметим только, что само поле ввода мы создали так:

```
    int inp_w = button_w * 3 + 2 * spacing - label_w;
    Fl_Input *inp = new Fl_Input(spacing + label_w, spacing,
                                 inp_w, input_h, "Type:");
```

а после создания объектов кнопок назначили им callback-функции вот так:

```
    say_b->callback(say_callback, (void*)inp);
    clear_b->callback(clear_callback, (void*)inp);
```

Создание объекта главного окна и объектов кнопок, вывод окна на экран и запуск главного цикла производятся точно так же, как во всех предыдущих примерах. После этого всё заработает, но у программы можно будет обнаружить существенный недостаток: чтобы что-то сделать с набранным текстом, приходится тянуться к мышке, а ведь набирали-то мы текст, очевидно, на клавиатуре. Не все помнят, что нарисованную кнопку можно нажать с клавиатуры с помощью Tab и пробела, да и вообще это долго и неудобно; вы и сами можете заметить, что неосознанно ожидаете возможности после ввода нужного текста нажать на клавишу Enter, чтобы с этим текстом произошло то, для чего вы его вводили.

Заставить нашу программу адекватно реагировать на Enter проще простого. Виджет `Fl_Input` *тоже умеет вызывать callback-функции*; собственно говоря, это умеют вообще все виджеты FLTK, соответствующая функциональность реализована методами класса `Fl_Widget`, базового для всех виджетов. По умолчанию виджет поля ввода вызывает свой callback, когда из него уходит фокус ввода и при этом вводимый текст был изменён, но это не совсем то, что нам нужно, поэтому мы задействуем метод `when` и сообщим виджету, что callback нужно вызывать при нажатии на Enter, причём даже в том случае, если текст не менялся. Что касается самой callback-функции, то нам вполне подойдёт уже имеющаяся `say_callback`. Итак, в текст программы сразу после создания объекта `Fl_Input` мы вставим следующие две строки:

```
    inp->callback(say_callback, (void*)inp);
    inp->when(FL_WHEN_ENTER_KEY|FL_WHEN_NOT_CHANGED);
```

Файл с текстом этой программы называется `fltk/say_it.cpp`.

Класс `Fl_Input` предусматривает целый набор методов для управления происходящим. Так, в классе имеется три метода с именем `position`: без параметров, с одним целочисленным параметром и двумя; метод без параметров возвращает текущую позицию курсора, метод с одним параметром *устанавливает* её (курсор при этом перемещается в указанное место), метод с двумя параметрами устанавливает позицию для курсора и для противоположного конца выделенного текста. Этот последний метод избыточен, поскольку поддерживаются также два метода `mark` — один с параметром, другой без; как можно догадаться, метод `mark` без параметра возвращает границу выделения, противоположную курсору, а `mark` с параметром устанавливает её. Когда выделения нет, эта позиция совпадает с позицией курсора.

Метод `size` без параметров возвращает текущую длину текста в поле ввода (текста, возвращаемого функцией `value`). Такое именование не вполне удачно, поскольку метод `size` с двумя параметрами — не только для этого виджета, а вообще для всех — предназначен для управления шириной и высотой самого виджета.

Метод `replace`, имеющий четыре параметра, из которых обычно используются только три первых, позволяет заменить любой фрагмент текста от одной заданной позиции до другой (не включая её) на новую строку; первые два параметра задают позиции, третий — строку. Если позиции совпадают, ничего не удаляется, строка вставляется перед указанной позицией; если строка пустая или передан нулевой указатель, текст между заданными позициями просто удаляется. Два метода, специально предназначенные для удаления и вставки — `cut` и `insert` — на самом деле просто вызывают `replace` с соответствующими параметрами. Надо сказать, что и в ходе нормальной работы, когда пользователь вводит или удаляет текст, окончательная его модификация происходит через метод `replace`, что позволяет объекту виджета помнить историю редактирования; вызовом метода `undo` можно последовательно отменить (естественно, в обратном порядке) действия, сохранённые в истории.

В классе предусмотрены и другие методы; подробности мы опускаем, их всегда можно найти в технической документации на сайте `http://www.fltk.org`.

От класса `Fl_Input` унаследованы специфические классы `Fl_Int_Input` и `Fl_Float_Input`, позволяющие вводить соответственно только целые числа (состоящие из десятичных цифр либо шестнадцатеричные, начинающиеся с `0x`) и числа с плавающей точкой (в том числе в *научной* нотации — с буквой `E` или `e`). Ещё один наследник — `Fl_Secret_Input` — при вводе вместо текста показывает в зависимости от используемой системы то ли звёздочки, то ли кружочки, так что его можно использовать для ввода паролей. В остальном работа с этими виджетами строится точно так же, как и с самим `Fl_Input`.

Среди наследников выделяется класс `Fl_Multiline_Input`; как можно догадаться по его названию, он позволяет вводить и редактировать текст, состоящий более чем из одной строки — точнее говоря, вводимый текст с программной точки зрения по-прежнему представлен как строка языка Си, но в этой строке могут присутствовать символы перевода строки, и виджет отображает такой текст в своём окошке разделённым на отдельные строки. В остальном он довольно примитивен. Количество строк, видимых на экране, определяется высотой виджета (сколько поместилось, столько поместилось); текст, не поместившийся в отведённое поле, уползает за экран, как это обычно бывает, но при этом ни вертикальной, ни горизонтальной полосы прокрутки не появляется, а догадаться, что сейчас виден не весь текст, пользователю предлагается самостоятельно, видимо, на основе шестого чувства.

Набор методов у этого виджета точно такой же, как у `Fl_Input`, ведь это его базовый класс. Отметим разве что метод `wrap` с одним целочисленным параметром; этот параметр, в действительности трактуемый как логическое значение (истина/ложь), определяет, как виджет будет показывать длинные строки, не влезающие в поле ввода по горизонтали: если параметр задать истинным, тем самым включив режим переноса, строки будут переноситься по пробельным символам, так что не влезть по горизонтали теперь может разве что слишком длинное слово, ну а если режим выключен, то каждая строка текста отображается ровно в одну строку на экране (длинные строки отображаются не целиком).

Метку (последний параметр конструктора) для многострочных полей ввода лучше не указывать, поскольку располагается она по-прежнему слева от

окошка и выглядит это довольно нелепо — правильнее будет разместить некую надпись сверху от поля ввода, воспользовавшись для этого объектом `Fl_Box`. Кроме того, для такого виджета лучше не включать выполнение callback'а по нажатию Enter, как мы это делали в примере с обычным полем ввода; здесь клавиша Enter вам потребуется, чтобы вводить перевод строки.

Авторы FLTK в документации рекомендуют использовать класс `Fl_Text_Editor` вместо `Fl_Multiline_Input` для любых случаев, кроме самых примитивных. Этот виджет предусматривает не только скроллинговые полоски, но и разнообразные другие возможности и, судя по всему, рассчитан на редактирование текстов большого объёма. В частности, сам редактируемый текст здесь представляется не как традиционная строка, а как особый объект (типа `Fl_Text_Buffer`), который, по-видимому, каждую строчку редактируемого текста хранит отдельно. Впрочем, получить из него весь текст целиком проблем не составляет, для этого есть метод `text`, возвращающий значение типа `char*` — адрес искомой строки, причём эту строку метод формирует в динамической памяти специально чтобы вернуть по запросу, и вызывающему предлагается её удалить самостоятельно с помощью функции `free`, что вообще-то не слишком характерно для Си++ — но в документации сказано именно это, и, по-видимому, авторы FLTK для создания этой строки пользуются то ли `malloc`'ом, то ли `calloc`'ом.

Подготовка объекта `Fl_Text_Editor` к работе выглядит примерно так:

```
Fl_Text_Buffer *buf = new Fl_Text_Buffer;
Fl_Text_Editor *ed = new Fl_Text_Editor(x, y, w, h);
ed->buffer(buf);
```

Если бы нам пришло в голову с помощью этого виджета создать демо, подобное рассмотренному выше для `Fl_Input`, то callback-функции для выдачи текста в стандартный вывод и очистки текста могли бы быть такими:

```
static void say_callback(Fl_Widget *w, void *user)
{
    char *s = ((Fl_Text_Editor*)user)->buffer()->text();
    fputs(s, stdout);
    fflush(stdout);
    free(s);
    ((Fl_Text_Editor*)user)->take_focus();
}
static void clear_callback(Fl_Widget *w, void *user)
{
    ((Fl_Text_Editor*)user)->buffer()->text("");
    ((Fl_Text_Editor*)user)->take_focus();
}
```

В документации неоднократно подчёркивается, что один буфер можно использовать одновременно несколькими объектами класса `Fl_Text_Editor`, а также его базового класса `Fl_Text_Display`, предназначенного для просмотра текста без возможности его изменения. Верно и другое: можно завести несколько таких буферов и переключать редактор между ними, используя метод `buffer` — например, при одновременном редактировании нескольких файлов.

Более подробно этот виджет мы рассматривать не будем, поскольку нужен он сравнительно редко, и если всё же понадобится, для его изучения можно воспользоваться технической документацией.

10.11.5. Обрамление и метки

Класс `Fl_Widget`, базовый для всех виджетов FLTK, предусматривает методы, задающие цвет фона и текста, стиль обрамления (*box type*). Поскольку всё это реализовано в базовом классе, соответствующими возможностями обладают практически все виджеты, реализованные в FLTK — точнее, некоторые виджеты в силу своей природы часть возможностей игнорируют (например, `Fl_Window` использует только текст метки для заголовка окна), но для большинства вся эта механика работает с вполне ожидаемыми эффектами. При этом виджет `Fl_Box`, с которым мы уже встречались в §10.11.1, *только это и может*, то есть он специально предназначен, чтобы нарисовать рамку или бокс и, возможно, разместить в этой рамке или рядом с ней некий текст.

Начнём с обрамления. Здесь у нас две основные возможности выбора — форма (тип) рамки/бокса и цвет фона. Методы, позволяющие их настраивать, нам уже знакомы по примеру `fltk/options.cpp` (см. §10.11.3); в этом примере мы установили виджету класса `Fl_Box` тип бокса `FL_FLAT_BOX` с помощью метода `box`, а в ходе работы программы меняли его фоновый цвет с помощью метода `color`. Доступные в FLTK типы рамок и боксов показаны на рис. 10.8. Хотя этот рисунок и заимствован нами из официальной документации по FLTK, в той же самой документации есть замечание, что доступные типы обрамления на рисунке показаны не все; впрочем, пожалуй, такого ассортимента рамочек вам должно хватить. Боксы, как можно заметить, отличаются от рамок тем, что закрашивают всё пространство виджета установленным для него фоновым цветом, тогда как рамки этого не делают — виджет, которому в качестве типа оформления выбрали одну из рамок, использует цвет фона только при рисовании самой рамки, а внутреннее пространство остаётся имеющим фоновый цвет вышестоящего виджета (например, главного окна). Между прочим, для виджетов с `FL_NO_BOX` цвет фона вообще ни на что не влияет — он нигде не используется.

Метод `color`, устанавливающий для виджета фоновый цвет, сам по себе устроен просто — его единственным аргументом служит некий код цвета, и всё; несколько сложнее ситуация с этими вот *кодами цветов* и вообще моделью работы с цветом библиотеки FLTK. Подробно мы всю эту конструкцию обсуждать не станем, отметим только, что для основных цветов имеются обозначения `FL_WHITE` (белый), `FL_BLACK` (чёрный), `FL_RED` (красный), `FL_GREEN` (зелёный), `FL_YELLOW` (жёлтый), `FL_BLUE` (синий), `FL_MAGENTA` (пурпурный), `FL_CYAN` (бирюзовый), `FL_DARK_RED` (тёмно-красный), `FL_DARK_GREEN`, `FL_DARK_YELLOW`, `FL_DARK_BLUE`, `FL_DARK_MAGENTA`, `FL_DARK_CYAN`; несколько сложнее с оттенками серого, но `FL_GRAY`, скорее всего, окажется тем, что вам требуется. Кроме того, цвета, используемые по умолчанию, тоже имеют обозначения: `FL_FOREGROUND_COLOR` (цвет текста), `FL_BACKGROUND_COLOR` (цвет фона), `FL_INACTIVE_COLOR` (цвет неактивных/выключенных виджетов), `FL_SELECTION_COLOR` (цвет выделения текста и пунктов меню). Если этих цветов не хватило, можно вместо кода цвета использовать результат, возвращаемый функцией `fl_rgb_color`; эта функция принимает три параметра — целых числа, соответствующих компонентам RGB (красный, зелёный, синий), и возвращает число, подходящее в качестве аргумента для методов, настраивающих цвета.

Отметим заодно, что цвет текста, используемый для отображения метки, устанавливается методом `labelcolor`.

С текстом метки в FLTK имеются свои хитрости. Как мы уже знаем, этот текст может быть задан через параметр конструктора при создании объекта виджета (если его не указать, метки у виджета не будет), но позже текст можно изменить, вызвав метод `label`. Кроме него, есть ещё метод `copy_label`, принимающий точно так же, как и `label`, строку; разница между ними в том, что `copy_label` *создаёт копию строки* внутри объекта виджета, тогда как `label` просто присваивает указатель, предполагая, что переданная ему строка будет продолжать существовать в неизменном виде всё время, пока объект виджета не исчезнет или ему не поменяют текст метки. Ясно, что при использовании для этих целей строкового литерала целесообразнее применить метод `label` — строковый литерал (как область памяти), как известно, вообще нельзя поменять во время работы программы, так что копировать его заведомо нет смысла. Если же вы хотите, чтобы виджет отобразил в качестве метки строку, скомпонованную во время работы программы (например, содержащую какие-нибудь вычисленные результаты, сообщения и т. п.), пожалуй, проще будет попросить виджет создать копию, а своим экземпляром строки пользоваться без оглядки на нужды виджета.

Отметим, что **строка, передаваемая в качестве текста метки через конструктор, не копируется**, то есть с ней объект виджета обращается так же, как и со строкой, переданной через метод `label` — просто запоминает её адрес; как ни странно, в документации об этом нет ни слова, но это легко понять из исходных текстов класса `Fl_Widget`.

Вторая хитрость состоит в том, что символ `@` в метках виджетов играет особую роль, так что если вам нужен сам этот символ, его придётся «удвоить»; например, адрес электронной почты следует передать примерно в таком виде: `vasya@@example.com`. Символ `@` используется в тексте меток, чтобы вставлять в них пиктограммы из предопределённого набора (см. рис. 10.9).

Набор доступных пиктограмм, как можно заметить, довольно скромен; при большом желании и если вы не боитесь сложностей, этот набор можно расширить, изучив входящие в FLTK функции рисования (там есть и такие; вообще там много чего есть, но полный набор возможностей FLTK в формат нашей книги никак не вписывается) и посмотрев, как сделаны имеющиеся пиктограммы, в файле `src/fl_symbols.cxx` в исходных текстах библиотеки. Но и имеющиеся пиктограммы могут оказаться очень полезны, особенно если учесть, что любую из них можно повернуть на любой угол, кратный 1°, развернуть вокруг горизонтальной или вертикальной оси, увеличить и уменьшить.

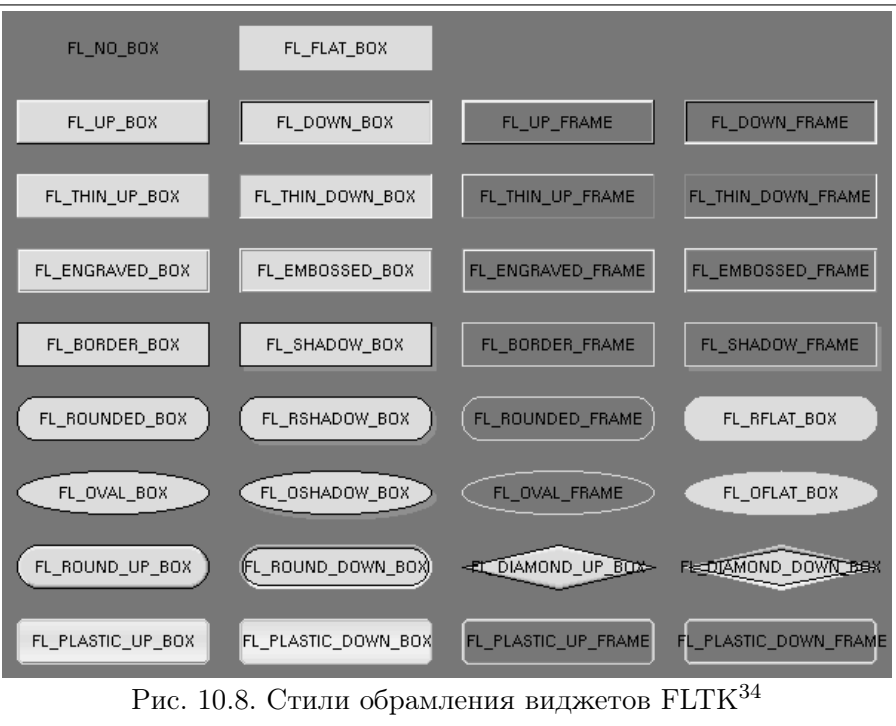

Рис. 10.8. Стили обрамления виджетов FLTK[34]

Рис. 10.9. Пиктограммы для текстовых меток FLTK[35]

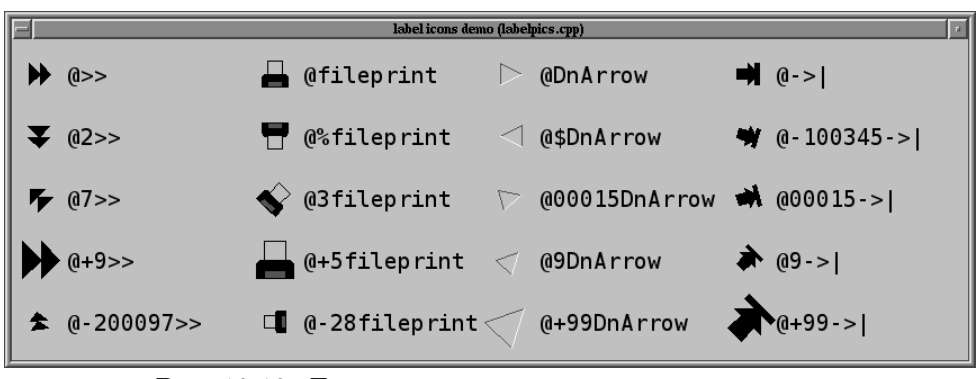

Рис. 10.10. Демонстрация меток с пиктограммами

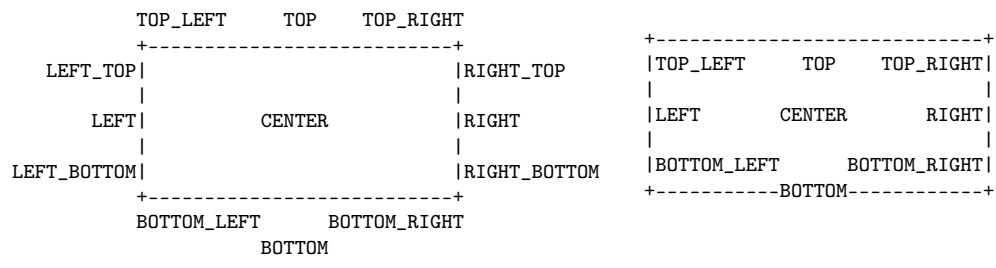

Рис. 10.11. Константы для обозначения положения метки

Посмотрим, как это делается. Для примера рассмотрим значок быстрой перемотки вперёд, который обозначается `@>>`. Сразу после знака `@` можно вставить (в указанном порядке):

- модификатор размера от `-9` (самый маленький) до `+9` (самый большой); учтите, что знак (минус или плюс) тут обязателен;
- знак `$` (обозначает «отражение» вокруг вертикальной оси) или `%` (вокруг горизонтальной);
- одну цифру, отличную от нуля (без знака минуса или плюса), либо ноль, после которого ещё ровно четыре цифры — для обозначения угла поворота.

Последнее нуждается в некоторых пояснениях. Если угол, на который вы хотите повернуть пиктограмму, кратен 45°, градусы можно не высчитывать: угол поворота задаётся циферками в соответствии с их расположением на дополнительной клавиатуре, причём за ноль принимается традиционное направление по горизонтали вправо, так что цифра `6` никакого эффекта не даёт (как и цифра `5`), цифра `9` обозначает поворот на 45° против часовой стрелки, цифра `8` — на 90°, цифра `7` — на 135°, цифра `4` — на 180°; повороты по часовой стрелке задаются цифрами `3`, `2` и `1` — на 45°, 90° и 135° соответственно. Если же угол требуется более хитрый, написать придётся ровно пять цифр, причём первая из них должна быть нулём; вторая, впрочем, тоже, скорее всего, будет нулём, а вот оставшиеся три задают угол в градусах (откладываемый, опять-таки, против часовой стрелки от направления вправо, как это обычно делается в математике).

Например, `@8>>` выдаст всё тот же символ быстрой перемотки, но повёрнутый на 90° против часовой стрелки, т. е. смотрящий «в потолок». Того же эффекта можно добиться и строкой `@00090>>`, а строки `@2>>` и `@00270>>` заставят пиктограмму, наоборот, смотреть «в пол». Строка `@+9>>` выдаст пиктограмму увеличенного размера (примерно в полтора раза) в обычном положении, а, например, строка `@+93>>` — увеличенную пиктограмму, смотрящую вправо-вниз (здесь `+9` задаёт размер, `3` — угол поворота). Строчка `@$-->` перевернёт стрелку, заставив её смотреть влево, а `@%fileprint` выдаст значок принтера, перевёрнутый вверх тормашками.

На рис. 10.10 показан ещё ряд примеров того, что можно сделать с этими пиктограммами. Исходный текст программы, нарисовавшей это окно, находится в файле `fltk/labelpics.cpp`.

Работа с пиктограммами в метках имеет одну особенность, о которой, опять-таки, ни слова не сказано в документации: **последовательность символов, обозначающая пиктограмму, должна находиться либо в самом начале строки метки, либо в самом её конце**. Если символ `@` является первым символом в строке, то в качестве имени пиктограммы рассматриваются все символы строки до первого пробела (или до конца строки, если нет ни одного пробела); если же символ встречен не в первой позиции строки, то FLTK воспримет все последующие символы строки как имя пиктограммы, и никакие пробелы уже не помогут. Чаще всего это нас устраивает: пиктограмма выводится либо одна, без текста, либо весь текст метки расположен от неё с одной стороны — слева или справа. Можно даже выдать две пиктограммы — одна будет в начале, другая в конце — а между ними какой-нибудь текст; если текст вам не нужен, оставьте между пиктограммами хотя бы три пробела (и не спрашивайте, почему с меньшим их количеством ничего работать не будет).

За расположение метки относительно границ виджета отвечает метод `align`, аргументом которому служат константы, имеющие префикс «`FL_ALIGN_`». Обычно виджеты по умолчанию устанавливают это значение в соответствии с собственной природой: так, `Fl_Box` и `Fl_Button` по умолчанию используют `FL_ALIGN_CENTER` (метка располагается в геометрическом центре виджета); `Fl_Input` по умолчанию использует `FL_ALIGN_LEFT`, что, как мы видели, соответствует расположению метки слева от виджета.

Указание константы `FL_ALIGN_INSIDE` задаёт положение метки *внутри* виджета; с прочими константами она комбинируется через операцию побитового *или*. Например,

```
wp->align(FL_ALIGN_INSIDE|FL_ALIGN_BOTTOM);
```

расположит метку объекта виджета `*wp` вдоль его нижней границы *с внутренней стороны виджета* (т. е. *над* нижней границей), тогда как

[34] Иллюстрация из официальной документации FLTK, оригинал находится на странице `https://www.fltk.org/doc-1.3/Enumerations_8H.html`.

[35] Иллюстрация из официальной документации FLTK, оригинал находится на странице `https://www.fltk.org/doc-1.3/common.html`.

Рис. 10.12. Основные шрифты FLTK

```
wp->align(FL_ALIGN_BOTTOM);
```

расположит метку *за пределами виджета* — непосредственно *под* его нижней границей. Все основные константы показаны на рис. 10.11: слева — при расположении за пределами виджета, справа — при расположении внутри виджета (с использованием `FL_ALIGN_INSIDE`). Обратите внимание, что, например, `FL_ALIGN_LEFT_BOTTOM` с `FL_ALIGN_INSIDE` сочетать нельзя, нужно применить `FL_ALIGN_BOTTOM_LEFT`.

Ещё одна константа, `FL_ALIGN_WRAP`, обозначает перенос строк, не поместившихся в отведённое по горизонтали место. Обычно она используется только при внутреннем расположении метки. Отметим, что *многострочную* метку можно сделать и без этого — достаточно в текст метки вставить символы перевода строки.

Цвет текста метки устанавливается методом `labelcolor`, с которым мы уже встречались в §10.11.3. Аргументом ему служит код цвета того же вида, как для рассмотренного выше метода `color`. Размер текста метки (высоту букв) можно изменить с помощью метода `labelsize`, который мы тоже уже неоднократно видели; отметим, что единица измерения здесь — пиксель.

Шрифт, используемый в тексте метки, тоже может быть изменён — с помощью, как можно догадаться, метода `labelfont`. Основные поддерживаемые семейства шрифтов — Helvetica, Courier, Times и некий загадочный Screen; обозначаются они соответственно константами `FL_HELVETICA`, `FL_COURIER`, `FL_TIMES` и `FL_SCREEN`. Ко всем этим константам можно добавить (**в этот раз — с помощью простого сложения, т. е. символом плюса**) константу `FL_BOLD` для включения ожирнения, а ко всем, кроме `FL_SCREEN` — ещё `FL_ITALIC`. Предусмотрены, впрочем, имена вроде `FL_HELVETICA_BOLD`, `FL_COURIER_ITALIC`, `FL_TIMES_BOLD_ITALIC` и т. п.; например, значение `FL_TIMES_BOLD_ITALIC` (как водится, целочисленное) в точности равно сумме констант `FL_TIMES`, `FL_BOLD` и `FL_ITALIC`. FLTK предусматривает возможность добавления дополнительных шрифтов, но обсуждение этой возможности потребовало бы сначала описать всю модель управления шрифтами X Window; для нашей книги это слишком долго.

Напомним, что Helvetica — это шрифт без засечек, Times — шрифт с засечками, а Courier — моноширинный шрифт. Как известно, на экране лучше всего смотрятся шрифты без засечек, на бумаге — шрифты с засечками, а моноширинный шрифт нужен для воспроизведения фрагментов текста на формальных языках (например, исходных текстов программ, текстов в формате HTML и т. п.)

Если верить документации, шрифт Screen вроде бы должен представлять собой моноширинный шрифт, используемый (здесь и сейчас) в окнах эмуляторов терминала, но, как показывает практика, в действительности этот шрифт оказывается, следуя классике, «исключительно чем попало». Если вам нужен моноширинный шрифт, используйте Courier.

Как *обычно* выглядят основные доступные шрифты, показано на рис. 10.12; программу, демонстрирующую это, вы найдёте в файле `fltk/fonts.cpp`. К сожалению, нам придётся предупредить читателя, что библиотека использует шрифты, доступные из X Window (или другой графической подсистемы), так что их внешний вид может различаться от машины к машине, от среды к среде и т. п.

Последний из методов, влияющих на внешний вид текста метки, называется `labeltype`; он позволяет применить к тексту метки некий визуальный эффект вроде «текста с тенью», «выгравированного текста» или «текста бугорком»; кроме того, он же позволяет составить «сложную» метку, содержащую изображение. Подробный рассказ о нём мы опускаем.

Некоторые виджеты, включая `Fl_Input`, используют перечисленные выше возможности для достижения своего штатного внешнего вида; чтобы понять, о чём идёт речь, попробуйте как-нибудь поменять объекту класса `Fl_Input` рамку на какую-нибудь вроде `FL_OFLAT_BOX` и полюбуйтесь результатом, особенно в ходе редактирования текста. Конечно, в «боевых» программах, и тем более в программах, предназначенных для передачи другим людям, такого делать ни в коем случае не следует.

10.11.6. Виджеты для вывода

Любая интерактивная программа — а программы с GUI, естественно, другими быть не могут — нуждается в том, чтобы донести до пользователя информацию о ходе и результатах работы. Конечно, вывести текст на экран можно с помощью меток, подробно рассмотренных в предыдущем параграфе, причём, как можно заметить, FLTK позволяет здесь действовать достаточно гибко — попросту говоря, мы можем заставить выдаваемый текст выглядеть практически *как угодно*, надо только не забывать про разницу между методами `label` и `copy_label` (см. стр. 98) — для информации, которая только что сгенерирована, обычно подходит только второй вариант. К сожалению, у меток FLTK есть один фундаментальный недостаток: **текст метки невозможно выделить мышкой и скопировать в другое окно**. Для многих случаев выводимой информации, даже, возможно, для большинства, этот недостаток оказывается фатальным.

Создавая виджеты, специально предназначенные для вывода и, в отличие от меток, допускающие выделение текста, авторы FLTK пошли весьма незамысловатым путём: унаследовали их от знакомого нам класса `Fl_Input`. Класс `Fl_Output` реализует пассивный виджет, предназначенный для вывода текста в одну строку; по сути это просто то же самое поле ввода, для которого заблокированы собственно операции ввода (редактирования текста). От него (именно от него, а не от `Fl_Multiline_Input`, как можно было бы ожидать) наследуется класс `Fl_Multiline_Output`, позволяющий выводить многострочный текст. Если текст в виджет не помещается, его можно пролистывать стрелками как по горизонтали, так и по вертикали; к сожалению, скроллинговых полосок здесь не предусмотрено точно так же, как и в виджете для ввода.

Виджеты вывода можно снабдить метками, чтобы, например, прокомментировать, что за информация в них выдаётся, но метки будут недоступны для выделения, как и метки всех остальных виджетов; текст, который можно будет выделять, задаётся не как метка, а как *значение* — тем же методом `value`, что и для базового класса (см. стр. 96). При этом объект всегда создаёт копию строки, поданной в качестве аргумента (аналогично тому, как это делает метод `copy_label` для метки), так что можно не беспокоиться о сохранности области памяти, в которой размещалась строка, и использовать её в дальнейшем по своему усмотрению, в том числе освободить.

К недостаткам этих виджетов в сравнении, например, с применением `Fl_Box` и его метки можно отнести невозможность выбора расположения текста: как и в обычных полях ввода, текст всегда располагается вдоль левой границы виджета. Единственное, чем тут можно управлять — это переносом строк, не помещающихся в виджет по горизонтали; как можно догадаться, это делается методом `wrap`, знакомым нам по классу `Fl_Multiline_Input` (см. стр. 97); в действительности этот метод вводится ещё в абстрактном классе `Fl_Input_`, который служит предком классу `Fl_Input`). Виджета, допускающего, например, центрирование выдаваемого текста или его выравнивание по правому краю, но при этом позволяющего выделять текст и копировать его, FLTK не предусматривает.

По умолчанию виджеты вывода выглядят на экране точно так же, как поля ввода, что провоцирует пользователя пытаться редактировать их содержимое. Это можно скорректировать, установив другой тип обрамления и цвет фона. Например, можно сделать виджет «плоским», поставив с помощью метода `box` тип оформления `FL_FLAT_BOX`; этого уже хватит, чтобы не возникало желания начать редактировать выданный текст. Если добавить к этому вызов метода `color(FL_BACKGROUND_COLOR)`, в результате получится виджет, границ которого не видно, а текст появляется как будто прямо в главном окне. Чего точно не стоит делать — это убирать бокс (устанавливать тип оформления `FL_NO_BOX`) или заменять его рамкой. Дело в том, что очистку видимого содержимого все наследники `Fl_Input_` производят перерисовыванием своего бокса, так что в отсутствие бокса при изменениях текста или, например, при его прокрутке вы увидите странные артефакты, наложение старого текста на новый и прочие эффекты, которых нормальная программа проявлять не должна.

В вашей задаче может потребоваться вывести на экран для просмотра большой объём текста с возможностью его прокрутки — например, содержание текстового файла или поток сообщений о происходящих событиях. При этом работать без полос прокрутки может стать совсем неудобно, и появится смысл применить виджет

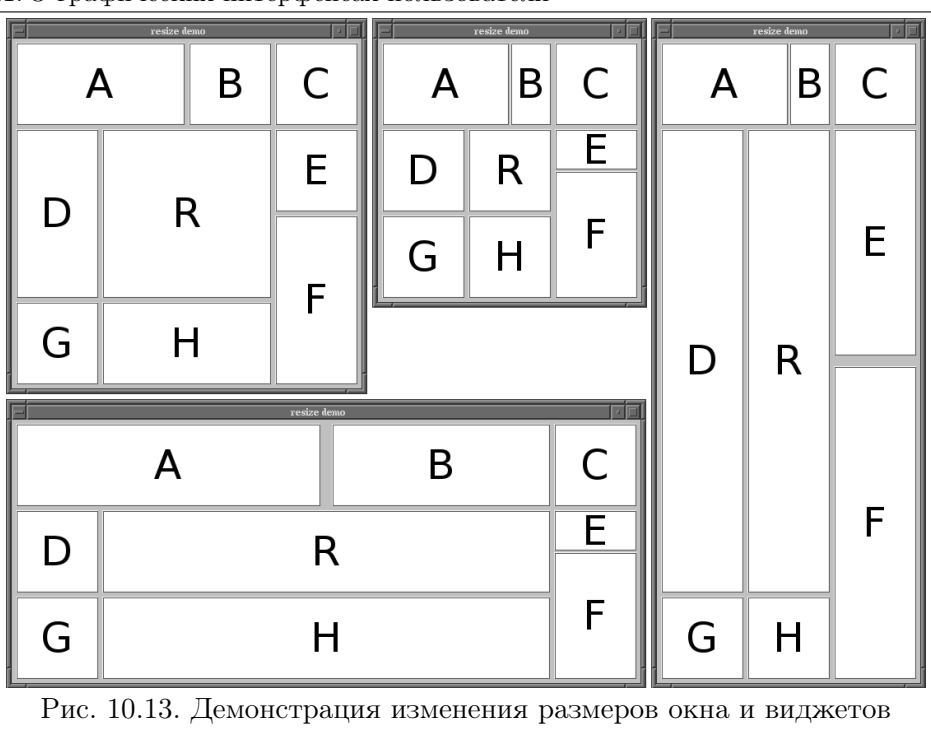

Рис. 10.13. Демонстрация изменения размеров окна и виджетов

`Fl_Text_Display`, который мы уже обсуждали на стр. 98. Напомним, что сам текст в этом случае хранится в объекте класса `Fl_Text_Buffer`; изменения текста в этом «буфере» немедленно становятся видны во всех виджетах, которые с ним связаны. В частности, метод `append` позволяет добавить текст (строку) в конец имеющегося в буфере текста, что оказывается удобно при выводе последовательных сообщений.

10.11.7. Окна, допускающие изменение размера

Все рассмотренные нами до сих пор примеры рисовали на экране окна, не допускающие изменения размера. В большинстве случаев так проще для программиста — не нужно ломать голову над тем, во что превратится выверенное расположение виджетов, когда пользователю придёт в голову схватиться за мышь и начать растягивать и сжимать окошко программы; но на практике такие программы встречаются крайне редко, потому что окна, не допускающие изменения размера, *неудобны для пользователя*. Если же мы позволим пользователю менять размеры окна (что в большинстве случаев попросту обязательно), нам волей-неволей придётся озаботиться тем, как при этом должны измениться (или не измениться) размеры каждого из наших виджетов.

Подход, принятый к этому в библиотеке FLTK, сами авторы библиотеки расписывают как исключительно гибкий, очень простой в понимании, универсальный настолько, насколько это вообще возможно, и вообще просто замечательный со всех точек зрения. Правы они в этом или не очень, предложим читателю рассудить самостоятельно по итогам прочтения этого параграфа.

За возможностью/невозможностью для отдельных виджетов менять размеры по горизонтали и/или вертикали в библиотеке FLTK следят объекты класса `Fl_Group`; напомним, что это виджет, задача которого — содержать другие виджеты. Класс, объекты которого соответствуют окнам — `Fl_Window` — является наследником `Fl_Group` и, как следствие, обладает всей функциональностью своего родителя, включая умение управлять изменением размеров отдельных виджетов.

Принцип, по которому определяется, какие виджеты и вдоль какой оси (или обеих осей) можно растягивать и сжимать, описывается действительно довольно просто. Каждому объекту `Fl_Group` можно задать *один* виджет, для которого допускается изменение размера (англ. *resizable*). Делается это методом `resizable`; параметром ему служит адрес объекта типа `Fl_Widget`. В качестве такого виджета может выступать любой из виджетов, *вложенных* в данную группу, а также и сам виджет группы.

В качестве параметра метода `resizable` можно указать ноль, тогда данная группа не будет допускать изменения размеров, поскольку в ней не будет виджета, допускающего изменение размеров. Если `resizable` для группы объявить *сам объект группы*, то менять размер (и положение) будут *все* входящие в группу виджеты — пропорционально тому, насколько (точнее, во сколько раз) пользователь сжал или растянул окно; вертикаль и горизонталь при этом обрабатываются независимо.

Наиболее интересен случай, когда параметром `resizable` выступает некий виджет, *входящий* в группу. Тогда группа будет способна менять размер, а с виджетами внутри неё будет происходить вот что. Виджет, указанный в качестве `resizable`, будет менять и длину, и ширину; виджеты, имеющие с ним непустое пересечение по горизонтальной оси, будут растягиваться и сжиматься только по горизонтали; виджеты, пересекающиеся с тем, который `resizable`, по вертикальной оси, будут менять только вертикальный размер. Что касается виджетов, не имеющих с `resizable` пересечения ни по горизонтали, ни по вертикали — они свои размеры менять не будут вообще.

На рис. 10.13 показан пример, иллюстрирующий этот принцип; слева вверху расположено окно программы-примера в его оригинальном размере, остальные окна получены изменением размеров исходного окна с помощью мышки. Исходный текст примера находится в файле `fltk/resize.cpp`. В роли `resizable` здесь выступает виджет, расположенный по центру окна, он помечен буквой `R`. Как можно заметить, виджеты `C` и `G`, не имеющие пересечения с `R` ни по горизонтали, ни по вертикали, во всех случаях сохраняют свой оригинальный размер. Виджеты `A`, `B` и `H` могут растягиваться и сжиматься по горизонтали, но высоту (размер по вертикали) сохраняют одну и ту же; с виджетами `D`, `E` и `F` ситуация симметричная — они меняют высоту, но сохраняют неизменную ширину.

Создатели FLTK в документации утверждают, что с использованием этого механизма можно любой набор видимых виджетов заставить менять свои размеры любым способом, какой нам только может прийти в голову — нужно только правильно построить иерархию вложенных друг в друга групп и кое-где, возможно, добавить невидимые виджеты (обычно это `Fl_Box`'ы с типом обрамления `FL_NO_BOX` и без метки). Не будем утверждать, что это действительно так (как не будем и утверждать противоположного), отметим лишь, что в некоторых случаях оказывается довольно сложно сообразить, как именно добиться от виджетов нужного поведения.

Полезно будет иметь в виду несколько основных приёмов. Так, чтобы заставить некий виджет всегда сохранять размер, что бы ни творилось вокруг, достаточно создать для него отдельную группу, в которую будет входить только он сам, и запретить этой группе изменение размеров, вызвав `resizable` от нуля. Чтобы заданный виджет заставить меняться только по горизонтали, а высоту сохранять, можно создать группу, в которую будет входить этот виджет и вместе с ним невидимый `Fl_Box`, расположенный выше или ниже нашего виджета (кстати, можно его расположить даже за пределами координат группы, всё равно его не видно), имеющий такую же, как он, ширину, и при этом *нулевую* высоту; именно этот `Fl_Box` нужно объявить `resizable` для группы. Аналогичным образом можно создать виджет, изменяющий только высоту, но не ширину.

Коль скоро мы столкнулись со случаем, требующим построения иерархии групп виджетов, уместно будет рассказать, как такие иерархии строятся.

Прежде всего напомним, что класс `Fl_Window` унаследован от `Fl_Group`, то есть окно является частным случаем группы виджетов. Каждый объект-виджет обычно принадлежит к той или иной группе либо не принадлежит ни к какой, но последнее имеет смысл только для окон (объектов типа `Fl_Window`), а любой другой виджет, не будучи включённым в группу, окажется бесполезен. Группу, к которой принадлежит виджет, можно узнать, вызвав метод `parent` (без параметров).

Выстраивание иерархии групп несколько облегчает введённое в FLTK понятие *текущей группы*. В каждый момент времени в нашей программе либо есть ровно одна группа, называемая текущей (*current*), либо такой нет ни одной. Если текущая группа есть, все создаваемые виджеты автоматически помещаются в эту группу (становятся её дочерними виджетами). При создании новой группы она автоматически становится текущей. Кроме того, группу можно сделать текущей, вызвав (без параметров) её метод `begin`. Уже знакомый нам метод `end` в действительности делает текущей группу, родительскую для данной, либо оставляет нашу программу без текущей группы, если у той группы, для которой его вызвали, нет родителя. Можно для переключения между группами воспользоваться также статическим методом `Fl_Group::current` — собственно говоря, `begin` просто вызывает `current(this)`, а `end` — `current(parent())`.

Виджет можно «загнать» в заданную группу напрямую с помощью метода `add`, но потребность в этом возникает редко — например, когда набор виджетов меняется во время работы в ответ на действия пользователя или какие-то другие события. В большинстве случаев автоматическое добавление новых виджетов в текущую группу позволяет при построении иерархии ограничиться явным вызовом одного только метода `end`, всё остальное про-

исходит само собой. Для этого иерархию групп начинают строить с главного окна, а сразу после создания объекта группы создают все виджеты, которые должны в неё входить (тогда они оказываются в этой группе автоматически), вызывают для объекта группы метод `end` и продолжают формировать её родительскую группу.

Экспериментируя с любой программой на FLTK, допускающей изменение размеров окна, можно с удивлением заметить, что окно упорно не желает становиться меньше определённого размера. В частности, в примере на рис. 10.13 исходный размер каждого из «квадратиков» составлял 100×100 пикселей, а промежутки между ними — 5 пикселей; как следствие, виджет `R` имел исходный размер 205×205, виджет `A` — 205×100 и т. д. Размер окна, показанный на рисунке в середине верхнего ряда, оказался для нашего примера наименьшим, окно отказывается уменьшаться дальше; сразу же отметим, что размер виджета `R` при этом составил ровно 100×100 пикселей, но это совершенно не потому, что другие виджеты имели такой или «аналогичный» размер.

Происходящее обусловлено одной особенностью поведения объекта окна, которое, хоть и является частным случаем группы, всё же представляет собой случай весьма специфический. В классе `Fl_Window` (именно в нём, а не в `Fl_Group`) предусмотрен метод `size_range`, имеющий аж семь параметров:

```
size_range(int minw, int minh,
           int maxw=0, int maxh=0,
           int dw=0, int dh=0,
           int aspect=0);
```

К счастью, в большинстве случаев нужны только первые два параметра, задающие минимально допустимые размеры окна по горизонтали и вертикали. Оба параметра обязаны быть строго больше нуля; «вырожденные» окна, имеющие нулевую длину и/или ширину, не допускаются. Параметры `maxw` и `maxh` задают максимально возможный размер окна (и если он совпадает с соответствующим минимумом, окно вообще не может менять размер по этой оси), параметры `dw` и `dh` позволяют указать *шаг* изменения размера, т. е. любое изменение размера будет кратно заданному числу (например, это может пригодиться для окна терминала, в котором всегда должно быть целое число знакомест). Параметр `aspect` в действительности логический[36]; если его задать ненулевым (истинным), окну можно будет изменить размер только с сохранением соотношения длины и ширины.

Возвращаясь к странному поведению нашего примера, отметим, что в программе `fltk/resize.cpp` мы вообще не вызывали метод `size_range`, а если его не вызывать, объект окна будет демонстрировать некоторое *поведение по умолчанию*, зависящее от исходных размеров виджета, объявленного как `resizable`. Если по одному из измерений размер этого виджета *меньше 100*, то именно этот размер принимается как минимальный (для виджета, не для всего окна), если же исходный размер *больше 100*, то за минимально допустимый принимается как раз 100. Что, собственно, мы и наблюдали. Интересно, что если размер `resizable` по данной оси равен нулю, то этот ноль принимается заодно и за максимум, то есть по этой оси окно вообще не будет менять размер.

10.11.8. Обработка аргументов командной строки

Программы, взаимодействующие с X Window, обычно обрабатывают целый ряд аргументов командной строки, влияющих на то, к какому X-серверу следует обратиться, окно какого размера создать, в каком месте экрана его расположить и т. д. Библиотека FLTK содержит функции, позволяющие проанализировать командную строку и обработать полтора десятка опций, среди которых `-display`, указывающая, к какому из X-серверов следует подключаться (обычно эта информация берётся из переменной окружения `DISPLAY`); `-geometry`, позволяющая задать положение и размер главного окна (например, `-geometry 200x300` означает размер окна 200 пикселей в ширину, 300 в высоту, `-geometry 200x300+100+150` — окно того же размера, левый верхний угол которого будет располагаться в точке (100;150)); `-bg`, которая меняет основной цвет фона, и другие. Полный список находится в документации к функции `Fl::arg`.

Если ваша программа не нуждается в своих собственных параметрах командной строки, можно пойти самым простым путём: передать сакраментальные `argc` и `argv` параметрами методу `show` главного окна; в отличие от объектов других виджетов, `Fl_Window` поддерживает версию метода `show` с двумя параметрами как раз для этой цели. Выглядит это всё примерно так:

```
int main(int argc, char **argv)
{
    Fl_Window *win = new Fl_Window( /* ... */ );

    // ... подготовка виджетов ...

    win->show(argc, argv);
    return Fl::run();
}
```

Именно так поступают авторы большинства примеров программ из документации к FLTK. К сожалению, как раз такой вариант при всей его простоте почти никогда не годится, ведь если ваша программа вообще обрабатывает параметры командной строки, то почти наверняка среди них будут параметры, обусловленные решаемой задачей и не имеющие никакого отношения к FLTK и вообще подсистеме графического интерфейса.

Коль скоро ваша программа должна обрабатывать свои собственные параметры командной строки, но вы при этом хотите, чтобы FLTK обрабатывала параметры, адресованные ей (а вы этого, по-видимому, хотите, уж во всяком случае `-display` и `-geometry` — это опции, поддержки которых пользователь ожидает от любой оконной программы), у вас есть два основных варианта: организовать просмотр флагов командной строки самостоятельно, а те из них, которые не распознаете, передавать библиотеке FLTK (для этого служит функция `Fl::arg`), либо доверить организацию просмотра библиотеке, с тем чтобы уже *она* обращалась к вашему коду для анализа параметров, которые не смогла распознать. Для этого используется функция `Fl::args` с четырьмя параметрами, одним из которых передаётся адрес вашей callback-функции; её-то FLTK и вызовет.

С рассмотрения этого способа как более простого мы и начнём. Функция `Fl::args` имеет следующий профиль:

```
int Fl::args(int argc, char **argv, int &i,
             Fl_Args_Handler cb = 0);
```

Идентификатором `Fl_Args_Handler` обозначен тип адреса функции, имеющей вид

```
int func(int argc, char **argv, int &idx);
```

(как если бы мы написали «`int (*cb)(int, char**, int&)`»). Назначение первых двух параметров функции `Fl::args` очевидно: это хорошо знакомые нам параметры функции `main`, через которые передаётся командная строка. Через свой третий параметр функция возвращает нам индекс (номер) первого параметра командной строки, который она не смогла или не стала анализировать; исходное значение передаваемой переменной игнорируется. Четвёртым параметром мы, как можно догадаться, передаём свою функцию, отвечающую за обработку наших собственных флагов.

В качестве флагов анализатор рассматривает только аргументы, начинающиеся со знака «-» и содержащие хотя бы ещё один символ. Такой аргумент может иметь параметры, а может их не иметь; в качестве параметров могут рассматриваться один или больше аргументов, следующих в командной строке непосредственно за флагом. Встретив аргумент, начинающийся с «-», анализатор *сначала* (!) вызывает нашу функцию, переданную четвёртым параметром. Параметры `argc` и `argv` нашей функции передаются без изменений; через третий параметр (`idx`) наша функция получит ссылку на переменную, которую анализатор использует как счётчик в цикле просмотра аргументов. Далее у нас есть два варианта: если мы не знаем и не хотим обрабатывать текущий аргумент (тот, который `argv[idx]`), просто возвращаем ноль; если же мы аргумент обрабатываем, то возвращаем количество аргументов командной строки, которые «съели» (сам флаг и, возможно, его параметры); то же самое число следует прибавить к переменной `idx`. Отметим, что таким способом мы можем ввести аргументы, имеющие такие же имена, как у обрабатываемых библиотекой — у нашей функции в любом случае приоритет, если она вернула не ноль, то этот аргумент анализатор уже обрабатывать не будет.

[36] Создатели FLTK предпочитают не использовать тип `bool`; автор книги с определённым удовлетворением хотел бы заметить, что в мире существуют люди ещё более консервативные, чем он сам.

Анализатор прекращает работу в случае, когда все аргументы просмотрены, а также если он встретил аргумент, не начинающийся с минуса, либо состоящий только из него одного, либо начинающийся с минуса, но при этом не съеденный нашей функцией и не знакомый самому анализатору. Последняя ситуация считается ошибочной. Также считается ошибочной ситуация, когда в командной строке *последним* стоит флаг, предполагающий параметр. При возникновении ошибки функция `Fl::args` возвращает ноль, если же всё (с её точки зрения) в порядке, возвращаемое значение совпадает со значением `idx`.

Подчеркнём, что, встретив первый же аргумент командной строки, который сам с минуса не начинается и не является параметром для предшествующего аргумента с минусом, `Fl::args` немедленно возвращает управление, так что даже если дальше будут флаги, она их обрабатывать не станет, и переданная ей callback-функция для этих параметров вызвана не будет.

Как можно заметить, обычного в таких случаях параметра для *пользовательских данных* здесь callback-функция не предусматривает, так что общаться с ней приходится через глобальные переменные. Это можно рассматривать как весомый недостаток подхода с использованием `Fl::args`. Она и сама использует глобальные переменные, ведь надо же где-то сохранить информацию, касающуюся окон и их объектов, которых пока ещё нет.

А теперь самое странное: после всего этого *метод* ***`show`*** *для главного окна всё равно нужно вызвать с параметрами* ***`argc`*** *и* ***`argv`***, причём надежда на то, что анализ уже произведён и повторно проводиться не будет, не оправдывается — метод реально что-то из этих параметров извлекает, дать ему вместо них какую-нибудь ерунду не получится. Спасибо хоть он не ругается на «не свои» параметры — видимо, `Fl::args` всё-таки выставляет какие-то флажки. Если `show` вызвать без параметров, то те из аргументов командной строки, которые должны быть обработаны библиотекой FLTK, обработаны не будут и на работу программы не повлияют.

Для примера напишем программу, похожую на `speak.cpp` (см. стр. 94). Она будет отображать в столбик несколько кнопок (по умолчанию — три), каждую со своей надписью, а при нажатии на кнопку будет эту надпись выдавать на свой стандартный вывод. Надписи будут задаваться параметрами командной строки, расположенными в её конце (то есть после аргументов, начинающихся с минуса). Наша программа будет поддерживать всего один собственный ключ командной строки — «`-count`», параметр которого будет задавать количество кнопок. При этом она будет обрабатывать стандартные ключи вроде `-display`, `-geometry` или `-bg`.

Для начала зададим константы, влияющие на размер и расположение виджетов (собственно кнопок):

```
enum { spacing = 5, button_w = 200, button_h = 40, fontsz = 20 };
```

Поскольку количество рисуемых кнопок будет устанавливаться из callback-функции, не имеющей параметра для передачи пользовательской информации, для хранения этого количества придётся использовать глобальную переменную:

```
static int buttons_count = 3;
```

Анализ того аргумента командной строки, в котором должно быть число, вынесем для удобства в отдельную функцию:

```
static void set_buttons_count(const char *arg)
{
    char *errptr;
    buttons_count = strtol(arg, &errptr, 10);
    if(errptr == arg || *errptr) {
        fprintf(stderr, "Invalid count %s\n", arg);
        exit(1);
    }
}
```

Теперь можно написать функцию, которая послужит нам callback'ом для `Fl::args`. Работать она будет очень просто: если текущий параметр — это строка `"-count"`, функция проверит, что параметров в командной строке достаточно (должен существовать как минимум ещё один после `"-count"`), и «съест» текущий параметр и следующий за ним, установив при этом нужное значение переменной `buttons_count`; во всех остальных случаях она сообщит вызывающему, что никакие параметры не обработала (ничего не съела), вернув ноль:

```
static int cmdline_callback(int argc, char **argv, int &idx)
{
    if(0 == strcmp(argv[idx], "-count")) {
        if(argc < idx + 2) {
            fprintf(stderr, "Count (number) expected\n");
            exit(1);
        }
        set_buttons_count(argv[idx+1]);
        idx += 2;
        return 2; /* "съели" текущий параметр и следующий */
    }
    return 0;     /* других флагов мы не знаем */
}
```

В главной функции мы вызовем `Fl::args`; она вернёт через свой третий параметр индекс первого из аргументов командной строки, не начинающегося с минуса и не являющегося параметром предшествующего аргумента, а в нашем случае это означает, что нам попалась первая из строк, которые должны служить метками нашим кнопкам. Остаток массива `argv` мы используем просто как массив текстовых меток. Выглядеть это будет примерно так:

```
int main(int argc, char **argv)
{
    int arg_idx, res;
    res = Fl::args(argc, argv, arg_idx, cmdline_callback);
    if(!res) {
        fprintf(stderr, "Malformed command line\n");
        return 1;
    }
    int win_w = button_w + spacing * 2;
    int win_h = (button_h + spacing) * buttons_count + spacing;
    Fl_Window *win =
        new Fl_Window(win_w, win_h, "command line demo");
    int i;
    int y = spacing;
    for(i = 0; i < buttons_count; i++) {
        const char *msg =
            arg_idx + i < argc ? argv[arg_idx + i] : "Not set";
        Fl_Button *b =
            new Fl_Button(spacing, y, button_w, button_h, msg);
        b->labelsize(fontsz);
        b->callback(say_callback, (void*)msg);
        y += button_h + spacing;
    }
    win->end();
    win->show(argc, argv);
    return Fl::run();
}
```

Функция `say_callback` имеет точно такой же вид, как и в предыдущих примерах. Полностью текст нашей программы вы найдёте в файле `fltk/cmdline.cpp`.

Альтернативой использованию `Fl::args` служит подход, при котором мы сами организуем цикл просмотра параметров командной строки, в теле которого, обнаружив неизвестный нам ключ, начинающийся с минуса, отдаём его для обработки библиотеке FLTK через функцию `Fl::arg` (обратите внимание на отсутствие буквы `s` на конце). Профиль этой функции таков:

```
int Fl::arg(int argc, char **argv, int &idx);
```

С первыми двумя параметрами всё очевидно, сюда передаются наши `argc` и `argv` без изменений; параметр `idx` — ссылка на счётчик, хранящий индекс текущего рассматриваемого аргумента. Работает функция примерно так же, как в предыдущем примере работал наш callback: если обрабатывать аргумент она не стала, то возвращаемое значение будет равно нулю, если же аргумент обработан, она вернёт число `1` или `2` в зависимости от того, один или два аргумента командной строки она «употребила», и на такое же число будет увеличено значение переменной `idx`. Надо только обязательно проследить, была ли функция `Fl::arg` вызвана хотя бы один раз; тогда метод `show` будет знать, что анализ командной строки произведён головной программой. Если же ни одного параметра, относящегося к компетенции FLTK, обнару-

жено не было, следует этот факт запомнить и потом вызвать функцию `show` без параметров, чтобы она не ругалась на «неизвестные» параметры, не зная, что вы командную строку уже проанализировали.

Разбирать пример такого анализа командной строки мы не будем — надеемся, что читатель теперь справится сам; но на всякий случай отметим, что пример, аналогичный предыдущему по своей работе, но реализованный через `Fl::arg`, приведён в файле `fltk/cmdline2.cpp`.

10.11.9. Обзор нерассмотренных возможностей

Объём книги не позволяет нам рассмотреть возможности FLTK подробнее, чем мы это уже сделали, но на случай, если читателю потребуется что-то из оставшегося за кадром, попытаемся провести краткий обзор.

Пожалуй, одна из самых важных сущностей, не поместившихся в нашу книгу — это подсистема реакции на события. С её помощью можно перехватить обработку нажатий на клавиши клавиатуры и кнопки мыши, перемещение мыши в пределах окна нашей программы и многое другое. Если что-то подобное кажется подходящим для вашей программы, начните изучение этой подсистемы с констант перечислимого типа `Fl_Event` и функций `Fl::add_system_handler`, `Fl::add_handler` и `Fl::event_dispatch`, а также виртуального метода `handle`, объявленного в классе `Fl_Widget`.

Кроме того, хотя бы краткого упоминания заслуживает целый ряд не обсуждавшихся нами виджетов. Так, мы ни разу не пользовались классическими меню, хотя это очень важный элемент графического интерфейса. В FLTK для создания меню предусмотрен базовый класс `Fl_Menu_`, от которого унаследовано три потомка: `Fl_Menu_Bar` — горизонтальное меню с выпадающими вниз подменю, `Fl_Menu_Button` — кнопка, при нажатии на которую вниз выпадает меню, и `Fl_Choice` — виджет, похожий на предыдущий, предназначенный для выбора из нескольких доступных вариантов; текущий вариант показывается в окошке, снабжённом кнопкой, по которой показывается меню со списком других доступных вариантов. Ещё один довольно полезный виджет — `Fl_Tree`. Это что-то вроде меню с древовидной структурой и возможностью скрывать и раскрывать отдельные ветки.

Весьма интересен виджет `Fl_Scroll`, позволяющий создать некое виртуальное пространство, содержащее виджеты и способное прокручиваться по горизонтали и вертикали; виджет использует полосы прокрутки, реализованные классом `Fl_Scrollbar`. В сочетании с `Fl_Scroll` полезен виджет `Fl_Pack`, представляющий собой группу, т. е. являющийся потомком класса `Fl_Group`. Эта версия группы автоматически располагает дочерние виджеты в вертикальный или горизонтальный ряд. Иногда виджет `Fl_Pack` бывает полезен и сам по себе.

Для индикации постепенного исполнения каких-нибудь длительных вычислений можно показать пользователю соответствующую «полоску» (*progress bar*) с помощью виджета `Fl_Progress`.

Если вам нужно, чтобы пользователь задал значение с плавающей точкой (например, какой-нибудь коэффициент), для этого можно воспользоваться одним из полутора десятков потомков класса `Fl_Valuator`; здесь открывается довольно широкий простор для фантазии — помимо привычных слайдеров, можно нарисовать что-то вроде круглого циферблата со стрелкой или трёхмерного колёсика, и т. п.

Расположить информацию в виде таблицы поможет `Fl_Table`; здесь при желании можно расположить по виджету в каждой клетке таблицы, но можно ограничиться простым текстом. От этого класса часто наследуют потомков, переопределяя в них виртуальные функции — в документации рекомендовано поступить именно так.

Класс `Fl_Tile` позволяет расположить в окне несколько виджетов (обязательно «встык», то есть дочерние виджеты должны располагаться без перекрытий и без пустого места между ними) и предоставить пользователю возможность менять их размеры, «таская» мышкой границы между виджетами.

Класс `Fl_Tabs` реализует привычное многим пользователям «окно с вкладками». На каждой вкладке располагается свой собственный набор дочерних виджетов. Несколько похож на него виджет `Fl_Wizard` — только здесь пользователю не предоставляется возможности переключать вкладки, это делается только программно; обычно он используется для организации «многоступенчатых» диалогов с пользователем, которые по-английски так обычно и называются *wizards* (адекватного русского перевода автор этих строк не видел).

Класс `Fl_Tooltip`, состоящий только из статических функций, позволяет управлять «подсказками» (*tooltips*), которые появляются на экране при наведении курсора мыши на виджет. Сам текст этих подсказок задаётся методами класса `Fl_Widget` — `tooltip` и `copy_tooltip`.

FLTK предусматривает ряд предопределённых диалоговых окон, позволяющих быстро задать пользователю тот или иной вопрос или просто выдать сообщение. Все эти окна являются ***модальными***, то есть на время их работы все прочие виджеты вашей программы блокируются. Окно выдаётся на экран вызовом соответствующей функции; так, для выдачи информационного или предупреждающего сообщения используются функции `fl_message` и `fl_alert`, при этом пользователь увидит диалоговое окно с заданным текстом и единственной кнопкой Ok; предусмотрено несколько альтернативных вариантов для функции `fl_message`, позволяющих снабдить окно иконкой, открыть окно там, где сейчас курсор мыши и т. д. Функция `fl_ask` выдаёт окно с заданным текстом и двумя кнопками — Yes и No, и возвращает `0` или `1` в зависимости от того, какую из кнопок пользователь в итоге нажал. Функция `fl_choice` позволяет вывести до трёх кнопок (возвращаемое значение, соответственно, будет от `0` до `2`). Все перечисленные функции предполагают формирование сообщения с помощью форматной строки и дополнительных аргументов подобно стандартной функции `printf`.

Функция `fl_input` выдаёт диалоговое окно с полем для ввода текста; `fl_password` делает то же самое, только текст вводится вслепую. Функции `fl_file_chooser` и `fl_dir_chooser` выдают диалоговое окно, предлагающее пользователю выбрать файл или директорию на диске, функция `fl_color_chooser` — выбрать цвет. Меняя значение глобальных переменных `fl_yes`, `fl_no`, `fl_ok`, `fl_cancel` и `fl_close`, вы можете изменить тексты кнопок во всех «стандартных» диалогах (кроме тех, для которых текст кнопок задаётся параметрами формирующей их функции).

Отдельного упоминания заслуживает иерархия, сформированная абстрактным классом `Fl_Image` и его потомками, такими как `Fl_Bitmap`, `Fl_PNG_Image` и другими. Объекты этих классов позволяют разместить в памяти изображение в одном из поддерживаемых форматов; такое изображение потом можно назначить практически любому виджету — это делается методами `image` и `deimage`.

Даже с учётом нашего обзора мы не рассмотрели и пятой части всего, что есть в библиотеке FLTK. Как всегда в таких случаях, предлагаем читателю самостоятельно обратиться к технической документации.

10.11.10. FLTK и парадигма ООП

Поскольку нашим основным предметом рассмотрения в этом томе книги остаются парадигмы, посвятим отдельный параграф обсуждению FLTK с точки зрения объектно-ориентированного программирования.

Несмотря на то, что FLTK — это библиотека классов языка Си++ и в ней активно используются иерархии наследования, её архитектура спроектирована, если можно так выразиться, «не совсем объектно-ориентированно». В пользу этого странного на первый взгляд утверждения можно привести два основных аргумента. Во-первых, в FLTK очень часто встречается ситуация, когда базовый класс реализует всю функциональность своих будущих потомков, а сами потомки только выбирают, какой частью этой функциональности воспользоваться. Так, изрядная доля возможностей всех виджетов реализована в базовом классе `Fl_Widget`, при том что многие из этих возможностей в заметной части виджетов не нужны (и даже не доступны). Ещё сильнее это проявляется с потомками класса `Fl_Input_` — там в базовом классе реализовано буквально всё, включая однострочность и многострочность, а потомки вообще ничего к реализации не добавляют, только выбирают режим работы.

Второй аргумент несколько сложнее. Дело в том, что ***callback-функции***, которые, как мы видели, в FLTK применяются на каждом шагу, **для объектно-ориентированного программирования — приём чуждый**.

В самом деле, обычный способ работы с callback-функциями состоит в том, что некоторой «обслуживающей» подсистеме передаётся адрес функции, которую нужно вызывать при наступлении определённого события, и адрес области памяти, содержащей *пользовательские данные* (чаще всего — в виде `void*`), который нужно передать функции одним из параметров при её вызове. Callback-функция, будучи вызвана, сама преобразует адрес типа `void*` в то, чем он там на самом деле является, и делает свою работу, имея доступ ко всем нужным данным.

Используя средства объектно-ориентированного программирования, мы можем добиться *совершенно того же эффекта*, передав обслуживающей подсистеме *адрес объекта, у которого есть виртуальная функция*. Роль пользовательских данных в этом случае исполнит сам объект, вместо callback-функции обслуживающая подсистема будет вызывать его (объекта) виртуальный метод, который мы можем сделать каким угодно — для этого достаточно унаследовать от базового класса (адрес объекта которого от нас ожидают) свой собственный класс-потомок со своей собственной версией виртуальной функции. Согласитесь, выглядит намного естественнее: вот тебе объект, когда что-то случится — сообщи ему. И ведь, главное, никаких бестиповых указателей и явных преобразований типа адреса! Вместо этого работает преобразование адреса по закону полиморфизма (от потомка к предку), причём для *неявного* указателя `this`, который при желании можно вообще выкинуть из головы.

В применении к нашим виджетам всё это должно было бы выглядеть так: вместо того, чтобы описывать странные callback-функции, как мы это делали в §§10.11.2–10.11.4, мы бы создали *порождённый класс*, унаследовав его от класса библиотечного виджета, переопределили бы некий виртуальный метод, который в классе-предке вызывается при наступлении события (или, возможно, несколько разных методов, соответствующих разным событиям), а все свои пользовательские данные хранили бы в полях своего объекта-наследника. И всё — никаких callback'ов.

На этом месте можно сообразить, что вообще-то никто не мешает нам поверх имеющихся в FLTK классов создать промежуточный слой, который обеспечит именно такой, как мы хотим, подход к работе. Покажем это на примере кнопок — класса `Fl_Button`. Базовый класс, который мы создадим на его основе, будет предполагать создание наследников с целью переопределения виртуальных функций. Поскольку этот класс уже не является частью библиотеки FLTK, поддерживать тот же стиль именования нам не нужно; примем соглашение, что имена всех классов нашего промежуточного слоя будут начинаться с префикса «`OOFL`» и не будут содержать символа подчёркивания. В примере мы опишем только один такой класс, но смысл происходящего именно в том, чтобы в «боевой» программе описать аналогичные классы для всех используемых виджетов; эти классы как раз и составят искомый промежуточный слой. Для именования методов, вводимых в новых классах, мы будем придерживаться стиля, который использовали при рассказе о Си++ — каждое слово, входящее в идентификатор, будем начинать с заглавной буквы.

Для иллюстрации сказанного переделаем пример `fltk/speak.cpp`, который мы разбирали в §10.11.2. Новая версия этой программы будет делать абсолютно то же самое, но с архитектурной точки зрения (и в плане используемых парадигм) мы построим её несколько иначе.

Прежде всего напишем класс `OOFLButton`, который будет представлять тот же виджет, что и класс `Fl_Button`, но с иной организацией работы. Естественно, наш класс мы *унаследуем* от `Fl_Button`. Конструктор нашего класса будет сразу же устанавливать callback-функцию, вызвав метод `callback` класса-предка; установленная callback-функция будет вызывать виртуальный метод, который мы назовём `OnPress`. В классе `OOFLButton` этот метод не будет делать ничего — мы снабдим его пустым телом; потомки нашего класса реализуют свою функциональность, переопределив метод `OnPress` по-своему. Параметров метод принимать не будет — всё, что потребуется его версиям в классах-потомках, можно будет разместить в полях объектов.

Может показаться, что логичнее было бы сделать метод `OnPress` чисто виртуальным. На самом деле это не совсем так. К чисто виртуальным методам мы прибегаем в случаях, когда в принципе не знаем и не можем знать, как нам отреагировать на сообщение (в данном случае — на сообщение о нажатии кнопки); но это не наш случай. Иногда бывает полезно создать кнопку, которая ничего не делает, и именно такая реакция — ничего не делать — оказывается вполне логичной для базового класса.

Остаётся решить, как сделать callback-функцию. Обычные методы класса для этой цели не годятся из-за наличия у них неявного параметра `this`, а оставлять эту функцию внешней по отношению к классу не хочется — она очевидно представляет собой деталь реализации, которая не должна беспокоить никого, кроме нашего класса. Выход из положения есть: мы можем использовать ***статический*** метод класса (см. §10.4.26). Указателя `this` у статических методов нет, так что фактически это простые функции с хитрым именем и правилами видимости. Функцию мы назовём `CallbackFunction` и уберём в приватную часть класса. Параметры этой функции нам придётся сделать точно такие же, как у обычных callback-ов, чтобы при вызове метода `callback` не возникало несоответствия типов. При этом параметр `user` мы использовать не будем — в роли пользовательских данных по нашему замыслу должен выступать сам объект виджета, точнее — некий наследник нашего `OOFLButton`, а его адрес и так будет передан в `CallbackFunction` её первым параметром.

Поскольку мы вводим новый виртуальный метод, не лишним будет описать явным образом виртуальный деструктор, пусть даже нам в нём нечего чистить, так что его тело окажется пустым. В нашем случае это ни на что не повлияет, именно такой деструктор был бы сгенерирован неявно. Но в общем случае мы можем не знать, есть ли в классе-предке таблица виртуальных методов; если её там нет, то нет и виртуального деструктора, а нам он, очевидно, нужен — как всегда в классах, имеющих хотя бы один виртуальный метод.

Полностью описание класса `OOFLButton` окажется намного короче, чем рассказ о нём. Тела всех методов здесь тривиальны, так что мы оставляем их в заголовке класса:

```
class OOFLButton : public Fl_Button {
public:
    OOFLButton(int x, int y, int w, int h, const char *lb)
        : Fl_Button(x, y, w, h, lb)
        { callback(CallbackFunction, 0); }
    virtual ~OOFLButton() {}
    virtual void OnPress() {}
private:
    static void CallbackFunction(Fl_Widget *w, void *user)
        { static_cast<OOFLButton*>(w)->OnPress(); }
};
```

Этот класс не содержит никакой специфики, привязанной к решаемой задаче; его можно использовать в любой программе, где требуется виджет «кнопка». В нашем примере он описан прямо в главном файле (поскольку это единственный файл программы), но в реальном проекте подобные классы лучше выносить в отдельные модули.

Имея в своём распоряжении класс `OOFLButton`, мы можем приступить к решению поставленной задачи. Кнопок у нас предполагается два вида: те, нажатие на которые приводит к печати сообщения, и та, которая по нажатию завершает работу программы. Кнопки первого вида мы опишем классом `SayButton`, кнопку второго вида — классом `ExitButton` (поскольку это уже классы, специфичные для задачи, мы не будем снабжать их имена префиксом `OOFL`, который договорились использовать для нашего объектно-ориентированного слоя).

Вспомним, что в исходном примере мы всем кнопкам назначали размер шрифта метки, равный константе `font_size`. Для краткости программы сделаем это в конструкторах наших классов, для чего описание `enum`'а, вводящего константы, придётся поместить в программе *до* описания классов `SayButton` и `ExitButton`. Коль скоро одну из этих констант мы всё равно используем, можно заметить, что вообще-то задействовать их можно все, ведь у всех трёх наших кнопок совпадает ширина, высота и горизонтальная координата, меняются только координата по вертикали, текст метки, а для кнопок `SayButton` ещё и текст выдаваемого сообщения. Итак, введём константы:

```
enum { spacing = 5, button_w = 200, button_h = 40, fontsz = 20 };
```

Все общие свойства наших трёх кнопок мы зададим промежуточным классом, который назовём `ButtonCommon`. Собственно говоря, это будет пример «наследования ради конструктора», которому мы посвятили §10.6.8:

```
class ButtonCommon : public OOFLButton {
public:
    ButtonCommon(int y, const char *lb)
        : OOFLButton(spacing, y, button_w, button_h, lb)
        { labelsize(fontsz); }
};
```

Классы `SayButton` и `ExitButton` теперь окажутся совсем простыми (на всякий случай отметим, что тексты выдаваемых сообщений в нашей программе всегда задаются строковыми литералами, так что копировать их не требуется, можно ограничиться запоминанием адреса):

```
class SayButton : public ButtonCommon {
    const char *msg;
public:
    SayButton(int y, const char *lb, const char *amsg)
        : ButtonCommon(y, lb), msg(amsg) {}
    virtual void OnPress() { printf("%s\n", msg); }
};
class ExitButton : public ButtonCommon {
public:
    ExitButton(int y) : ButtonCommon(y, "Quit") {}
    virtual void OnPress() { exit(0); }
};
```

От функции `main`, которая в примере `fltk/speak.cpp` была довольно громоздкой, в новой версии, прямо скажем, мало что осталось:

```
int main(int argc, char **argv)
{
    int win_w = button_w + spacing * 2;
    int win_h = button_h * 3 + spacing * 4;
    Fl_Window *win = new Fl_Window(win_w, win_h, "buttons demo");
    new SayButton(spacing, "Say hello", "Hello, world!");
    new SayButton(2 * spacing + button_h,
                        "Say goodbye", "Goodbye, world!");
    new ExitButton(3 * spacing + 2 * button_h);
    win->end();
    win->show();
    return Fl::run();
}
```

Полный текст примера читатель найдёт в файле `fltk/oo_speak.cpp`. В сравнении с исходной версией файл стал длиннее на восемь строк, но такое сравнение не совсем корректно: класс `OOFLButton` по смыслу библиотечный, так что его описание не должно находиться в тексте нашей программы — так же, как в нём нет классов библиотеки FLTK. Если же вычесть 11 строк, занятых описанием этого класса, и пустую строку, отделяющую его от остальной программы, получится, что текст стал короче, пусть даже всего на четыре строчки (при общей длине около 50 строк это не так уж мало). Ну а что программа стала яснее благодаря декомпозиции — это вообще вне всякого сомнения. Во-первых, тело функции `main` теперь состоит всего из девяти строк, против 17 строк кода и трёх пустых строк (вынужденно вставленных для ясности) в оригинальной версии. Во-вторых, теперь нам не нужно передавать данные из одного места программы в другое, сначала преобразуя их адрес к типу `void*`, а потом обратно к типизированному адресу. В небольшой программе уследить за корректностью таких преобразований несложно, но если программа перевалит за несколько тысяч строк (и это, заметим, совсем не так много), подобные вольности могут стать причиной изрядной головной боли.

10.11.11. (*) Если нужен свой главный цикл

Как и большинство библиотек виджетов, FLTK создана в предположении, что, коль скоро в программе вообще используется графический интерфейс, весь мир должен вращаться исключительно вокруг него; это выражается в числе прочего в том, что библиотека сама организует главный цикл и не предусматривает штатной возможности оставить его организацию головной программе. Откуда у разработчиков библиотек такая железобетонная уверенность в абсолютно исключительной важности именно подсистемы графического интерфейса — не вполне понятно, ведь можно легко представить себе программу, которая вообще не всегда запускает графический интерфейс — например, делает это только при указании какого-нибудь ключа в командной строке.

Так или иначе, что делать, если ваша программа активно работает с источниками событий, отличными от графической подсистемы? Сложно, конечно, представить себе TCP-сервер с графическим интерфейсом (хотя бывает и такое), но если вашей программе приходится много работать с сетью в роли клиента, то и дело обращаясь к разным серверам, всю её «сокетную» часть следует, несомненно, реализовывать в событийно-управляемом стиле, *особенно* если она снабжена ещё и графическим интерфейсом.

В принципе, можно смириться с тем, что FLTK узурпировала главный цикл, благо она предусматривает анализ событий на дескрипторах, указанных пользователем (программистом), и добавление пользовательских тайм-аутов; когда соответствующее событие произойдёт, FLTK вызовет указанную вами callback-функцию. Если такой вариант вас устраивает, посмотрите документацию на функции `Fl::add_fd`, `Fl::remove_fd`, `Fl::add_timeout`, `Fl::remove_timeout` и `Fl::repeat_timeout`.

К сожалению, компромиссы годятся отнюдь не всегда; да и вообще, главный цикл — забота головной программы, и организовать его именно в головной программе будет правильнее в идейном плане, особенно если вы это умеете[37]. FLTK не предусматривает *штатного* (документированного) способа взаимодействия с главным циклом, построенным вне её, но, как оказалось при внимательном изучении вопроса, добиться корректной работы в такой конфигурации довольно просто.

Общая идея тут в том, чтобы самостоятельно установить связь с X-сервером, передать полученный дескриптор сокета библиотеке (увы, если позволить ей связаться с X-сервером, как она это обычно делает, дескриптор из неё потом не добыть), в главном цикле отслеживать на этом дескрипторе готовность к чтению, и если таковая появилась — вызывать метод FLTK, представляющий собой точку входа в её диспетчер событий.

Итак, описываем указатель типа `Display*`, используемый библиотекой Xlib (по идее для этого надо бы подключить заголовочник `X11/Xlib.h`, но если вы используете FLTK, то он у вас уже подключён — косвенно, из заголовочных файлов самой FLTK) и используем функцию `XOpenDisplay` для соединения с X-сервером:

```
Display *disp;
disp = XOpenDisplay(0);
```

Обязательно проверьте, что `XOpenDisplay` вернула ненулевой адрес — ноль здесь свидетельствует об ошибке, и такая ошибка совсем не редкость (например, можно случайно попытаться запустить графическую программу на удалённой машине в сеансе ssh, не поддерживающем «проброс» соединения с X-сервером). Если всё в порядке и соединение успешно установлено, мы можем узнать дескриптор его сокета:

```
int fd;
fd = ConnectionNumber(disp);
```

Сообщаем библиотеке FLTK, что с X-сервером мы уже связались и ей этого делать не нужно:

```
fl_open_display(disp);
```

Всё, мы готовы к работе. Теперь добавляем дескриптор `fd` к множеству тех дескрипторов, на которых мы отслеживаем готовность к чтению (например, в программе, которую мы рассматривали в главе 10.10, мы могли бы для этого воспользоваться наследником класса `FdHandler`) и при наступлении такой готовности вызываем функцию `Fl::wait(0.0)`; параметр `0.0` означает «вернуть управление немедленно, не дожидаясь ничего» — в этом случае FLTK обработает события, которые уже пришли, больше ничего делать не станет, и это именно то, что требуется.

К сожалению, как всегда при использовании недокументированных возможностей, не исключено, что изложенное в этом параграфе не будет работать со следующими версиями FLTK; но, возможно, в них всё-таки появится поддержка работы с внешним главным циклом. На всякий случай упомянем ещё функцию `Fl::flush`, вызываемую без параметров, которая «вытесняет» все накопившиеся изменения, связанные с виджетами, и принудительно отправляет соответствующую информацию X-серверу через сокет. Если верить документации, функция `Fl::wait` сама вроде бы должна вызывать `Fl::flush`, но есть сведения, что в некоторых версиях FLTK этого не происходит.

Си++: что дальше

Мы ограничились изучением языка Си++ как такового, полностью проигнорировав его стандартную библиотеку, а сам язык рассматривали в том виде, в котором он существовал до появления стандартов, и даже классические возможности Си++ рассмотрели не все: за рамками книги остались множественное наследование, пространства имён, указатели на методы классов и многое другое.

Несомненно, при приёме на работу едва ли не в любую коммерческую организацию от вас на собеседовании потребуют знания STL. Программирование на Си++ без STL не только возможно, но и позволяет получать более эффективные и качественные программы, вот только вам вряд ли удастся убедить вашего работодателя (будущего начальника) отказаться от STL, поскольку он, скорее всего, относится к числу программистов, изучавших Си++ после 1999 года. Более того, с неплохой вероятностью от вас потребуется владение возможностями Си++, которые мы оставили за кадром, в том числе сомнительными новшествами, которые внесли в язык пресловутые стандартизаторы.

Если такое собеседование предстоит вам в течение ближайших двух недель, то у вас, очевидно, нет иного выхода, кроме как немедленно обрести недостающие знания. Для изучения STL можно воспользоваться практически любой книгой по Си++ (есть даже книги, специально посвящённые

[37] Конечно, для тех, кто падает в обморок при виде вызова `select`, возможность спихнуть построение главного цикла на библиотеку — несомненное благо, но вряд ли такой персонаж дочитает нашу книгу до этого места.

STL) и, конечно, компьютером, поскольку без практических навыков никакие знания в области программирования не будут ничего стоить. Про возможности из «новых стандартов» тоже не написал разве что ленивый; здесь стоит отметить, что чем новее стандарт, тем меньше вероятность, что от вас потребуют владения возможностями из него, так что C++11, возможно, стоит посмотреть подробно, C++14 пробежать «по верхам», а C++17 и тем более C++20 вообще проигнорировать; кстати, если работодатель потребует применять C++17, стоит подумать, нужен ли вам такой работодатель.

Если же немедленно устраиваться на работу программистом в ваши планы пока не входит, будет лучше не браться за изучение STL и не использовать его по крайней мере до тех пор, пока вы не напишете на Си++ одну-две практически применимые программы; под таковыми следует понимать программы, которые реально использует кто-то кроме их автора. После этого можно будет сказать, что основы языка Си++ вы знаете и к изучению STL подойдёте осознанно и с достаточным для этого опытом. Что касается новшеств от стандартизационного комитета, то ни одно из них не сделало язык лучше — но чтобы это оценить, опять-таки нужен опыт работы с классическими средствами Си++ и вообще опыт программирования. Пока у вас есть такая возможность, не обращайте внимания на «стандарты». Чем позднее вам придётся столкнуться с безумными стандартами Си++ и его кошмарной стандартной библиотекой, тем целостнее будет ваше восприятие происходящего.

В качестве дополнительного чтения можно посоветовать книгу Джеффа Элджера, которая в оригинале называется «C++ for real programmers», а в русском переводе вышла в серии «Библиотека программиста» под заголовком «Си++» [10]; куда при переводе делись остальные слова из оригинального названия — вопрос к издателям. Если вам удастся понять эту книгу, вас останется только поздравить: вы действительно поняли, что такое Си++. Стоит отметить, что Элджер ни словом не обмолвился про STL, хотя к тому времени STL уже существовал.

Часть 11

Неразрушающие парадигмы

Все языки программирования, рассматривавшиеся нами ранее, так или иначе предполагали существование некоего *состояния*, которое *изменяется с течением времени* в результате *модифицирующих действий*; парадигмы, не предполагающие модификации состояния, — в основном функциональное программирование — мы упоминали и даже приводили примеры программ, написанных в функциональном стиле, но делали это с помощью языков, которые такой подход если и допускают, то неохотно.

Эту часть книги мы полностью посвятим языкам программирования, в которых модифицирующие действия в лучшем случае «не в почёте», их даже называют не модифицирующими, а ***разрушающими***. Больше того, модификации среды выполнения могут и вовсе оказаться невозможны. Так, в Лиспе и его более молодом диалекте, который называется Scheme[1], привычное нам присваивание существует, но его обычно предпочитают не использовать; в Прологе и Хоупе присваивание начисто отсутствует. В Прологе всё же предусмотрены разрушающие действия в виде побочных эффектов при вычислениях — это, во-первых, ввод-вывод, и во-вторых, *модификация базы данных*. Что касается Хоупа, то он относится к немногочисленной категории *ленивых языков*, в которых выражение вычисляется лишь тогда, когда кому-то потребовалось его значение, и не раньше; в таких условиях побочные эффекты выражений оказываются вообще недопустимы, поскольку нельзя (невозможно!) предсказать, в какой последовательности они проявятся и проявятся ли вообще.

Чтобы программировать на этих языках, программисту, привыкшему к традиционной императивной модели, приходится буквально вывернуть собственное мышление наизнанку. Кто-то преодолевает этот барьер легко, кто-то тяжело, для кого-то он и вовсе оказывается слишком высок. Одно можно сказать с уверенностью: обретённая в итоге гибкость мышления не раз сослужит вам службу, даже если вы никогда не столкнётесь с «неразрушающими» языками на практике.

11.1. Язык Лисп и его S-выражения

11.1.1. Немного истории

Язык Лисп[2] относится к числу «реликтов» — это один из немногих языков программирования, появившихся ещё в 1950-е годы. Среди языков, используемых в наши дни, старше Лиспа только Фортран; впрочем, оба языка за свою долгую жизнь, естественно, претерпели существенные изменения.

Довольно интересна история появления первой работающей реализации Лиспа. Его автор — Джон Маккарти — начал размышлять над основными принципами будущего языка программирования в 1956 году в рамках исследовательского проекта, посвящённого искусственному интеллекту. Целью Маккарти была возможность машинной обработки «символических выражений», таких как математические формулы. Идею использования *списков* ему подсказал язык IPL, разработанный А. Ньюэллом и Г. Саймоном. Летом 1958 года Маккарти опробовал списки в деле, попытавшись применить FLPL — реализацию списков для Фортрана — к задаче символьного дифференцирования, т. е. взятия производной по заданной формуле функции. Как утверждает сам Маккарти, «дифференцирующая программа не была реализована тем летом, поскольку FLPL не позволяет ни условных выражений, ни рекурсивного использования подпрограмм» [11].

В том же году была образована исследовательская группа под руководством Маккарти и Марвина Мински, но Лисп ещё некоторое время оставался набором разрозненных подпрограмм, реализованных на языке ассемблера для компьютера IBM 704. Следующее значительное событие в истории Лиспа потребует некоторых пояснений.

В теории алгоритмов активно эксплуатируется понятие ***универсальной машины Тьюринга*** (УМТ) — это такая машина Тьюринга, на ленту которой можно записать в некоторой нотации другую (произвольную) машину Тьюринга $\mathcal{M}$ и некоторое слово $\mathcal{W}$, и результатом работы УМТ станет результирующее слово, как если бы машину $\mathcal{M}$ применили к слову $\mathcal{W}$ в качестве входного. Иными словами, УМТ — это ***интерпретатор*** машин Тьюринга, который сам написан в виде машины Тьюринга. Известны довольно компактные реализации УМТ, но их достаточно тяжело понять; сложность там вытеснена в *нотацию*: чтобы воспользоваться компактной реализацией УМТ, ту машину Тьюринга, которую нужно интерпретировать, придётся для начала представить в виде входного слова для УМТ, и именно это представление потребует больших усилий. С математической точки зрения это совершенно не важно, главное — точно знать, что соответствующее преобразование *существует*, ведь реально выполнять его для сколько-нибудь серьёзной машины Тьюринга всё равно никому не придётся.

Аналогичное понятие — «универсальная функция» — есть и в теории вычислимых функций. Маккарти в какой-то момент поставил перед собой задачу продемонстрировать возможности разрабатываемой системы, реализовав универсальную функцию; для этого потребовалась некая формальная нотация, представляющая как ***данные*** (в роли которых выступали списки), так и функции. Именно тогда Маккарти придумал скобочный синтаксис, который сейчас как раз и считают «языком Лисп». Довольно интересен тот факт, что реализовывать Лисп в таком виде на вычислительных машинах и использовать его для реального создания программ Маккарти не собирался — новую нотацию он считал явлением сугубо теоретическим, вроде той же машины Тьюринга. Так или иначе, основываясь на уже реализованных функциях и используя списочное представление программы и данных, Маккарти смог написать (на бумаге, не на компьютере) функцию EVAL.

Далее Стивен Рассел — один из студентов, входивших в исследовательскую группу — предложил перевести эту «теоретическую» реализацию на язык ассемблера, как это делалось с другими функциями, с тем чтобы получить реализацию языка Лисп целиком. Маккарти отнёсся к идее скептически, но Рассела это не остановило — он действительно написал такую реализацию, получив интерпретатор Лиспа, ставший, кстати, первым в истории интерпретатором языка программирования. Заодно скобочная нотация записи списков, придуманная Маккарти для теоретических целей, оказалась зафиксирована в качестве синтаксиса реального языка программирования.

Причиной скепсиса Маккарти был как раз этот синтаксис — попросту говоря, Маккарти не верил, что кто-то станет пользоваться таким синтаксисом для написания настоящих программ. Синтаксис действительно выглядит несколько неожиданно. В Лиспе всё — и программы, и данные — состоит из *списков*, которые записываются в круглых скобках, а их элементы отделяются друг от друга пробелами. В виде списков записываются в том числе и обращения к функциям, при этом имя функции указывается первым элементом списка, а остальные его элементы задают значения фактических параметров (которые ещё, возможно, предстоит вычислить). Например, выражение `(+ (* 5 10) (- 55 5))` содержит три обращения к функциям — умножения, вычитания и сложения, а результатом его будет число 100, полученное как $(5 \times 10) + (55 - 5)$. Функции в Лиспе не ограничиваются арифметикой, с их помощью делается почти всё; например, чтобы напечатать сакраментальную фразу «Hello, world!» и перевести строку, нужно будет вычислить подряд два выражения: `(princ "Hello, world!")` и `(terpri)`.

Скепсис Маккарти можно понять, но практика показала, что к этой экзотике человек адаптируется очень быстро. За шестьдесят лет истории в Лиспе многое менялось, временами очень сильно, но базовый подход к синтаксису — списки в скобках, между элементами пробелы, имя функции указывается первым элементом списка — так и остался тот, что поддерживался в интерпретаторе Стивена Рассела.

Подробное изложение истории появления и раннего развития Лиспа приведено в статье Герберта Стояна [13], текст которой можно найти в Интернете через поисковые машины. Кроме того, можно посоветовать неформальную статью Пола Грэхема «Revenge of the Nerds» [14], в которой тоже обсуждаются вышеописанные события.

Научной общественности новый язык был представлен в 1960 году в статье Джона Маккарти «Символические выражения и их вычисление с помощью машины» [12]. За прошедшие с тех пор без малого шестьдесят лет в мире появились десятки, если не сотни разнообразных диалектов Лиспа, существовали даже специализированные компьютеры для выполнения программ на Лиспе. Как часто бывает в таких случаях, называть сейчас «Лиспом» какой-то конкретный язык было бы неправильно, в наше время это скорее семейство языков, причём довольно многочисленное.

Известна попытка создать «один Лисп для всех», который так и назвали — Common Lisp; работа по созданию этого диалекта началась в 1981 году, а в 1994 завершилась официальным принятием и публикацией стандарта ANSI. Сейчас, через четверть века, более-менее очевидно, что очередное комитетское изделие, как водится, пользы не принесло. С одной стороны, наличие официального стандарта, пусть даже он назывался не просто Лисп, а именно Common Lisp, создало у широкого круга программистов ощущение, что только это вот и есть язык Лисп, а всё остальное — это уже не Лисп. С другой стороны, спецификация, принятая в качестве стандарта, огромна по своему размеру и кошмарно сложна, но при этом, разумеется, не в состоянии удовлетво-

[1] По поводу названия языка Scheme см. сноску 4 на стр. 13.

[2] Англ. *Lisp* от слов *list processing*, т. е. *обработка списков*. Несколько забавно, что в английском языке есть слово *lisp*, оно означает «шепелявый».

рить потребности и «хотелки» поголовно всех пользователей, так что каждая реализация Common Lisp содержит ещё и свои собственные расширения. Поэтому переносимости исходных текстов между разными интерпретаторами — то есть именно того, чем хоть как-то можно оправдать «стандартизацию», — принятие стандарта толком не обеспечило. Зато поддерживаемых реализаций Лиспа стало гораздо меньше: соответствовать спецификации Common Lisp тяжело из-за её общей навороченности, так что много таких реализаций быть просто не может; ну а те, что спецификации соответствовать не могли и не хотели, и вовсе вымерли. Более того, принятие стандарта негативно повлияло также и на число сторонников Лиспа: распухшие спецификации нравятся далеко не всем, при этом «официальный» статус данной конкретной спецификации зачастую превращает утверждение «мне не нравится эта спецификация» в фатальное «мне не нравится этот ваш Лисп».

Сказанное, впрочем, почти не повлияло на ***диалекты*** Лиспа, изначально созданные для конкретных предметных областей — если угодно, выступающих в роли DSL[3]. Среди таких диалектов чаще всего упоминаются Emacs Lisp, лежащий в основе знаменитого текстового редактора Emacs, и AutoLISP, который используется в качестве встроенного языка в AutoCAD и других «продуктах» компании Autodesk.

Забегая вперёд, отметим, что оба диалекта используют так называемое ***динамическое связывание***, несмотря на то, что на момент их создания проблемы, вытекающие из такого подхода, были давно известны и хорошо изучены. Подробное обсуждение способов связывания символа с его значением и так называемой ***фунарг-проблемы*** будет приведено в §11.1.9.

Одним из самых известных и популярных диалектов Лиспа можно считать язык[4] Scheme. Первая его реализация появилась в 1975 году, причём, судя по воспоминаниям его создателей, получилось это непреднамеренно: изучая возможности вычислительной модели, основанной на новых концепциях «акторов» и «продолжений», исследователи в какой-то момент обнаружили, что у них получился компактный и эффективный диалект Лиспа, во многом непохожий на другие его варианты. В современных условиях язык Scheme популярнее Common Lisp, для него можно найти заметно больше активно поддерживаемых реализаций. «Раскрутке» этого диалекта весьма способствовало использование Scheme в качестве иллюстративного языка в знаменитом учебнике «Structure and Interpretation of Computer Programs» [15] (название часто сокращают аббревиатурой «SICP»); эту книгу написали Харольд Абельсон (Harold Abelson), Джеральд Сассман (Gerald Jay Sussman) и его жена Джулия, при этом Сассман также является одним из двух основных авторов Scheme. Вторым автором языка был ученик Сассмана Гай Льюис Стил мл. (Guy Lewis Steele Jr.), известный также как автор книги «Common Lisp the Language», описывающей стандарт Common Lisp, и заодно как руководитель стандартизационного комитета X3J13, создавшего этот пресловутый Common Lisp.

Гай Стил отметился также в процессе стандартизации других языков, в том числе ECMAScript (JavaScript), Фортрана и, что особенно печально, языка Си. Оставим всё это на его совести.

Языку Scheme будет посвящена отдельная глава нашей книги; современное положение вещей таково, что, выбирая версию Лиспа, стоит обязательно обратить внимание на Scheme — этот вариант Лиспа сейчас явно чувствует себя лучше, чем Common Lisp и основанные на нём диалекты. Несмотря на это, начнём мы именно с описания подмножества Common Lisp, поскольку во многом он ближе к «классическим» версиям Лиспа, а они заслуживают того, чтобы составить о них хотя бы общее представление.

К сожалению, выверты стандартизации не миновали и Scheme. Формальный стандарт, принятый IEEE, к счастью, никогда не использовался; зато в качестве стандарта (иногда уточняя, что он, видите ли, *de-facto*) многие воспринимают спецификации под названием «Revised^n Report on the Algorithmic Language Scheme», в названии которых индекс «^{n}» обозначает порядковый номер. Название этих спецификаций обычно сокращают до *R5RS*, *R6RS* и *R7RS*. Тексты спецификаций этой серии, в том числе ранних, можно найти на сайте `http://www.scheme-reports.org/`. R5RS вышла в 1997 году, и она оказалась последней, к которой не приложили руку никакие комитеты.

Обновлённая версия, получившая название R6RS, готовилась в соответствии с формальной процедурой, якобы учитывающей мнение участников сообщества, с проведением голосований и других типичных для комитетской работы мероприятий. Результат обескураживал одним только своим объёмом: он состоял из четырёх отдельных документов, представлявших описание самого языка на 90 страниц, описание стандартных библиотек на 70 страниц, «ненормативные приложения» на 6 страниц и пояснительную записку (*rationale*) на 20 страниц — всё это при том, что текст R5RS занимал всего 50 страниц.

В целом спецификация, вышедшая в своём окончательном варианте в 2007 году, неоправданно распухла, её сложность стала недоступна авторским коллективам многих существовавших реализаций и, что самое печальное, в таком виде это уже совершенно не было похоже на тот лаконичный язык Scheme, к которому мир успел привыкнуть. Совершенно не удивительно, что многие известные люди поспешили от этого «стандарта» дистанцироваться и заявить о том, что не собираются его признавать, а вместо этого намерены продолжать ориентироваться на R5RS.

В 2009 году так называемый Scheme Steering Committee, создавший R6RS, не нашёл ничего лучшего, как рекомендовать разделить Scheme на две разные спецификации — «большую» якобы для промышленного программирования и «маленькую» для использования в образовании и где-то там ещё. Изначальная бредовость такого деления, предполагающая, что безобразно распухший язык может быть чем-то лучше минималистичного в индустриальных условиях, осталась незамеченной. Так или иначе, документ R7RS, вышедший в 2013 году, описывает как раз эту «уменьшенную» версию Scheme: утверждается, что R7RS не отменяет R6RS, а «дополняет» её, если можно так выразиться в сложившейся ситуации.

Интересно, что к настоящему моменту самой поддерживаемой из спецификаций остаётся по-прежнему R5RS. Так, реализация Chicken Scheme, которую мы будем использовать в качестве наглядного пособия, содержит некоторые возможности из R7RS (причём возможности, надо отметить, полезные — настолько, что мы будем их использовать в примерах), но при этом создатели Chicken Scheme в документации настойчиво подчёркивают имеющуюся поддержку R5RS, а всё, что находится за её пределами, называют расширениями; ни R6RS, ни R7RS они даже не упоминают.

11.1.2. SBCL, GCL и ECL

Чтобы попробовать Лисп в деле, можно воспользоваться одним из существующих интерпретаторов Common Lisp; в среде Unix сейчас доступны три основных поддерживаемых интерпретатора: SBCL (Steel Bank Common Lisp), GCL (GNU Common Lisp) и ECL (Embeddable Common Lisp). Установочные пакеты всех трёх интерпретаторов имеются в большинстве основных дистрибутивов Linux; в Debian/Ubuntu соответствующие пакеты так и называются `sbcl`, `gcl` и `ecl`, и с хорошей степенью вероятности в вашем дистрибутиве они называются точно так же.

Наиболее простой и общепринятый способ создания на Лиспе (и других интерпретируемых языках) программы, которую можно запускать, состоит в том, чтобы оформить её в виде ***скрипта***, распознаваемого системой Unix; такие скрипты мы обсуждали ещё в первом томе (см. §1.2.15). Все три рассматриваемых нами интерпретатора Common Lisp предусматривают специальную поддержку для работы в режиме интерпретатора скрипта: для SBCL этот режим включается флагом командной строки `--script`, для GCL — флагом `-f`, для ECL — флагом `-shell`. В частности, программа «Hello, world» на SBCL будет выглядеть так:

```
#!/usr/bin/sbcl --script
(princ "Hello world!")
(terpri)
```

Для GCL имеем почти то же самое:

```
#!/usr/bin/gcl -f
(princ "Hello world!")
(terpri)
```

И для ECL:

```
#!/usr/bin/ecl -shell
(princ "Hello world!")
(terpri)
```

Скрипт можно будет запустить как обычную программу, если установить ему права на исполнение с помощью команды `chmod`:

```
chmod +x myfile.lsp
```

[3] *Domain Specific Language* (англ.); в этом контексте слово *domain* наиболее корректно переводится как «предметная область», а весь термин превращается в корявое «специфический язык для предметной области»; обычно используется не совсем точный, но зато менее корявый перевод «предметно-ориентированный язык».

[4] Вопрос, чем всё-таки считать Scheme — самостоятельным языком программирования или диалектом Лиспа — относится к сфере бесплодных и бессмысленных терминологических дискуссий. Считайте так, как вам удобнее.

После этого скрипт запускается привычным нам способом — командой вроде `./myfile.lsp`.

На всякий случай напомним, что для скриптов действует целый ряд ограничений: во-первых, интерпретатор сам по себе не может быть скриптом, то есть в роли интерпретатора может выступать только бинарный исполняемый файл[5]; во-вторых, в большинстве unix-систем на длину первой строки скрипта налагается определённое ограничение, причём в некоторых случаях достаточно жёсткое — всего 32 символа; ещё одно ограничение, пожалуй, самое неудобное: интерпретатору (самому интерпретатору как программе) можно указать параметр командной строки, но максимум один (вторым послужит имя файла скрипта, его система подставит сама). К счастью, разработчики интерпретаторов об этом знают и обычно предусматривают специальный флажок для этих целей — как мы только что видели, это относится и к нашим трём интерпретаторам Common Lisp.

Традиционно интерпретаторы Лиспа поддерживают интерактивный режим работы, известный под названием *read-eval-print loop* (*цикл чтения-вычисления-печати*), сокращённо ***REPL***. Именно в этот режим вы попадёте, дав в терминале команду `sbcl`, `gcl` или `ecl`.

Если вы уже написали какие-то функции на Лиспе и хотите их попробовать в интерактивном режиме, имеет смысл при запуске интерпретатора указать ему, что перед входом в REPL следует прочитать и обработать ваш файл. Для SBCL это делается флагом `--load`; добавив ещё флаг `--noinform`, вы сэкономите место на экране, запретив интерпретатору печатать информацию о своей версии и правовом статусе:

```
sbcl --noinform --load myfile.lsp
```

Для GCL и ECL файл, который нужно загрузить, обозначается почти так же, только знак минуса в начале флага используется один, а не два:

```
gcl -load myfile.lsp
```

К сожалению, флага для отмены печати информации о версии ни GCL, ни ECL не предусматривают, вся эта информация печатается всегда при входе в интерактивный режим.

При работе в интерактивном режиме обнаруживается ещё одно существенное различие между разными интерпретаторами. GCL читает вводимые пользователем строки с помощью библиотеки GNU Readline, что позволяет редактировать вводимую строку стрелочками, пользоваться историей ввода (клавиши стрелок вверх и вниз), поиском в истории (`Ctrl-R`), автодополнением слов по нажатию `Tab` — словом, всеми возможностями интерактивного ввода команд, которые привычны нам по интерпретатору командной строки. SBCL и ECL не включают в себя GNU Readline (возможно, из-за лицензионных ограничений), так что сами по себе они вводимую строку редактировать не позволяют, не предоставляют ни истории команд, ни автодополнения; всё это, впрочем, можно получить в своё распоряжение, если воспользоваться для их запуска программой `rlwrap`:

```
rlwrap sbcl --noinform --load myfile.lsp
rlwrap ecl -load myfile.lsp
```

Это может быть не столь удобно, поскольку `rlwrap` ничего не знает об именах символов Лиспа, так что автодополнением имён символов воспользоваться не удастся; но в остальном работать с интерпретатором станет намного приятнее.

Программа `rlwrap` специально предназначена для работы с интерактивными интерпретаторами, которые не поддерживают редактирование вводимой строки сами, и нам она ещё много раз пригодится; скорее всего, в вашей системе она уже есть, но если это не так — обязательно установите её. Кроме собственно редактирования вводимой строки, `rlwrap` также предоставит в ваше распоряжение возможности повтора последних введённых команд, хранения истории команд, текстового поиска по этой истории и т. п. Интерфейс ко всему этому великолепию — традиционный для GNU Readline: стрелки вверх и вниз позволяют просматривать историю, `Ctrl-R` (от слова *reverse*) включает поиск, клавиша `Tab` служит для автодополнения вводимых слов. Чтобы включить автодополнение имён файлов, добавьте в командной строке флажок `-c`. При большом желании от `rlwrap` можно добиться автодополнения имён функций и спецформ Лиспа (как и вообще любых слов), для чего требуется создать специальный текстовый файл и указать его в командной строке ключом `-f` или через переменные окружения. Подробности вы найдёте в man-странице на `rlwrap`.

Разница между GCL, SBCL и ECL проявляется не только в том, как запускать интерпретатор, но и в особенностях реализуемого языка. Например, GCL и SBCL предоставляют встроенную глобальную переменную для доступа к аргументам командной строки, но в SBCL эта переменная называется `*posix-argv*` (вот прямо так, со звёздочками), тогда как в GCL — несколько сложнее: `si::*command-args*`. Значение этой переменной — список строк, соответствующих привычным нам `argv[0]`, `argv[1]` и т. д., в этом аспекте интерпретаторы едины. Например, программа `echo`, печатающая свою командную строку, на SBCL будет выглядеть так:

```
#!/usr/bin/sbcl --script
(mapcar
    #'(lambda (arg) (princ arg) (princ " "))
    (cdr *posix-argv*)
)
(terpri)
```

а на GCL — почти так же, но не совсем:

```
#!/usr/bin/gcl -f
(mapcar
    #'(lambda (arg) (princ arg) (princ " "))
    (cdr si::*command-args*)
)
(terpri)
```

В ECL возможность доступа к аргументам командной строки тоже есть, но организована она несколько странно: нужно вызвать функцию с именем `ext:command-args` (без параметров), и она вернёт список строк, только при работе в режиме интерпретатора скрипта функция возвращает список параметров, полученных *интерпретатором*, а не скриптом; в этот список входит имя самого интерпретатора (обычно `/usr/bin/ecl`) и параметр `-shell`, и только за ним располагаются имя скриптового файла (то, что мы привыкли считать за `argv[0]`) и все остальные аргументы, если они есть. Поэтому `echo` на ECL выглядит несколько сложнее (обратите внимание на `cdddr`, которая отбрасывает от списка первые *три* элемента, а не один):

```
#!/usr/bin/ecl -shell
(mapcar
    #'(lambda (arg) (princ arg) (princ " "))
    (cdddr (ext:command-args))
)
(terpri)
```

Между реализованными в SBCL, GCL и ECL диалектами есть и другие, более существенные различия; некоторые из них мы упомянем далее, но с большинством так и не столкнёмся, поскольку изучать будем «классическое» подмножество Лиспа. Ничего из того, чем диалект SBCL отличается от диалекта GCL или ECL, в это подмножество не входит.

Оставшуюся часть этого параграфа мы посвятим попыткам (увы, не очень успешным) *откомпилировать* программу на Лиспе; её можно безболезненно пропустить, если вы не планируете в ближайшее время написать на Лиспе программу, которую придётся передавать сторонним пользователям. Информации, приведённой выше, вам хватит, чтобы попробовать в режиме интерпретации все возможности и примеры, которые мы будем обсуждать.

Создатели SBCL и GCL в документации упорно называют свои реализации компиляторами, что может создать у вас неверное представление об их возможностях. Начиная с первого тома нашей книги, мы постоянно пользовались компиляторами и могли привыкнуть к тому, что под ***компиляцией*** подразумевается действие, дающее в результате *исполняемый файл*, то есть такой файл, который можно *запустить*, и в результате произойдёт всё, что написано в нашей программе. Посмотрим, как с этим обстоят дела у SBCL, GCL и ECL.

Начнём с SBCL. Создать с его помощью исполняемый файл возможно, но для этого требуются весьма нетривиальные телодвижения; чтобы понять, почему они такие, стоит для начала разобраться, что́ *на самом деле* при этом происходит. **SBCL формирует исполняемый файл, сохраняя в него образ (дамп) всей памяти своего процесса** — и это, к сожалению, не шутка и не преувеличение. Чтобы получить из нашей программы исполняемый файл, нужно запустить интерпретатор SBCL (пусть другие, включая его создателей, сколько угодно кричат, что это не интерпретатор, реальности это не изменит), загрузить в него нашу программу на Лиспе и — да! — воспользовавшись специальной встроенной функцией, записать в файл сразу всё, что сейчас есть в

[5] Вообще-то для Linux это не так, он позволяет интерпретатору быть скриптом, но в большинстве других unix-систем интерпретатор обязан быть обычным исполняемым файлом.

памяти интерпретатора, включая, кроме функций нашей программы, весь код самого интерпретатора и все его служебные структуры данных.

Делается это с помощью функции `sb-ext:save-lisp-and-die`, которая принимает один обязательный параметр — имя исполняемого файла, и плюс к тому несколько *ключевых параметров*, влияющих на то, что́ это в итоге будет за файл. Нам нужны по меньшей мере два таких ключевых параметра, один из которых заставит функцию оформить дамп памяти как исполняемый файл (иначе за что мы вообще боремся), а второй укажет, с какой функции начать исполнение, когда кто-то этот наш файл запустит. Если этот второй параметр не указать, после запуска файла пользователь попадёт в REPL и увидит всё то же самое, что и при обычном запуске SBCL: информацию о версии и приглашение к вводу.

В принципе можно вызвать функцию `sb-ext:save-lisp-and-die` прямо из файла нашей программы, и это, пожалуй, проще, чем остальные способы (задать её вызов в командной строке или в запущенном REPL). Например, чтобы получить «откомпилированную» с помощью SBCL программу «Hello world», можно написать следующий текст:

```
; sbcl_hello_comp.lsp
(defun main ()
    (princ "Hello, world!")
    (terpri)
)
(sb-ext:save-lisp-and-die "sbcl_hello"
    :executable t
    :toplevel 'main
)
```

и заставить SBCL обработать этот файл, т. е. вычислить все его формы:

```
sbcl --load sbcl_hello_comp.lsp
```

В результате в текущей директории появится исполняемый файл `sbcl_hello`, который можно запустить привычным нам способом, и он даже напечатает «`Hello, world!`». Вот только есть тут, как говорят, один нюанс: его *размер*. В системе на компьютере автора этих строк получившийся бинарник весил *больше 60 мегабайт*.

Слегка помочь делу может добавление в вызов `save-lisp-and-die` ещё одного ключевого параметра — `:compression t`. После этого получающийся исполняемый файл будет размером «всего лишь» 12 MB, но радоваться, пожалуй, преждевременно: во-первых, это всё равно безумно много, а во-вторых, при этом программа будет тратить ощутимое время на распаковку собственного кода перед началом работы: после запуска она, прежде чем напечатать «`Hello, world!`», примерно полсекунды потормозит. В общем и целом это не компиляция, а издевательство какое-то.

GCL в этом плане несколько привлекательнее, но не сильно. У этого интерпретатора даже предусмотрен специальный флаг командной строки `-compile`, но пусть он вас не обнадёживает: результатом запуска с этим флагом становится *объектный* файл, который, как мы знаем, для запуска надо ещё слинковать. Пытаясь узнать, с чем же и как его линковать, мы словно натыкаемся на глухую стену.

«Секрет» здесь прост: создатели GCL вовсе не это имели в виду. Объектный файл, получаемый в результате «компиляции», предназначен не для того, чтобы с ним линковаться и получать исполняемый файл, а для того, чтобы *загружать его из интерпретатора* — и да, он, конечно, грузится быстрее, чем исходник на Лиспе. Есть здесь, впрочем, и функция для записи дампа в исполняемый файл (почему-то этот способ очень нравится создателям реализаций Common Lisp). Однако если не принять хитрых мер, при запуске полученного исполняемого файла мы попадаем в REPL, и избежать этого можно только указанием параметров командной строки.

«Хитрые меры» состоят в том, что в нашей программе нужно обязательно предусмотреть функцию с именем `si:top-level` (да, вот прямо так, с двоеточием) и именно из неё всё сделать, воспринимая её как главную функцию нашей программы, вроде хорошо знакомого нам `main`. Эта функция заменит собой REPL, то есть при запуске полученного исполняемого файла вместо REPL будет запущена наша программа, что нам и требуется. Текст программы «Hello, world» в этот раз получится таким:

```
(defun si:top-level ()
    (princ "Hello, World!")
    (terpri)
)
```

Превратить его в исполняемый бинарник можно следующим образом. Запускаем интерпретатор `gcl` и для начала «компилируем» наш текст (это можно сделать и из командной строки с помощью параметра `-compile`, но интерпретатор потом всё равно придётся запустить), затем загружаем полученный объектный файл в интерпретатор и после этого сохраняем состояние интерпретатора в исполняемом файле:

```
>(compile-file "hello.lsp")

Compiling hello.lsp.
End of Pass 1.
End of Pass 2.
OPTIMIZE levels: Safety=0 (No runtime error checking), Space=0, Speed=3
Finished compiling hello.lsp.
#p"hello.o"

>(load "hello.o")

Loading hello.o
start address -T 0xd96610 Finished loading hello.o
160

>(si:save-system "hello")
```

После последней команды интерпретатор почему-то завершается, но в этом ничего плохого нет — мы бы всё равно из него вышли, чтобы посмотреть, что́ там у нас получилось. А получился у нас исполняемый файл размера более скромного, чем от SBCL, но всё равно весьма и весьма впечатляющего: на машине автора получилось 12 MB, причём это, видимо, без компрессии, поскольку при его запуске никаких задержек не заметно.

На первый взгляд с ролью компилятора неплохо справляется ECL, он даже позволяет перевести вашу программу на Лиспе в текст на Си, который потом можно связать с программой, написанной на Си или Си++. Попробуем сделать исполняемый файл программы «Hello, world». Берём файл `hello.lsp`:

```
(princ "Hello world!")
(terpri)
```

Теперь запускаем интерпретатор ECL (да, придётся это сделать, хоть это и странно) и в нём даём две хитрые команды:

```
> (compile-file "hello.lsp" :system-p t)

;;;
;;; Compiling hello.lsp.
;;; OPTIMIZE levels: Safety=2, Space=0, Speed=3, Debug=0
;;;
;;; End of Pass 1.
;;; Finished compiling hello.lsp.
;;;
#P"/home/avst/hello.o"
NIL
NIL
> (c:build-program "hello" :lisp-files '("hello.o") :epilogue-code ())

#P"hello_ecl"
>
```

Всё, после этого в текущей директории появится исполняемый файл `hello`, причём даже не очень большого размера — на машине автора этих строк получилось чуть больше 38 KB. Пусть вас, впрочем, это не обманывает — полученный файл зависит от разделяемых библиотек, включающих всю реализацию ECL.

Да и вообще радоваться явно рано. Если попробовать проделать то же самое с приведённой выше программой `echo` и запустить её с параметрами командной строки, мы можем с удивлением убедиться, что первые два параметра «кто-то съел»:

```
avst@host:~$ ./echo_ecl abra schwabra kadabra foo bar
kadabra foo bar
```

Впрочем, нетрудно догадаться, кто именно их съел — это была функция `cdddr`, которая в программе-скрипте делала всё правильно, выкидывая два «лишних» параметра, а вот в откомпилированной программе этих двух параметров просто нет, так что выкинутыми оказываются параметры обычные, т. е. нужные. Всё заработает корректно, если заменить `cdddr` обратно на `cdr` — но, естественно, перестанет работать скриптовая версия. Что мешало автору ECL выкинуть два параметра командной строки из выдачи функции `ext:command-args` при наличии флага `-shell` — вопрос, наверное, риторический.

Откровенно говоря, кажется, будто авторы реализаций Common Lisp вообще не предполагают использования их творений для создания настоящих программ, передаваемых конечным пользователям. К сожалению, это впечатление только усилится, когда мы в §11.1.14 доберёмся до организации ввода-вывода.

11.1.3. S-выражения: гетерогенная модель данных

В основе вычислительной модели языка Лисп лежат так называемые ***S-выражения***, нацеленные прежде всего на построение *гетерогенных списков* — в один такой список могут входить элементы разных типов, в том числе и другие списки. S-выражениями в Лиспе представляются как данные, так и функции, составляющие программу.

Формально говоря, S-выражение есть либо ***атом***, либо ***точечная пара***. Примерами атомов служат прежде всего ***константы*** (литералы), такие как числа (`12`, `165.37`, `-7`), символьные литералы (англ. *character constants*: `#\a`, `#\5`, `#\&` для представления соответственно буквы `a`, цифры `5` и символа `&`, `#\Space`, `#\Newline`, `#\Tab` и т. п. для обозначения пробельных и неотображаемых символов) и строковые литералы (`"This is a string"`).

Многие современные диалекты Лиспа умеют работать с рациональными и комплексными числами. Так, токен `2/3`, введённый, что очень важно, без пробелов (поскольку это именно один токен, а не арифметическое выражение), в большинстве реализаций обозначает рациональное число $\frac{2}{3}$, причём физически представленное в виде числителя и знаменателя, а значит — без потерь точности, в отличие от чисел с плавающей точкой. Для комплексных чисел используются конструкции вроде `#C(1 -2)`, при этом действительная и мнимая части могут быть представлены целыми, рациональными или «плавучими» числами. Для работы с целыми числами большинство реализаций Лиспа использует так называемую *длинную арифметику*, представляя целые в виде битовых массивов произвольной длины, что позволяет не заботиться о нехватке разрядности. В нашем разговоре о парадигмах все эти материи не представляют серьёзного интереса, но иметь их в виду полезно.

Другой пример атомов — так называемые ***символы*** (англ. *symbols*)[6], роль которых в Лиспе приблизительно соответствует роли *идентификаторов* в других языках программирования. Большинство диалектов Лиспа позволяет использовать в именах символов не только буквы и цифры, но и многие знаки препинания, а также знаки арифметических действий; так, имена `X`, `AS21`, `ABRAKADABRA`, `GO-JOHNNY-GO`, `==^==`, `*STDIN*` в Common Lisp и большинстве других диалектов считаются корректными именами символов. Если не предпринимать специальных мер, латинские буквы в именах символов автоматически приводятся к верхнему регистру, так что `AS21`, `as21`, `As21` и `aS21` — это разные способы записи одного и того же имени символа.

Символы используются в качестве имён функций и переменных, в том числе формальных параметров тех же функций, но этим их роль не исчерпывается. Символ *сам по себе является значением*, т. е., например, один символ может быть значением переменной, обозначенной другим символом (или даже тем же самым). Это позволяет использовать символы в роли своеобразных *констант*, равных только самим себе и более ничему; приблизительным аналогом этого явления в знакомых нам языках будут идентификаторы, входящие в множество значений перечислимого типа в Паскале или типа `enum` в Си и Си++, но в отсутствие статической типизации в Лиспе возможности символов оказываются намного шире. Например, мы можем написать функцию, которая по своему смыслу должна возвращать целое число, но вынуждена в некоторых случаях фиксировать наступление особой ситуации — ошибки или каких-то ещё хитрых обстоятельств, делающих возврат числа бессмысленным; в Лиспе мы можем для каждой такой ситуации предусмотреть свой символ. Так, функция, подсчитывающая сумму целых чисел, записанных в текстовый файл с заданным именем, может возвращать эту сумму, если всё в порядке, возвращать символ `open_error` в случае, если файл не удалось открыть на чтение, символ `wrong_data` — если файл открылся, но внутри обнаружилось нечто, не являющееся представлением целых чисел, и символ `empty`, если файл оказался пуст.

Особую роль играет в Лиспе символ, имеющий имя `nil`, который используется для обозначения логической лжи, а также ***пустого списка***; конструкция `()` считается другим способом записи этого атома. *Значением* символа `nil` всегда является он сам.

Существует много других видов атомов, с которыми мы постепенно познакомимся; напомним, что атомом, строго говоря, считается вообще всё, что не является загадочной ***точечной парой*** — и, по-видимому, нам пора рассказать, что́ же это за зверь.

Синтаксически точечная пара представляет собой два произвольных S-выражения, разделённых точкой, а вся конструкция при этом берётся в круглые скобки. Сразу же отметим, что **точка не считается в Лиспе разделителем**, поскольку может входить в состав имён символов, в запись дробных чисел и некоторых других атомов. Разделителями в классическом смысле в Лиспе выступают только круглые скобки, двойные кавычки, используемые для записи строковых констант, и ещё одиночный апостроф, обратный апостроф и запятая, играющие в Лиспе особые роли; от всего остального точку в записи точечной пары необходимо отделять пробелами. В левой или правой частях точечной пары могут стоять совершенно произвольные S-выражения, как атомы, так и другие точечные пары. Так, корректной записью точечных пар будут следующие выражения:

```
(1 . 2)    (abra . kadabra)    (12 . "String")    (75.7 . 100.0)
((1 . 2) . (3 . 4))    (1 . (2 . (3 . ())))    ((10 . 20) . 30)
```

Семантически точечную пару удобно представлять как область памяти из двух указателей — благо у читателя, можно надеяться, с понятием указателя никаких проблем уже нет. Думая о левой и правой частях точечной пары как об указателях (а не как о значениях), будет проще осознать, что **одно и то же выражение** — то есть *физически* один и тот же объект в памяти — **может входить более чем в одну точечную пару.** Пока в качестве таких выражений выступают неизменяемые константы, это не так уж важно, но некоторые объекты данных в Лиспе могут в процессе выполнения программы *изменяться*, и прежде всего это касается самих точечных пар; и вот тут уже свойство разделяемости объектов становится весьма заметно. Пусть, например, одна и та же точечная пара `(1 . 2)` входит в состав выражений `("Foo" . (1 . 2))` и `("Bar" . (1 . 2))`; если теперь, воспользовавшись одним из этих выражений, заменить в левой части пары `1` на `100`, это одинаково отразится на обоих выражениях.

Последовательность из нескольких точечных пар, в которой каждая следующая пара является значением правой части предыдущей, а правая часть последней пары в последовательности имеет значением пустой список (символ `nil`, он же `()`), называется ***списком***; элементами такого списка считаются значения, находящиеся в *левых* частях его точечных пар. Для списка предусмотрен сокращённый способ записи: в двойных скобках перечисляют все его элементы, разделяя их пробелами. Например, конструкция

```
(1 . (2 . (3 . (4 . ()))))
```

может быть записана намного короче: `(1 2 3 4)`. Естественно, список может содержать элементы разных типов, в том числе элементом списка вполне может быть как простая точечная пара, так и список, и они, в свою очередь, могут содержать элементы произвольных типов. Например, все следующие списки корректны:

```
(1 foo "Bar")    ((1 2) (3 4 5) (6 . 7))    (((((((())))))))
```

Правая часть последней точечной пары списка может содержать значение, отличное от пустого списка; в этом случае говорят о ***неправильном*** или ***точечном списке***. Для такого списка тоже есть специальное обозначение; например, вместо `(1 . (2 . (3 . 4)))` можно написать `(1 2 3 . 4)`. Точечные списки в Лиспе применяются сравнительно редко, но полностью игнорировать их существование было бы ошибкой.

Изначально точечные пары и состоящие из них списки были в Лиспе единственными «сложными» структурами данных, но для практического программирования это слишком неудобное ограничение. Большинство существующих ныне реализаций Лиспа наряду со списками поддерживает также хорошо знакомые нам по Паскалю и Си массивы, к элементам которых можно обращаться по номеру (индексу); структуры с именованными полями; спецификация Common Lisp включает даже средства работы с упоминавшимися в первом томе ***хеш-таблицами*** (см. т. 1, §2.10.8), причём благодаря гетерогенной модели данных как ключом, так и значением в каждом звене такой хеш-таблицы могут служить S-выражения произвольного типа. Спецификация Common Lisp включает в себя так называемую CLOS — Common Lisp Object System, которая представляет собой поддержку объектно-ориентированного программирования в виде объектов, содержащих как данные, так и функции-методы.

Все эти сущности представляют собой S-выражения, которые могут входить в другие S-выражения на произвольную глубину вложенности. Любопытно отметить, что поскольку ни векторы, ни структуры, ни хеш-таблицы, ни объекты не являются са-

[6] Здесь мы вынуждены в очередной раз обратить внимание читателя на сложности перевода с английского. Английские термины *character* и *symbol* оба традиционно переводятся на русский язык словом «символ», но под словом *character* подразумевается символ, отображаемый на экране, вводимый с клавиатуры и т. п. — такой символ, из каких складываются слова, строки и тексты; английское слово *symbol* имеет совершенно иной смысл — это «символ» в смысле «знак», «обозначение» (например, «условные обозначения на картах»), «эмблема», либо «символ» во всяческих философских и прочих гуманитарных контекстах вроде фразы «голубь — символ мира» или «изображение черепа символизирует смерть».

ми по себе точечными парами (хотя и могут содержать в себе прямо или косвенно неограниченное их количество), все они считаются атомами.

Рассматривать работу с несписочными структурами данных в Лиспе мы не будем, поскольку они не привносят никаких принципиально новых парадигм, но иметь в виду их существование может быть полезно.

11.1.4. Вычисление S-выражений

S-выражения как структуры данных с их гетерогенностью, бесспорно, следует считать основой Лиспа, но данные сами по себе ещё не составляют вычислительной модели. Чтобы с помощью S-выражений можно было записать *программу*, в Лиспе вводится операция ***вычисления S-выражений*** — довольно обширный свод правил, по которым большинству S-выражений ставится в соответствие *результат их вычисления*; именно эти вычисления и составляют выполнение программы, написанной на Лиспе. Правила вычислений часто зависят от контекста: одно и то же S-выражение в различных случаях может при вычислении давать разные значения.

Проще всего обстоит дело с простейшими атомарными константами — числами, алфавитными символами (*characters*; не путайте их с символами-идентификаторами), строками и некоторыми другими атомами: они вычисляются *сами в себя*. Например, число `25` является, как мы знаем, S-выражением; результатом его вычисления будет оно само. Точно так же обстоит дело, например, с заглавной латинской буквой `A`, которая в Лиспе обозначается как `#\A`: при вычислении она даёт сама себя. Строка `"Simple string"` тоже при вычислении превратится сама в себя.

Несколько сложнее обстоят дела с символами (идентификаторами). Среди них есть минимум два, которые всегда вычисляются сами в себя: это уже знакомый нам `nil` (он же пустой список) и ещё символ `t`, обозначающий логическую истину[7]. Изменить значения этих символов невозможно. Скорее всего, в любой реализации Лиспа найдутся и другие символы, значение которых неизменно — например, значением символа `pi` всегда будет наилучшее на данной архитектуре (или в данной реализации Лиспа, что не всегда то же самое) приближение к числу π.

Остальные символы в зависимости от контекста могут *иметь или не иметь значение*; если значение у символа есть, то вычисление символа как S-выражения даст это значение, в противном случае в большинстве реализаций попытка вычисления приведёт к ошибке; существуют, впрочем, диалекты Лиспа, в которых любой символ всегда имеет то или иное значение (в таких реализациях, как правило, символ, значение которого не было задано явно, имеет в качестве значения `nil` или сам себя), но такое встречается редко. Следует отметить, что вычисление символов зависит от контекста чаще, чем вычисление других выражений; контекстам мы позже посвятим отдельный параграф.

Сложнее всего выглядят правила для вычисления списков — или ***вычислимых форм***, как обычно называют списки, предназначенные к вычислению. Отметим прежде всего, что в роли вычислимых форм обычно выступают «правильные» списки, то есть списки, заканчивающиеся атомом `nil`; попытка вычислить в качестве формы точечный список или простую точечную пару, правая часть которой не содержит списка (ни обычного, ни пустого), как правило, приводит к ошибке.

Дальнейший ход вычисления формы зависит от того, что́ этот список содержит в своём первом элементе. В большинстве случаев первым элементом формы должен быть символ; крайне редко используется имеющаяся во многих диалектах, включая Common Lisp, возможность, при которой первым элементом формы выступает так называемый ***лямбда-список***; эту сущность мы рассмотрим позже, а пока можно не обращать внимания на этот экзотический случай.

Символ, стоящий в форме первым элементом, определяет дальнейший выбор стратегии вычисления этой формы. Возможностей тут три. Прежде всего, с символом может быть связана ***функция*** — библиотечная (встроенная в реализацию Лиспа) или описанная в программе. Этот случай можно считать основным; программа на Лиспе состоит из функций, а применение функций к аргументам — своего рода фундамент, на котором построено её исполнение. Стратегия вычисления здесь самая простая и понятная: все элементы формы, кроме первого, *вычисляются*, после чего функция, идентифицированная первым элементом формы, *применяется* к результатам этих вычислений как к параметрам. Рассмотрим для примера вычисление S-выражения

```
(* (+ 3 7) 5 (- 6 4))
```

Первый элемент этой формы — символ с именем «`*`» (напомним, что в большинстве диалектов Лиспа это допустимый идентификатор), с которым связана встроенная в Лисп функция умножения: она принимает произвольное число числовых аргументов и возвращает их произведение. Поскольку это простая функция, сначала лисп-вычислитель должен вычислить остальные элементы формы: список `(+ 3 7)`, константу `5` и список `(- 6 4)`. Как можно догадаться, символы с именами «`+`» и «`-`» связаны со встроенными функциями сложения и вычитания; вспомнив, что константы, в том числе числовые, вычисляются сами в себя, мы увидим, что результатом вычисления элементов исходной формы будут числа `10`, `5` и `2`; именно к этим аргументам будет применена функция умножения, а результатом станет, очевидно, число `100`.

Случай, когда первый элемент формы идентифицирует функцию, одновременно самый простой и самый важный, но при этом никоим образом не единственный. С символом может быть связан также ***макрос***; кроме того, символ может задавать ***специальную форму***.

Система макросов в Лиспе существенно отличается от тех макропроцессоров, которые мы видели в языке ассемблера, Си и Си++: по сути лисповский макрос — это правило, написанное на Лиспе (и могущее, в частности, использовать функции из нашей программы), согласно которому должна быть построена *форма*, которая уже будет вычислена для получения результата. Макрос свои параметры получает в том виде, в котором они входят в макровызов, то есть макропараметры не вычисляются; макрорасширение представляет собой вычисление тела макроса подобно тому, как вычисляется тело обычной функции, но результатом вычисления тела макроса становится не конечный результат, а *форма*, которая и будет в итоге вычислена, чтобы дать результат. По возможности интерпретатор Лиспа вычисляет макросы (производит макрорасширение) один раз во время первичного анализа программы, но такая возможность у него есть не всегда: например, невозможно произвести макрорасширение, когда обработаны ещё не все функции, вызываемые в теле макроса. В таких случаях макрорасширение откладывается до первого вычисления формы, содержащей макровызов. Поскольку параметры макроса вычислению не подлежат, макрорасширение можно произвести один раз, после чего заменить в теле программы макровызов на результат макрорасширения.

Подробное рассмотрение макросов Лиспа мы оставим за рамками нашей книги; отметим только, что эта система может служить весьма наглядной иллюстрацией одного простого факта: макросы совершенно не обязаны быть такими опасными, кривыми и противными, как в языке Си. Тут вопросы не к макропроцессированию как явлению, а к конкретному языку и конкретному воплощению этого явления.

Если символ в первом элементе формы указывает на то, что эта форма — специальная, мы имеем дело с самым заковыристым и некрасивым случаем. Никаких единых правил для этого случая не существует. **Правила вычисления у каждой спецформы свои**: некоторые элементы формы могут вычисляться, другие при этом вычисляться не будут, спецформа может подразумевать некие вложенные списки, элементы которых используются по неким хитрым правилам, один и тот же элемент одной и той же формы может вычисляться или не вычисляться в зависимости от результатов других вычислений, спецформы могут манипулировать контекстами, так что часть вычислений пройдёт в одном контексте, часть — в другом. Описание каждой поддерживаемой спецформы содержит подробные указания на то, как будет происходить её вычисление. Отметим, что **набор спецформ для каждого диалекта Лиспа закрытый**, то есть написать на Лиспе новую спецформу невозможно. С помощью спецформ создатели Лиспа и его диалектов решают проблемы, которые не получается решить ни функциями, ни макросами. Можно встретить утверждение, что спецформам в Лиспе отведена роль, которую в других языках играет синтаксис; при этом обычно замечают, что синтаксиса в Лиспе вообще-то нет — не называть же синтаксисом правила записи списков.

В дальнейшем изложении нам потребуются некоторые из спецформ и встроенных функций Лиспа; попытаемся о них рассказать. Как обычно в таких случаях, не будем пытаться объять необъятное и перечислить, например, несколько сотен (!) функций, встроенных в тот же Common Lisp — ограничимся лишь теми, которые нам понадобятся в примерах, и добавим к ним некоторые из тех, что могут улучшить наше понимание происходящего.

Начнём со спецформы, которая применяется столь часто, что для неё введено специальное краткое обозначение. Эта спецформа называется `quote`, предполагает один аргумент, а служит она для *блокировки вычисления*; попросту говоря, свой единственный аргумент она возвращает в качестве зна-

[7] Вообще-то логической истиной в Лиспе считается любое S-выражение, отличное от `nil`, но функции, по своему смыслу возвращающие логическое значение, в качестве представления «истины» используют всегда именно символ `t`.

чения, не вычислив его. Например, значением выражения `(quote (1 2 3))` будет список `(1 2 3)`; если, скажем, нам нужно передать такой список в функцию (пусть она называется `f`), то без спецформы `quote` нам придётся туго: вычисление выражения `(f (1 2 3))` (если `f` — простая функция) вызовет ошибку, поскольку интерпретатор для начала попытается вычислить выражение `(1 2 3)` как форму, а первый элемент списка — константа `1` — совсем не похожа на символ, именующий функцию или спецформу. Поэтому, чтобы применить функцию `f` к списку `(1 2 3)`, конструкция потребуется более сложная: `(f (quote (1 2 3)))` или, если использовать краткую запись, `(f '(1 2 3))`. Конструкция вида `'x` как раз и есть то краткое обозначение для `(quote x)`. Кроме списков, защиту от вычисления часто применяют к символам, если хотят использовать их не в роли переменных, а в роли значений.

Арифметические функции сложения, вычитания и умножения мы уже видели, отметим только, что все эти функции принимают произвольное количество аргументов, причём функция вычитания, если ей дать один аргумент, меняет его знак на противоположный, т. е. работает как унарный минус. Функция деления обозначается, как мы и могли ожидать, символом «`/`» (здесь самое время вспомнить о рациональных числах, чтобы результат вычисления, скажем, формы `(/ 2 3)` не стал для нас сюрпризом). Функции арифметического сравнения чисел обозначаются символами «`<`», «`>`», «`=`», «`<=`», «`>=`»; для отношения «не равно» обычно используется символ «`/=`», хотя здесь возможны варианты. Эти функции могут вернуть либо `nil`, либо `t`.

Коль скоро у нас появились логические значения, стоит упомянуть функцию логического отрицания `not`. Здесь всё просто: у неё один аргумент, и если этот аргумент `nil`, она возвращает `t`, а для любого другого значения аргумента возвращает `nil`. Со связками для конъюнкции и дизъюнкции дела обстоят сложнее. Обозначаются они, как можно догадаться, символами `and` и `or`, но вычисляются при этом по особым правилам. Форма `and` вычисляет свои аргументы по одному слева направо, и если значением очередного аргумента оказалась ложь (т. е. `nil`), вычисления немедленно прекращаются, возвращается ложь; если все аргументы вычислились в истину, возвращается *значение последнего аргумента* (это не обязательно `t`, ведь логической истиной считается что угодно, кроме `nil`). Аналогичным образом работает и `or`: она вычисляет свои аргументы по одному слева направо, и если очередное значение — истина, то вычисления прекращаются, форма возвращает как раз это последнее значение; если же все вычисленные значения были ложны, возвращается ложь.

Внимательный читатель уже мог заметить подвох: мы ни разу не назвали `and` и `or` функциями. В самом деле, *функциями* они быть не могут, ведь при вычислении формы с вызовом обычной функции лисп-вычислитель сначала вычисляет значения всех аргументов, и лишь после этого получает управление собственно функция, тогда как для `and` и `or` все аргументы, кроме первого, могут так и остаться невычисленными (это важно, например, если у них есть ***побочные эффекты***).

Можно рассматривать `and` и `or` как спецформы, хотя, например, в спецификации Common Lisp они обозначены как *встроенные макросы*; их действительно можно реализовать как макросы, используя базовую спецформу `if`, о которой речь пойдёт ниже.

Для анализа и синтеза структур данных, составленных из точечных пар (в том числе списков), Лисп тоже содержит ряд встроенных функций. Для создания новой точечной пары используется функция `cons`, она принимает два аргумента и возвращает точечную пару, составленную из их значений. Так, список `(1 2 3)` можно составить следующим выражением:

```
(cons 1 (cons 2 (cons 3 nil)))
```

Название `cons` происходит от слова *construct*. Интересно, что в английских текстах точечные пары часто называются просто *conses*; можно встретить обороты вроде «*the last cons of the list*».

С именами двух основных функций для *анализа* списков всё несколько сложнее. Функция `car` извлекает из точечной пары значение левой части, функция `cdr` — значение правой части. Например, выражение `(car '(1 2 3))` вернёт значение `1`, выражение `(cdr '(1 2 3))` — значение `(2 3)`[8].

Чтобы понять, почему для этих функций используются столь странные имена, нужно обратиться к истории (см. §11.1.1). Машинное слово IBM-704 составляло 36 бит, в роли оперативной памяти использовались хранилища на магнитных сердечниках на 4096 ячеек по 36 бит каждая. Таких устройств к одной машине можно было подключить несколько (например, конкретный экземпляр IBM-704, на котором работал Джон Маккарти, имел два таких банка памяти); адресная шина была 15-разрядной, так что теоретически машина могла бы работать с восемью банками памяти, но на практике к одной машине столько памяти никогда не подключали ввиду её дороговизны, да и потребности в оперативной памяти в те времена были весьма скромными.

Машинные коды, всегда занимавшие ровно одну 36-битную ячейку памяти, имели два основных формата — с 3-битными и 12-битными кодами операций. Оба формата команд предусматривали место для 15-битного *адреса* и 3-битного поля *tag* для выборки индексных регистров (самих этих регистров было всего три, так что каждый из битов поля *tag* подключал один из регистров; если их подключалось больше одного, их содержимое зачем-то подвергалось побитовой дизъюнкции). В командах с 12-битным кодом операции после этого оставалось всего шесть свободных бит, два из них использовались для неких флагов, четыре оставались без применения. Для нас сейчас намного интереснее другой формат команды, в котором на код операции отводилось всего три бита. Оставшиеся 15 бит использовались, чтобы задать некое значение (как мы сказали бы сейчас, «непосредственный операнд») прямо в коде команды. В командах, работавших с адресами, это поле использовалось, чтобы вычесть его из значения адреса, и называлось *decrement*.

Косвенной адресации в современном её понимании в те времена ещё не было, поэтому в программах широко использовался приём, ныне невозможный в принципе — прямая модификация машинных инструкций во время исполнения. К ассемблеру прилагалась библиотека макросов, включавшая в том числе средства для анализа и модификации инструкций в обоих форматах; в частности, макрос для извлечения адресной части инструкции (для обоих форматов инструкции, поскольку адресная часть находилась в одних и тех же битах) назывался `CAR` (*Contents of the Address part of the Register*), а макрос для извлечения декрементной части из инструкции с трёхбитным кодом назывался, как можно догадаться, `CDR` (*Contents of the Decrement part of the Register*). Словом «*Register*» в этом контексте обозначалась, как ни странно, ячейка памяти. Регистров в современном понимании на IBM-704 было шесть — аккумулятор (`AC`), множитель-делитель (`MQ`), три индексных регистра и регистр `ILC` (*Instruction Location Counter*), хранивший адрес текущей (точнее, *следующей*) машинной команды; но само слово «регистр» в то время ещё не приобрело строгого узкого значения, используемого в наши дни.

Поскольку точечная пара представляет собой фактически запись из двух указателей, создателям Лиспа во главе с Маккарти показалось вполне естественным располагать два адреса в одной ячейке памяти точно так же, как располагались адресная и декрементная части в машинных инструкциях с трёхбитным опкодом: как минимум это позволяло применять уже имеющиеся макросы для манипуляций с частями 36-битного слова. В честь этих макросов и назвали лисповские функции `car` и `cdr`; впрочем, если уж говорить совсем точно, изначально эти две функции попросту *представляли собой* ровно то же, что и соответствующие макросы, потому за ними и закрепились названия макросов. В литературе часто можно встретить утверждение, что `CAR` и `CDR` — это машинные команды из системы команд IBM-704; как видим, это не соответствует действительности, команд с такими именами на IBM-704 не было.

В различных диалектах Лиспа неоднократно предпринимались попытки избавиться от столь «странных» имён функций. Так, в Common Lisp есть функции `first` и `rest`, делающие ровно то же самое, а в других диалектах встречаются функции с именами `head` и `tail`; похоже, однако, что программисты, работающие на Лиспе, попросту *не хотят* ни на что заменять `car` и `cdr`, рассматривая их (вместе со всей историей про IBM-704) как некую часть «культуры Лиспа» и своего рода знак собственной посвящённости.

Если для построения списка из нескольких элементов выписывать явно все вызовы `cons`, код получается довольно громоздкий и неудобочитаемый. Функция `list` позволяет сделать то же самое нагляднее: она строит список из своих аргументов. Например, вычисление формы `(list 'a 'b 'c)` даст в качестве результата список `(A B C)`.

При работе со списками могут быть полезны несколько ***предикатов***, т. е. функций, проверяющих некоторое условие и возвращающих логическое (истинностное) значение. Функция `listp` проверяет, является ли её единственный аргумент списком (возможно, пустым): возвращает `t` для произвольных точечных пар и атома `nil`, в противном случае возвращает `nil`. Функция `atom` возвращает истину, когда её аргумент — атом, то есть *не* точечная пара. Функция `null` проверяет, является ли её аргумент пустым списком; на самом деле она работает точно так же, как и `not`, но для наглядности в случаях, когда по смыслу имеется в виду именно проверка списка на пустоту, лучше использовать имя `null`.

Программировать на Лиспе совершенно невозможно без ***условных операций***, причём, похоже, Маккарти придумал условную операцию первым. Позже эта операция вошла во многие языки программирования, в том числе в Си и Си++ — это та самая тернарная операция, записываемая через

[8] Если тут что-то оказалось непонятно, самое время вернуться чуть назад и вспомнить, что такое список и как он строится из отдельных точечных пар. Пока не будет достигнуто понимание происходящего, двигаться дальше смысла нет.

знак вопроса и двоеточие. В Лиспе та же операция задаётся специальной формой `if`. Эта спецформа принимает три аргумента (последний можно опустить), вычисляет первый из них, затем, если получилось что-то, отличное от `nil`, то есть истина, вычисляет второй аргумент, если же результатом вычисления первого стала ложь (`nil`), то вычисляет третий аргумент. Результатом вычисления всей формы становится результат вычисления второго или третьего аргумента. Если третьего аргумента нет, а при вычислении первого получился `nil`, результатом всей формы становится `nil`. Например, форма

```
(if (> x 0) (f x) (g (- x)))
```

для строго положительных значений x вычислит значение $f(x)$, а для неположительных — значение $g(-x)$.

Как ни странно, программисты, пишущие на Лиспе, используют форму `if` очень редко, практически всегда предпочитая ей более сложный, но и более универсальный вариант ветвления, представленный спецформой `cond`[9]. Эта форма позволяет организовать ветвление на произвольное количество случаев, подобно операторам выбора — паскалевскому `case` и `switch` из Си; кроме того, для каждого варианта выбора можно указать последовательность из произвольного количества форм, подлежащих выполнению.

Форма `cond` обладает довольно хитрой структурой. Первым элементом формы, естественно, служит символ `cond`; второй и последующие элементы должны представлять собой списки, называемые `cond`-предложениями (*cond clauses*), которые будут рассматриваться один за другим, начиная с первого. Каждое предложение в свою очередь состоит из вычислимых выражений; первый элемент предложения вычисляется в качестве условия: если результатом стала истина (что угодно, кроме `nil`), то `cond` *выбирает* это предложение, в противном случае он переходит к рассмотрению следующего предложения. Если ни одно из предложений не было выбрано, результатом всей формы `cond` становится `nil`. Если же одно из предложений подошло, то есть его заглавный элемент вычислился в истину, то остальные формы из этого предложения, если таковые есть, будут вычисляться одна за другой, а результатом всего `cond` будет объявлен результат вычисления последней из них. Если предложение состояло только из заглавного элемента, результатом `cond` будет результат вычисления этого элемента.

Часто в последнем предложении формы `cond` заглавным элементом ставят атом `t`; такое предложение срабатывает, если ни одно из предыдущих не выбрано — аналогично `else` из паскалевского `case` или метке `default` из оператора `switch` языка Си.

Так, вместо приведённой выше формы `if` можно написать:

```
(cond
    ((> x 0) (f x))
    (t (g (- x)))
)
```

Почему программисты чаще всего предпочитают именно так и писать, даже если ветвей всего две и в каждой ровно одна форма — вопрос, на который автор этих строк ответа не нашёл.

В завершение разговора о встроенных возможностях Лиспа опишем предикаты, предназначенные для *сравнения* произвольных выражений. Самый простой из них — `eq`: он возвращает истину тогда и только тогда, когда оба его аргумента при вычислении дают физически один и тот же объект — не два одинаковых, а один и тот же. Гарантируется это обычно только для символов: двух символов с одинаковыми именами в среде выполнения быть не может.

Предикат `eql` чуть сложнее: он выдаёт истину в тех же случаях, что и `eq`, но в дополнение к этому — ещё и если оба его аргумента представляют собой одно и то же число (одного типа) или один и тот же алфавитный символ. Как ни странно, реально существующие реализации Лиспа часто делают так, что во всех случаях, когда два объекта равны в смысле `eql` (например, результаты вычисления `(+ 5 7)` и `(- 14 2)`), они оказываются равны также и в смысле `eq`. В некоторых диалектах одинаковые *строки* равны в смысле `eql`, даже если это физически разные объекты (две копии одной строки), но в Common Lisp это не так.

Следует обратить внимание, что `eql` — это не то же самое, что арифметическое сравнение (символ `=`). Так, числа `2` и `2.0` равны в арифметическом смысле — выражение `(= 2 2.0)` даст истину, но эти два числа не будут равны ни в смысле `eql`, ни тем более в смысле `eq`, поскольку это совершенно разные объекты, имеющие разные типы.

Наиболее сложным можно считать предикат `equal`: кроме сравнения объектов в смысле `eql`, он также сравнивает (рекурсивно) их подобъекты: левую и правую части точечной пары, соответствующие элементы массивов одинаковой длины и т. п.

Из дальнейшего текста мы узнаем ещё некоторое количество встроенных в Лисп спецформ и функций, но пока разговор о них заканчиваем — для примеров нам хватит того, что уже введено.

11.1.5. Пользовательские функции

Программа на Лиспе представляет собой последовательность ***форм верхнего уровня*** (англ. *top-level forms*), которые в ходе выполнения последовательно вычисляются одна за другой. Обычно большая часть этих форм представляет собой описания функций.

Для описания функции, имеющей имя (в виде символа), применяется спецформа `defun`[10]. Вторым элементом этой формы ставят символ, который будет служить именем для новой функции; третий элемент — список формальных параметров, то есть, опять же, символов, которые послужат в теле функции именами, связанными со значениями фактических параметров, указанных при вызове; далее один или несколько элементов представляют собой тело функции — произвольные формы, которые следует вычислить; значением функции автоматически становится значение, возвращённое последней из этих форм. Например, следующая функция вычисляет дискриминант квадратного уравнения:

```
(defun sqe-discrim (a b c)
    (- (* b b) (* 4 a c))
)
```

Здесь `sqe-discrim` — имя функции, символы `a`, `b` и `c` — формальные параметры, выражение `(- (* b b) (* 4 a c))` — тело, состоящее в данном случае из одной формы.

Сделаем одно важное замечание. В отличие от функций и форм, рассмотренных нами ранее, **форма `defun` имеет побочный эффект**: она связывает символ — имя функции — с объектом функции, и эта связь, разумеется, продолжает существовать после того, как вычисление формы `defun` уже завершено.

Рассмотрим более сложный пример. Следующая функция вычисляет длину списка (мы назовём её `length1`, поскольку в большинстве реализаций Лиспа функция `length` встроена):

```
(defun length1 (lst)
    (cond
        ((null lst) 0)
        (t (+ 1 (length1 (cdr lst))))
    )
)
```

Базисом рекурсии мы здесь выбрали случай пустого списка, при этом функция возвращает ноль; во всех остальных случаях мы вычисляем длину хвоста нашего списка и прибавляем единицу. У этой реализации есть определённый недостаток: если дать ей параметром точечный список (напомним, это такой список, который заканчивается чем-то отличным от `nil`), произойдёт ошибка. Этот недостаток легко исправить, заменив вызов предиката `null` на `atom`:

```
(defun length1 (lst)
    (cond
        ((atom lst) 0)
        (t (+ 1 (length1 (cdr lst))))
    )
)
```

Обе версии функции можно переписать вообще без `cond`, используя более простую форму `if`:

[9] Строго говоря, при наличии макросов форма `cond` может быть реализована через `if` в виде макроса; как следствие, в большинстве реализаций `cond` — это не спецформа, а встроенный макрос.

[10] Согласно спецификации Common Lisp, `defun` — это встроенный макрос; перечислять примитивы, через которые он реализуется, мы не будем.

```
(defun length1 (lst)
    (if (atom lst) 0 (+ 1 (length1 (cdr lst))))
)
```

Применять такой вариант или всегда использовать cond — решайте сами.

В таком виде функция будет считать любой атом списком нулевой длины. Попробуем написать такой вариант, который будет для списков (в том числе точечных) возвращать их длину, то есть количество образующих список точечных пар, для пустых списков — 0, а для произвольных атомов — специальное значение, в роли которого у нас выступит nil. Такая реализация окажется несколько сложнее, зато мы сможем продемонстрировать, как выглядит cond более чем на два предложения:

```
(defun length2 (lst)
    (cond
        ((null lst) 0)
        ((atom lst) nil)
        ((atom (cdr lst)) 1)
        (t (+ 1 (length2 (cdr lst))))
    )
)
```

Такое построение функции, когда телом выступает форма cond, для Лиспа типично. Рекурсия в этом языке выступает в роли основного инструмента, а при рекурсии, как мы знаем, требуется выделение её базиса; кроме того, часто требуется обработать особые случаи, как это получилось в нашем примере.

В §9.2.4 мы подробно обсудили оптимизацию остаточной рекурсии, сделав замечание, что для языка Си такая оптимизация технически возможна, но почти никогда не применяется, поскольку может привести к проигрышу в быстродействии, при этом рекурсия в Си используется не столь активно. Для Лиспа ситуация прямо противоположна: рекурсия здесь используется повсеместно, при этом вызовы функций и их исполнение реализованы так, что оптимизация остаточной рекурсии не только не снизит общего быстродействия, но может его даже повысить.

Все наши реализации подсчёта длины списка написаны так, что рекурсия в них не является остаточной — после рекурсивного вызова приходится ещё прибавлять единицу к полученному результату. Способ борьбы с этой проблемой нам уже известен: задействовать ***накапливающий параметр*** (см. §9.2.4). Перепишем функцию length2 так, чтобы она допускала оптимизацию остаточной рекурсии:

```
(defun length2_do (lst len)
    (cond
        ((atom (cdr lst)) len)
        (t (length2_do (cdr lst) (+ 1 len)))
    )
)
(defun length2 (lst)
    (cond
        ((null lst) 0)
        ((atom lst) nil)
        (t (length2_do lst 1))
    )
)
```

Техника с накапливающим параметром, как мы знаем, нужна не только для превращения рекурсии в остаточную. Например, функцию, «переворачивающую» список, без накопителя в принципе достаточно трудно реализовать, тогда как с его использованием реализация оказывается тривиальной:

```
(defun reverse1_do (lst res)
    (cond
        ((null lst) res)
        (t (reverse1_do (cdr lst) (cons (car lst) res)))
    )
)
(defun reverse1 (lst) (reverse1_do lst ()))
```

Например, для списка (1 2 3) функция reverse1_do будет вызвана последовательно со следующими парами аргументов:

```
(1 2 3) ()
(2 3)   (1)
(3)     (2 1)
()      (3 2 1)
```

— и последний вызов вернёт свой второй параметр в качестве результата.

11.1.6. Разрушающие функции

Как мы обсуждали в §9.1.3, при описании функциональных языков программирования вместо термина «модифицирующее действие» обычно используется термин «***разрушающее действие***» (или ***разрушающая операция*** и т. п.). Напомним, что примерами таких действий служат ввод-вывод, а также хорошо знакомое нам по Паскалю и Си *присваивание*.

Разумеется, любая реализация Лиспа предусматривает средства ввода-вывода, но этим ассортимент разрушающих действий не исчерпывается. Говоря о символах Лиспа, мы неоднократно упоминали, что они могут использоваться в роли переменных; больше того, мы даже видели их в этой роли, но пока только в специфическом случае *формальных параметров пользовательских функций*. При таком использовании переменная связывается со своим значением — ***фактическим параметром функции*** — непосредственно перед выполнением тела функции, а по завершении выполнения тела связь формального параметра с фактическим значением исчезает. В языках вроде Паскаля и Си формальные параметры представляют собой локальные переменные, так что получение ими значений при старте подпрограммы (процедуры или функции) не относится к разрушающим действиям: в самом деле, речь здесь идёт не об изменении (разрушении) значения существующей переменной, а о формировании начального значения для переменной, которая вот только сейчас начинает существовать. Как мы увидим позже, аналогичная ситуация имеет место и в большинстве современных реализаций Лиспа: связывание формального параметра с его фактическим значением ничего не разрушает и не изменяет.

Формально говоря, пока мы видели в Лиспе лишь одно разрушающее действие: вычисление формы defun (опять-таки, с сугубо формальной точки зрения) *изменяет* связанную с символом функцию, и то, что в большинстве случаев до вызова defun с символом никакой функции связано не было, формально ситуацию не меняет. На такой вариант разрушающей операции можно не обращать внимания, ведь обнаружить, что defun реально что-то модифицирует, можно лишь в случае, если мы начнём вычислять какие-то посторонние формы раньше, чем будут вычислены все defun'ы; в подавляющем большинстве программ ничего подобного не происходит. Но то, что мы не видели других разрушающих действий, отнюдь не означает, что их здесь нет: в Лиспе присутствует не только присваивание (включая присваивание значений глобальным переменным), но и средства, позволяющие модифицировать существующие S-выражения, в том числе изменять списки.

Начнём с присваивания. В Лиспе оно делается с помощью спецформы setq; буква q в названии происходит от слова *quote* и указывает на то, что первый аргумент формы setq как бы неявно «заквотирован» (чем-то вроде формы quote), то есть не вычисляется. Изначально форма setq предполагала два аргумента: символ, значение которого изменяется, и собственно значение; первый аргумент не вычисляется, второй, естественно, вычисляется, иначе было бы не интересно. Например, (setq x 25) присваивает переменной x значение 25; если после этого сделать (setq x (+ x 1)), с той же переменной окажется связано значение 26.

Как мы увидим позже, на самом деле применение формы quote нас бы, как говорится, не спасло; но чтобы понять причину этого, нужно представлять, как работают лексические контексты, разговор о которых у нас впереди.

Для полноты картины сделаем два замечания. Во-первых, в спецификации Common Lisp, а вслед за ней — и в большинстве реализаций форма setq принимает произвольное *чётное* количество параметров; первым, третьим, пятым и т. д. параметрами указываются имена переменных, а вторым, четвёртым, шестым и т. д. — присваиваемые значения. Больше того, Common Lisp наряду с формой setq включает ещё и форму psetq, которая делает всё то же самое, но «как будто параллельно» — то есть она сначала вычисляет все присваиваемые значения (т. е. если в них используются значения переменных из числа присваиваемых, то используются их старые значения), и лишь затем присваивает вычисленные значения переменным. Например, (psetq x y y x) поменяет местами значения переменных x и y.

Второе замечание не столь очевидно. В некоторых (отнюдь не во всех) существующих реализациях присваивать значения разрешается только локальным переменным, а также тем глобальным, которые заранее объявлены. К реализациям, которые этого требуют, относится, например, SBCL, но не относятся GCL, ECL и CLISP; по-

следние молча позволяют присваивать значения любым символам, тогда как SBCL выдаёт предупреждение, если попытаться присвоить значение необъявленной глобальной переменной. Объявление делается с помощью формы `defvar` — например, `(defvar *myglobvar* 100)` объявляет глобальную переменную с именем `*myglobvar*` и начальным значением 100. Звёздочки, обрамляющие имя — это дань традиции; хотя Лисп этого не требует, считается, что глобальные переменные нужно называть именно так, чтобы они выделялись из общего текста.

Отметим, забегая вперёд, что `defvar` не только «объявляет» глобальную переменную, но и устанавливает для символа, служащего её именем, ***динамический режим связывания***; подробное обсуждение этого у нас впереди.

Значение символа — далеко не единственное, что может быть изменено в Лиспе. Ещё в ранних диалектах появились функции, позволяющие изменить левую и правую части существующей точечной пары; эти функции, существующие до сих пор, традиционно называются `rplaca` и `rplacd` (*replace car*, *replace cdr*). Например, если мы последовательно вычислим формы

```
(setq x '(1 2 3))
(rplaca x 10)
(rplacd (cdr x) 30)
```

— то список, выступающий значением `x`, станет таким: `(10 2 . 30)`. Интересно, что само по себе значение переменной `x`, то есть то, что физически хранится в объекте `x`, мы при этом не изменили: переменная `x` в качестве своего значения по-прежнему ссылается на ту же самую (то есть *физически* ту же самую) точечную пару, просто левая часть этой точечной пары заменена значением `10`, правая осталась прежней, но это тоже точечная пара, и вот её правая часть заменена значением `30`.

Ещё более странный эффект получится, если сделать так:

```
(setq c '(1 2 3))
(rplacd (cdr (cdr c)) c)
```

Выражение `(cdr (cdr c))` возвращает *последнюю* точечную пару списка `(1 2 3)` — ту, где слева стоит `3`, а справа — пустой список. Вот этот самый пустой список функция `rplacd` в нашем примере заменит на значение переменной `c`, т. е. на сам исходный список — точнее, на ссылку на первую его точечную пару. Результатом станет «закольцованный» (или ***циклический***) список; он по-прежнему состоит из трёх точечных пар, но последняя из них своей правой частью ссылается на первую. При попытке напечатать такой список некоторые интерпретаторы, реализованные достаточно хитро, выдают что-то вроде

```
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 ...
```

и останавливаются, но большинство, увы, не столь «сообразительны» и продолжают печатать повторяющееся «`1 2 3`», пока их не прервут.

Возможность изменить содержимое существующего списка требует учитывать в работе знание о том, как в действительности реализованы списки Лиспа — в частности, тот факт, что списки никогда не копируются, если программист этого не потребует в явном виде. Например, если сделать так:

```
(setq a '(1 2 3 4))
(setq b (cons 10 (cdr a)))
```

— то значением переменной `a` будет список `(1 2 3 4)`, значением переменной `b` — список `(10 2 3 4)`, и если бы не `rplaca` и `rplacd`, мы могли бы совершенно не задумываться о том, что «хвост» `(2 3 4)` у этих двух списков *общий* — это не копии, это именно физически одна структура данных. Возможность модификации списков позволяет этот факт объективно обнаружить: если, например, мы сделаем `(rplacd (cdr a) nil)`, тем самым мы обрежем список `a`: теперь значением `a` будет список `(1 2)`. Но ведь список `b` состоит (или скорее состоял) из тех же точечных пар, за исключением самой первой, а изменили мы значение (правое) во второй паре; так что, если теперь посмотреть на значение переменной `b`, мы — возможно, с некоторым удивлением — обнаружим там список `(10 2)`.

Отметим, что создать копию списка можно, например, так:

```
(defun copylist (lst)
    (cond
        ((atom lst) lst)
        (t (cons (car lst) (copylist (cdr lst))))
    )
)
```

В Common Lisp для этого предусмотрена встроенная функция `copy-list`.

Современные реализации Лиспа, как мы уже говорили, не ограничиваются в плане сложных структур данных одними только списками — здесь можно обнаружить и массивы, и записи с полями, и хеш-таблицы, и даже объекты. По мере роста количества *изменяемых* сущностей в Лиспе росло и число функций, осуществляющих изменения, пока в какой-то момент не появилось понятие «обобщённой переменной» — произвольной области памяти, содержащей ссылку, которую можно изменить; для присваивания таким «обобщённым переменным» новых значений ввели единую спецформу `setf`. Как и `setq`, `setf` принимает произвольное чётное число аргументов, причём на втором, четвёртом и остальных чётных местах стоят выражения, которые следует присвоить. На первом, третьем и прочих нечётных местах располагаются выражения, которые, если бы их вычислить, дали бы *значение* той позиции в памяти, которую следует изменить. Например, `(setf (car x) v)` эквивалентно `(rplaca x v)`, `(setf (cdr (cdr (cdr x))) v)` делает то же, что и `(rplacd (cdr (cdr x)) v)`, а `(setf x y)` заменяет `(setq x y)`. Но для большинства возможных применений `setf` никаких аналогов не предусмотрено. Так, в Common Lisp мы можем создать массив из 10 элементов:

```
(setq arr (make-array 10))
```

Обращение к элементам такого массива происходит с помощью функции `aref` (*array reference*) — например, элемент с индексом 7 (индексирование идёт с нуля) можно получить с помощью `(aref arr 7)`; для *присваивания* элементам массива используется `setf`, например:

```
(setf (aref arr 3) '(abra kadabra))
```

Аналогичным образом функция `char` позволяет выделить отдельный символ из строки, а с помощью `setf` этот символ можно изменить; например, после вычисления форм

```
(setq str "foobar")
(setf (char str 4) #\u)
```

значением `str` будет строка `"foobur"` — физически это тот же объект в памяти, просто в нём изменился один элемент. Именно с помощью `setf` можно поместить значение в хеш-таблицу или изменить в ней уже имеющееся значение, изменить поле записи или объекта и многое другое. Некоторое представление о масштабе катастрофы можно составить на примере функции `symbol-function`, дающей доступ к той области памяти *внутри символа*, которая отвечает за ассоциированную с символом функцию. Так, выражение `(symbol-function 'car)` вернёт *функциональный объект*, соответствующий хорошо знакомой нам встроенной функции `car`; а теперь сделаем так:

```
(setf (symbol-function 'car1) (symbol-function 'car))
```

После этого символ `car1` будет ассоциирован с той же самой функцией, то есть станет её вторым именем. Как мы увидим позже, в Лиспе можно создавать безымянные функции; в сочетании с `setf` и `symbol-function` это позволяет при желании обойтись без формы `defun`.

Идея формы `setf` сама по себе интересна: например, возможна реализация лисп-вычислителя, которая, пока это возможно, хранит результаты вычислений в виде указателей на существующие в памяти ссылки на объекты, и переходит к созданию новых ссылок или копированию ссылок лишь в случае, когда результат только что создан (не является частью объемлющей структуры данных) или если менять содержимое ссылки никто не собирается. Именно так была реализована форма `setf` в библиотеке InteLib, когда-то давно написанной автором этих строк; подробности читатель найдёт на сайте `http://www.intelib.org`. Проблема в том, что в Common Lisp на уровне спецификации закреплён совершенно другой подход к реализации `setf`, на удивление тупой по своей сути: *для каждой функции, которая может выступать «селектором» в первом аргументе `setf`, попросту пишется соответствующая «изменяющая» функция*, а роль самого `setf` сводится к тому, чтобы нужную функцию найти и вызвать. Эти «модицифирующие функции» в Common Lisp имеют странные *имена* в виде списка (!) из двух элементов, первым из которых выступает символ `setf`: например, для функции `func` соответствующая изменяющая функция называлась бы `(setf func)`. Таким образом, `setf` в том виде, в котором он введён в Common Lisp, вовсе не снимает потребности в отдельной функции для каждого случая модифицируемой ссылки внутри существующей структуры данных; просто все они снабжаются неочевидными именами, а для их вызова используется единый «интерфейс» (собственно форма `setf`). Интересно, что GCL не поддерживает «модифицирующие функции», реализация `setf` там какая-то своя; в SBCL всё работает в соответствии со спецификацией.

Согласно спецификации Common Lisp форма `setf` имеет целый ряд «встроенных» случаев, которые она якобы обрабатывает сама без обращения к дополнительным функциям, и это касается всех наиболее очевидных применений `setf`, включая случаи `car`, `cdr`, `aref` и многие другие. Откровенно говоря, здесь даже не вполне понятно, становится ли от этого лучше или хуже, ведь это означает, что реализация `setf` должна обо всех

этих случаях знать — и иметь, как следствие, соответствующий объём реализации; если представить себе компилируемый вариант Common Lisp, реализованный так, чтобы в исполняемый файл вставлять только реально используемые в данной программе части библиотеки, то код `setf` придётся вставить едва ли не в любой исполняемый файл — и, разумеется, целиком, ведь это одна функция. Впрочем, в другом месте спецификации — в описании спецформы `function` — явным образом упоминается функция с именем `(setf cadr)`, и в SBCL реально присутствуют функции с подобными именами (в том числе и `(setf car)`, и `(setf cdr)`, и `(setf char)` и т. п.)

Основной вывод, который можно сделать после ознакомления с широким ассортиментом разрушающих операций Лиспа, очевиден: Лисп, несомненно, допускает и даже поощряет функциональное программирование, но назвать его *языком функционального программирования* было бы ошибкой. Основная парадигма Лиспа, как следует из его названия (напомним, оно происходит от *list processing*) — это гетерогенные S-выражения, если угодно, слабоструктурированные данные и средства для их обработки; возможность писать программы в функциональном стиле оказывается лишь дополнением, хотя и приятным.

11.1.7. Функция `eval`

S-выражения в роли единого представления для программы и данных открывают довольно интересную возможность: в Лиспе есть встроенная функция, которая *вычисляет произвольное S-выражение*. Эта функция называется `eval`; она принимает один параметр — собственно выражение, которое должно быть вычислено. Фактически параметр функции `eval` вычисляется дважды: сначала, как и для любой функции, интерпретатор вычисляет выражение, указанное фактическим параметром при вызове `eval`, а затем к полученному значению применяется сама функция `eval`, так что это значение *снова* вычисляется и уже результат этого вычисления становится результатом `eval`. Например, при вычислении формы

```
(eval (list 'cdr '(list 1 2 3)))
```

сначала выражение `(list 'cdr '(list 1 2 3))` будет вычислено в качестве аргумента формы вызова функции. Результатом вычисления станет список `(cdr (list 1 2 3))`; именно его функция `eval` получит фактическим параметром и именно его вычислит в соответствии со своим назначением. При этом перед обращением к функции `cdr`, поскольку это обычная функция, подвергнется вычислению её фактический параметр (в данном случае `(list 1 2 3)`), результатом вычисления будет список `(1 2 3)`, к нему будет применена функция `cdr`, так что конечным результатом станет список `(2 3)`.

Функция `eval` может не произвести должного впечатления на читателя, если не обратить внимание на то, что **S-выражение, служащее аргументом для `eval`, вычисляется во время работы программы**. С другой стороны, ничто не мешает выражению, переданному аргументом в функцию `eval`, содержать ту же форму `defun` или обращения к `setf` с `symbol-function` в роли селектора (см. предыдущий параграф); всё это означает, что **программа на Лиспе может во время исполнения менять сама себя**.

Эта возможность была бы до определённой степени ограниченной, если бы в S-выражениях, конструируемых во время выполнения программы, могли использоваться только символы, упоминаемые в самой программе. Однако в Лиспе можно найти произвольный символ по его текстовому имени — это делается встроенной функцией `intern`, аргументом которой служит строка; если символа с заданным именем нет, он создаётся, и в этом заключается исходный смысл `intern`, но если символ с таким именем уже есть, в качестве значения возвращается именно он. Функции `eval` и `intern`, если рассмотреть их совместно, предоставляют работающей программе доступ ко всем возможностям Лиспа, предусмотренным в данной реализации, и снимают практически любые ограничения по применению ***метапрограммирования*** (см. §9.4.3). В самом деле, с использованием `eval` и `intern` программа на Лиспе может во время исполнения породить совершенно произвольный код на том же Лиспе — выражение, функцию, сколь угодно большой набор функций — и немедленно их исполнить.

Ясно, что метапрограммирование, не связанное никакими рамками — инструмент очень мощный, но такая мощь имеет свою цену, тоже, увы, немалую. Глядя на программу на Лиспе, в которой используется `intern`, невозможно предсказать, какие строки будут поданы ей на вход и, следовательно, какими символами программа в итоге воспользуется; в общем случае это *алгоритмически неразрешимая проблема*, даже если не принимать строки из внешних источников, а только синтезировать их внутри программы. Конечно, во многих (простых) частных случаях такой анализ возможен, но если задаться соответствующей целью, можно будет без особых проблем сбить с толку любой анализатор.

Довершает дело наличие встроенной функции `read`, позволяющей прочитать из стандартного потока ввода (или из любого другого текстового потока) *произвольное S-выражение*, за исключением небольшого числа «нечитаемых» S-выражений, таких как уже знакомые нам *объекты функций*. При чтении `read` автоматически преобразует прочитанные имена символов в объекты символов (так же, как это делает `intern`). Если программа, которую мы рассматриваем, получает имена символов извне, в том числе с помощью `read`, о каких-либо прогнозах границ её поведения можно забыть. Таким образом, наличие `eval` и `intern` (и тем более `read`) означает, что **во время исполнения программы на Лиспе в памяти должен находиться целиком весь интерпретатор Лиспа**.

Как мы убедились в §11.1.2, многие реализации Лиспа, включая SBCL и GCL, позволяют программу «откомпилировать» в том смысле, что результатом компиляции становится исполняемый файл, не требующий присутствия породившего его интерпретатора, но на самом деле весь интерпретатор при этом оказывается внутри полученного исполняемого файла, размеры которого составляют десятки мегабайт, даже если ваша программа — простейший «Hello, world»; исполняемые файлы, порождаемые с помощью ECL, не столь велики, но зависят от динамически загружаемых библиотек, которые как раз и содержат весь интерпретатор.

Несмотря на это, например, в man-странице к интерпретатору SBCL можно найти следующую фразу:

> Even today, some 30 years after the MacLisp compiler, people will tell you that Lisp is an interpreted language. Ignore them[11].

Там же можно найти и объяснение столь безапелляционного заявления:

> SBCL compiles by default: even functions entered in the read-eval-print loop are compiled to native code...

Итак, прежде чем выполнять программу и её фрагменты, SBCL переводит их в другое представление, причём не просто в другое, а в *машинный код*. На этом основании авторы SBCL заявляют, что их реализация компилирующая; при этом их нисколько не смущает необходимость наличия в памяти во время исполнения программы на Лиспе не только интерпретатора Лиспа, но, как мы видим, ещё и компилятора (а также, добавим, и отладчика, и дизассемблера, и всех остальных подсистем, включённых в SBCL).

Здесь внимательный читатель может усмотреть некое противоречие с тем, что мы утверждали раньше — что работающая программа в нынешних условиях не может изменять собственный машинный код. Это действительно так, если говорить о коде, который операционная система прочитала из исполняемого файла и разместила в памяти: секция кода, как мы знаем, недоступна для записи. Размещать свежесгенерированный машинный код в секциях данных тоже бесполезно, поскольку эти области памяти недоступны для *передачи в них управления*, ну а что касается стека, то даже если он для выполнения доступен (что далеко не всегда так), то размещать там откомпилированные функции *неудобно* — стек имеет свойство увеличиваться и уменьшаться по мере вызовов подпрограмм и возвратов из них, так что размещённый в стеке машинный код сможет продолжать существовать лишь до тех пор, пока подпрограмма, сгенерировавшая его, не завершится. Иной вопрос, что в большинстве операционных систем процесс может запросить у ядра *новую* область памяти и указать для неё режим доступа — в том числе и доступность одновременно для чтения, записи и выполнения кода. В системах семейства Unix для этого используется системный вызов `mmap` (см. т. 2, §5.2.7).

По-видимому, причина столь вольного обращения с терминологией кроется в убеждении, что единственный недостаток интерпретируемого исполнения состоит в его низкой эффективности — попросту говоря, «честные» интерпретаторы работают медленно, а если их заставить работать быстро — например, встроив в них «компилятор», преобразующий фрагменты программы в машинный код, и исполнять эти фрагменты уже в преобразованном виде — то единственный недостаток интерпретации будет тем самым преодолён, и вообще это более не будет интерпретацией. Неоднократно отмеченную нами необходимость наличия интерпретатора в памяти во время исполнения программы предлагается игнорировать как несущественный фактор, и в

[11] В переводе на русский: «Даже сегодня, спустя около 30 лет после компилятора MacLisp, [некоторые] люди скажут вам, что Лисп — язык интерпретируемый. Не обращайте на них внимания».

пользу этого предложения, как правило, высказывается всё тот же тезис об эффективности, даже если про эффективность никто не спрашивал.

Возмём на себя смелость не согласиться со столь однобоким трактованием понятия «интерпретируемого» или «компилируемого» исполнения; к этому вопросу мы вернёмся в следующей части нашей книги.

11.1.8. Функции как объекты обработки

Как уже отмечалось (см. §9.1.3), одна из основополагающих особенностей функционального программирования — работа с функциями как объектами первого класса; это означает, что функция наравне с объектами данных может быть передана как параметр в другую функцию, может быть возвращена из функции в качестве возвращаемого значения и, в довершение картины, может быть создана во время выполнения программы. Функции, одним или несколькими параметрами которых служат другие функции, часто называют ***функционалами*** или ***функциями высшего порядка***[12].

Для начала, чтобы иметь материал для примеров, разберёмся, как добраться до объектов, соответствующих встроенным в Лисп функциям. Один способ для этого мы уже знаем: применить функцию `symbol-function` к символу, служащему именем встроенной функции. Например, вычислив выражение `(symbol-function 'cons)`, мы получим *объект функции* `cons`. Текстовое представление этого объекта зависит от интерпретатора — так, SBCL выдаст строку `#<function cons>`, а GCL — строку `#<compiled-function cons>`.

Через некоторое время нас ждёт подробный разговор о вводе-выводе, в том числе и о том, что обычно S-выражения могут быть прочитаны из текстового потока ввода штатными средствами Лиспа. Это позволяет выдать (например, в файл или на консоль) некую сложную структуру данных, и она будет представлена в виде текста в соответствии с синтаксическими правилами Лиспа, так что позднее можно будет применить к ней встроенные возможности для чтения и получить в памяти точную копию исходной структуры, причём это можно сделать где угодно и когда угодно — на следующий день или через год, в другом городе или на другом континенте — если это представление, например, сохранить в файле или передать по сети.

Объект функции в этом плане необычен: его текстовое представление, выдаваемое интерпретатором, не позволяет этот объект в будущем реконструировать, так что «прочитать его обратно» не получится. Для всех таких объектов, называемых «нечитаемыми» (*unreadable*), в Common Lisp зарезервировано представление вида `#<`*комментарий*`>`.

В дальнейших примерах мы предпочтём другой способ доступа к объектам функций, связанных с символами — спецформу `function`. Эта спецформа, не вычисляя свой единственный аргумент, выполняет над ним такое действие, какое выполнил бы интерпретатор Лиспа, если бы встретил этот аргумент в роли первого элемента формы вызова функции; для символов это действие — извлечь соответствующий функциональный объект, чтобы можно было применить к нему аргументы. Как мы уже упоминали при обсуждении операции вычисления S-выражения, первым элементом функциональной формы может быть не только символ — и в этом случае действие спецформы `function` может оказаться совсем не похоже на то, что делает функция `symbol-function`. В некоторых экзотических ситуациях результаты `symbol-function` и `function` не совпадают даже для символа (так происходит при использовании спецформы `flet`, вводящей локальные функции; мы эту возможность обсуждать не будем). Так или иначе, например, добраться до объекта функции `cons` можно, вычислив форму `(function cons)`.

Спецформа `function` применяется настолько часто, что для неё есть сокращённое обозначение — `#'` (по-английски это обозначение называется *funquote*). Именно его мы и будем использовать в дальнейшем тексте; например, чтобы добыть объект встроенной функции сложения, мы напишем `#'+`, а чтобы достать функцию `cons`, воспользуемся конструкцией `#'cons`.

Фундаментальная возможность объекта-функции, ради которой он, собственно говоря, и существует — это его применимость к аргументам. Основных способов *применить* функциональный объект в Лиспе два — это встроенные функции `apply` и `funcall`. Различие между ними на первый взгляд кажется косметическим: обе функции первым аргументом принимают объект функции, которую нужно применить к аргументам, но при этом `apply` предусматривает второй аргумент (один), который должен представлять собой список аргументов для применяемой функции, тогда как `funcall` принимает произвольное количество аргументов, и все, кроме первого, использует в качестве аргументов для применяемой функции. Например, если мы хотим сложить пять чисел, мы можем с одинаковым успехом написать:

```
(apply #'+ '(1 20 300 4000 50000))
(funcall #'+ 1 20 300 4000 50000)
```

Результатом в обоих случаях будет число `54321`. При более внимательном рассмотрении этих двух функций различие между ними перестаёт выглядеть «косметическим»: список аргументов, передаваемый в `apply`, мы можем сформировать во время работы программы, и при этом в зависимости от обстоятельств поменять его длину. Это позволяет применять вариадические[13] функции к списку аргументов, длина которого не была известна на момент написания программы. Если бы для этого пришлось применять `funcall`, достичь нужного эффекта можно было бы, разве что сформировав во время выполнения саму форму вызова `funcall` и вычислив её с помощью `eval`.

Спецификация Common Lisp допускает функциональным аргументом передавать в `apply`, `funcall` и другие встроенные функционалы не только собственно объект функции, но и имя функции — символ. Функционалы сами разбираются, что́ им передали, и если это символ, то извлекают из него функцию самостоятельно. В наших примерах, таким образом, не обязательно было писать `#'+`, достаточно было бы одного символа `+`. Мы этой возможностью пользоваться не будем; попустительство подобного рода со стороны компонентов системы программирования, позволяющее безалаберность в работе и написание программного кода без понимания происходящего, с нашей точки зрения однозначно и безоговорочно вредно.

Ещё один часто применяемый встроенный функционал называется `mapcar`; первым его аргументом служит функция, а второй и последующие аргументы (всего их может быть сколько угодно, но не меньше двух) представляют собой некие списки; `mapcar` применяет заданную функцию сначала к первым элементам списка, потом ко вторым, затем к третьим и так далее. Например, результатом вычисления

```
(mapcar #'+ '(1 2 3 4) '(10 20 30 40) '(100 200 300 400))
```

будет список `(111 222 333 444)`, а результатом

```
(mapcar #'list '(1 2 3 4) '(10 20 30 40) '(100 200 300 400))
```

станет список списков:

```
((1 10 100) (2 20 200) (3 30 300) (4 40 400))
```

Представим теперь, что нам потребовалось с помощью того же `mapcar` выполнить над каждым элементом имеющегося списка какую-нибудь простую операцию, для которой в Лиспе нет встроенной функции; прекрасным примером такой операции может служить увеличение на три. Итак, есть список чисел, нужно получить такой же список, в котором соответствующие элементы получены из исходных прибавлением тройки; что делать?

Разумеется, можно написать пользовательскую функцию вроде

```
(defun plus3 (x) (+ x 3))
```

и использовать её как аргумент для `mapcar`: вычислив форму

```
(mapcar #'plus3 '(10 20 30 40 50))
```

мы получим список `(13 23 33 43 53)`. Но с этим решением возникают две проблемы. Во-первых, таких мелких функций в программе может потребоваться очень много, так что мы запутаемся в их именах. Во-вторых, что делать, если, скажем, число, на которое нужно увеличить каждый элемент списка, не известно, как в нашем примере, заранее (на момент написания программы), а вычисляется уже в ходе выполнения?

На помощь здесь приходят так называемые ***лямбда-функции*** — безымянные функциональные объекты, создаваемые «на лету» во время исполнения программы. Лямбда-функции обладают двумя привлекательными особенностями: во-первых, они не требуют имени и, как следствие, не загромождают множество используемых идентификаторов; во-вторых, тело лямбда-функции может содержать ***свободные переменные***, значение которых берётся из окружения.

Основой для конструирования лямбда-функции служит так называемый ***лямбда-список***. Первым элементом такого списка должен быть символ `lambda` — причём это не потому, что с ним ассоциирована какая-нибудь функция или спецформа, и не потому, что у него какое-то значение, а исключитель-

[12] См. сноску 7 на стр. 19.

[13] Напомним, что вариадическими называются функции (и, говоря шире, подпрограммы), которые можно вызывать для разного числа аргументов (фактических параметров).

но потому, что это вот конкретный символ `lambda`[14]. Остаток лямбда-списка похож на хвост формы `defun`: вторым элементом в нём выступает список символов — формальных параметров, а дальше следует тело функции, состоящее из одной или нескольких форм. Интересно, что лямбда-список (целиком) можно поставить первым элементом формы — например, результатом вычисления формы

```
((lambda (a b c) (+ a (* b c))) 10 3 5)
```

станет число 25; таким способом лямбда-списки применяются редко, но для нас здесь важнее соображение, что коль скоро нечто может быть первым элементом формы, к нему должна быть применима спецформа `function`. Это действительно так: именно `function` (обычно сокращаемая до `#'`) используется для превращения лямбда-списка в функциональный объект. Например, если нам потребуется прибавить 3 к каждому из элементов заданного списка, это можно будет сделать так:

```
(mapcar #'(lambda (x) (+ 3 x)) '(10 20 30 40 50))
```

Результатом вычисления этого выражения станет список `(13 23 33 43 53)`. Если же число, которое потребуется прибавлять, во время написания программы ещё не известно, можно написать функцию, которая будет прибавлять ко всем элементам списка *произвольное* число:

```
(defun add-to-list (num lst)
    (mapcar #'(lambda (x) (+ x num)) lst)
)
```

Теперь, например, вычисление выражения

```
(add-to-list 7 '(10 20 30))
```

даст список `(17 27 37)`. Этот пример примечателен тем, что в теле лямбда-функции используется *свободная переменная* `num`; в том месте программы, где создаётся лямбда-функция, эта переменная присутствует (это имя первого из двух формальных параметров функции `add-to-list`), и функция, созданная из лямбда-списка, использует именно её.

Поскольку функции в Лиспе выступают «первоклассными объектами», то есть обрабатываются наравне с любыми другими данными, их можно не только передавать в другие функции параметрами, но и возвращать из функций в качестве значений. Например:

```
(defun make-adder (n)
    #'(lambda (x) (+ n x))
)
```

Функция `make-adder`, получая параметром произвольное число, будет создавать *функцию, прибавляющую это число к своему (единственному) аргументу*; так, результатом вычисления `(make-adder 13)` станет функция, прибавляющая 13. Если последовательно вычислить формы

```
(setq plus13 (make-adder 13))
(funcall plus13 77)
```

— то результатом второй из них станет 90. Обратите внимание, что здесь мы не применяем к символу `plus13` никаких хитрых операций вроде «`'`» или «`#'`», поскольку объект функции является его *значением*. Если вы решите попробовать этот пример на SBCL, предварительно объявите `plus13` глобальной переменной с помощью `defvar`; в GCL и ECL это делать не обязательно.

Впрочем, при желании можно занести функцию не в значение символа, а в ссылку, предназначенную для функций. Следующие две формы дадут тот же результат, что и предыдущие, только для символа `plus13` будет задействована ссылка на ассоциированную функцию, при этом *значения* у него не появится:

```
(setf (symbol-function 'plus13) (make-adder 13))
(plus13 77)
```

В целом лямбда-функции устроены несколько сложнее, чем может показаться на первый взгляд; этому вопросу мы целиком посвятим следующий параграф. Пока же попробуем в первом приближении ответить на вопрос, почему всё-таки средствами Лиспа нельзя написать новую спецформу.

Обычно новички, задающиеся этим вопросом, рассуждают примерно так: «Функция отличается от спецформы тем, что все её аргументы вычисляются; но если бы её аргументы не вычислялись, она могла бы это сделать сама — например, с помощью `eval`, так что, наверное, было бы даже правильнее, если бы автоматически никакие аргументы не вычислялись, ведь тогда можно было бы писать и обычные функции, и такие, которые свои аргументы не вычисляют или вычисляют не все». Как правило, человек, рассуждающий подобным образом, просто не задумывался о том, как происходит применение функции к аргументам внутри функционалов, будь то `mapcar`, `apply` или любая другая функция, вынужденная *применять* функциональный объект — полученный через параметры или любым другим способом — к заданным аргументам. Определяющим тут окажется тот очевидный факт, что функциональный объект внутри функционалов *применяется к уже вычисленным аргументам*; функция, следовательно, никак не может сама вычислять свои аргументы, ведь она не знает, вызвали её из функционала или при вычислении обычной формы.

Именно поэтому при вычислении простой функциональной формы интерпретатор сам вычисляет все её элементы, кроме первого, а затем уже применяет к ним (то есть к *готовым* зачениям аргументов) функцию — точно так же, как это происходит и внутри функционалов. По той же причине **функционалы не могут работать ни со спецформами, ни с макросами — только с функциями**: в самом деле, и спецформы, и макросы так или иначе *управляют вычислением своих аргументов*, тогда как при применении функции к аргументам внутри функционала всё уже давно вычислено; если макрос или спецформа попытаются вычислить часть своих аргументов (так, как они делают это в обычных условиях), то результаты получатся совсем не такие, каких можно было бы ожидать.

11.1.9. Фунарг-проблема и лексическое связывание

Ранние версии Лиспа использовали простейший возможный подход к хранению значения символа: ссылка на это значение хранилась в самом объекте, представляющем символ. Такая ссылка в объектах символов существует и сейчас, а в Common Lisp есть даже специальная встроенная функция, позволяющая до этой ссылки добраться — `symbol-value` (по аналогии с рассмотренной выше `symbol-function`); как ни странно, *значением символа* в современных версиях Лиспа далеко не всегда считается тот объект, на который ссылается это вот «`value`», и тому есть нетривиальная, но весьма важная причина.

Чтобы понять, в чём тут дело, нам придётся продолжить исторический экскурс и посмотреть, как поступали интерпретаторы Лиспа, когда та или иная переменная (т. е. символ), уже имевшая значение, использовалась в роли локальной переменной — например, выступала формальным параметром функции. Простой подход, который первым приходит в голову — это сохранить где-нибудь (условно говоря, в стеке, хотя это совершенно не обязательно аппаратный стек) старые значения локальных переменных, присвоить им новые (локальные) значения — например, значения фактических параметров, к которым применяется функция, потом выполнить тело функции, а перед возвратом управления восстановить обратно старые значения переменных, использовавшихся в качестве локальных.

Такой подход называется ***динамическим связыванием***. Нельзя сказать, чтобы он был совсем неправильным: как мы уже отмечали (см. §11.1.1), до нашего времени сохранились два известных диалекта Лиспа — Emacs Lisp и AutoLISP (встроенные интерпретаторы в редакторе Emacs и в AutoCAD), использующие динамическое связывание. Ранние реализации Лиспа прекрасно существовали почти десять лет, пока не обнаружилось, что в некоторых случаях они ведут себя не так, как от них ожидается.

Суть проблемы можно проиллюстрировать на примерах. Для начала рассмотрим уже знакомую нам функцию, создающую *функции* с одним аргументом, прибавляющие к своему аргументу заданное число; в отличие от ранее рассмотренного примера, переменную, на которую нужно обратить особое внимание, мы назовём `thevar`:

[14] До сей поры мы видели только один подобный символ, роль которого определяется не значением и не функцией, а самим по себе объектом символа — `nil`. На самом деле Common Lisp вводит довольно много таких символов, но мы постараемся без них обойтись.

```
(defun make-adder (thevar)
    #'(lambda (x) (+ thevar x))
)
```

Теперь для простоты картины занесём в `thevar` глобальное значение `100`:

```
(setq thevar 100)
```

После этого воспользуемся функцией `make-adder`, чтобы создать «прибавлялку пяти»:

```
(setq plus5 (make-adder 5))
```

и попытаемся эту «прибавлялку» задействовать:

```
(funcall plus5 10)
```

Если в нашем интерпретаторе Лиспа используется динамическое связывание, результат может нас несколько удивить: вместо ожидаемого значения `15` мы получим `110`. Между прочим, это даже можно попробовать в любом интерпретаторе Лиспа, включая ECL, GCL и SBCL — достаточно установить для `thevar` динамический способ связывания, выполнив `(defvar thevar)`.

Попробуем понять, что произошло. Когда мы вызвали функцию `make-adder` с параметром `5`, переменная `thevar`, выступающая в ней формальным параметром, временно — то есть на время выполнения `make-adder` — получила значение `5`; при этом старое значение `thevar` — число `100` — интерпретатор где-то сохранил, а после завершения функции восстановил его обратно. Следовательно, когда мы — уже, разумеется, после завершения `make-adder` — решили наконец воспользоваться созданной «прибавлялкой», переменная `thevar` имела значение `100`, а вовсе не `5`. Телом нашей «прибавлялке» служит тело лямбда-списка, из которого она была создана, а оно, как мы помним, состоит из единственной формы `(+ thevar x)`; переменная `x` тут обозначает формальный параметр, то есть она локально получила значение, переданное фактическим параметром, в нашем случае это число `10`, здесь всё в порядке; но переменная `thevar` здесь свободная, то есть, во всяком случае, она не является локальной по отношению к функции-«прибавлялке», так что интерпретатор обратился к ней и получил её *текущее* значение, а оно было на этот момент равно `100`; менять его на `5` и обратно было в этот раз некому.

Между прочим, если бы мы не присвоили переменной глобальное значение, всё получилось бы ещё интереснее: попытка использования «прибавлялки» вызвала бы ошибку, поскольку `thevar` перед началом выполнения `make-adder` никакого значения не имела, и после окончания выполнения именно это состояние переменной — отсутствие значения — будет восстановлено.

Приведённый пример демонстрирует так называемую ***восходящую фунарг-проблему*** (англ. *upwards funarg problem*), но это не единственный недостаток динамического связывания. Попробуем привести другой пример. Пусть некая функция, получив значение через формальный параметр, в роли которого выступает всё тот же символ `thevar`, создаёт ту же самую «прибавлялку», но не возвращает её, а передаёт параметром в другую функцию, где тоже задействована переменная `thevar`. Для простоты картины наша вторая функция будет получать два значения — функцию и аргумент для неё, и всё, что она будет делать — это применять заданную функцию к заданному аргументу; важно тут лишь то, что формальный параметр, через который получается аргумент, будет называться, опять же, `thevar` — точнее говоря, он будет обозначен всё тем же символом `thevar`:

```
(defun f2 (func thevar) (funcall func thevar))
```

Что касается основной функции, то она может быть, например, такой:

```
(defun sample (thevar)
    (f2 #'(lambda (x) (+ x thevar)) 1000)
)
```

Рассмотрим внимательно, что будет происходить при вычислении `(sample 7)` при условии, что интерпретатор использует динамическое связывание (как минимум для переменной `thevar`). На время выполнения тела функции `sample` эта переменная получает значение `7`; дальше для вызова `f2` интерпретатор должен вычислить значения её параметров, при этом константа `1000` вычисляется сама в себя, а первый аргумент — спецформа `function`, применённая к лямбда-списку — при вычислении даёт объект *функции*, от которой мы ожидаем, что она, принимая один аргумент, будет возвращать число, бóльшее на значение `thevar`, то есть в нашем случае на `7`.

К полученным значениям параметров применяется функция `f2`, про которую мы помним, что вроде бы она должна свой первый аргумент применить ко второму. Коль скоро первый аргумент — «прибавлялка семи», а второй — тысяча, получиться должно, по идее, число `1007`. *Если бы в описании `f2` формальные параметры назывались как-нибудь иначе, именно так бы всё и было*, то есть мы действительно получили бы `1007`; но в нашем определении `f2` имя `thevar` используется для второго формального параметра. Как следствие, на время выполнения `f2` символ `thevar` получает значение `1000`, и вот тут-то как раз и вычисляется тело нашей «прибавлялки», ссылающееся на `thevar` как на свободную переменную. Полученное в итоге значение `2000` (вместо ожидавшегося `1007`) может стать для программиста полным сюрпризом.

Между прочим, настоятельно рекомендуем читателю попробовать вбить указанные формы (описания функций `f2`, `sample` и вызов функции `sample`) в интерпретатор Лиспа, получить значение `1007`, затем закрыть интерпретатор, снова запустить его, выполнить `(defvar thevar)`, тем самым сделав связывание для `thevar` динамическим, снова вычислить те же самые три формы и убедиться, что полученное значение в этот раз — `2000`. Этот нехитрый эксперимент позволит избавиться от остаточного неверия в происходящее, которое — можете не сомневаться — у вас пока ещё есть.

Здесь мы продемонстрировали ***нисходящую фунарг-проблему***. Стоит признать, что этот вариант фунарг-проблемы более убедителен и в целом более эффектен, нежели предыдущий. Во-первых, здесь получается, что результат *зависит от имени локальной переменной в другой функции*, а это идёт вразрез с самой идеей локальных переменных: их имена не должны никого волновать за пределами той части программы, где они локализованы. Во-вторых, именно в нисходящем случае наглядно видно, что попросту взять и запретить свободные переменные в лямбда-списках, как это иногда предлагают сделать, — значит потерять очень важные возможности (вспомним, как мы с помощью `mapcar` увеличивали на заданное число все элементы списка).

Если не рассматривать маргинальные случаи вроде упоминавшихся выше встроенных диалектов Лиспа, современные реализации, естественно, с фунарг-проблемой справляются; достигается это применением ***лексического связывания***. В диалектах, основанных на Common Lisp, лексическое связывание используется по умолчанию, но для отдельных переменных, как мы видели, его можно отключить, вернувшись к связыванию динамическому. Другие диалекты ушли дальше и вообще отбросили возможность динамического связывания; пример тому — Scheme.

Чтобы понять семантику лексического связывания, можно представить себе некие таблицы из двух колонок: в первой колонке — ссылка на символ, во второй — ссылка на его значение, а в конце таблицы — ещё и ссылка (возможно, пустая) на вышестоящую таблицу такого же вида. Такая таблица называется ***лексическим контекстом***, каждая её строчка — ***лексической связью*** или ***биндингом*** (последнее есть транслитерация английского *binding*), а ссылка в конце таблицы нужна, чтобы создавать *вложенные* контексты, в которых к уже имеющимся связям объемлющего контекста добавляются дополнительные связи, как раз и образующие вложенный контекст; ссылка используется в таблице вложенного контекста, чтобы не потерять контекст объемлющий. Теперь, вычисляя символ как S-выражение, то есть выполняя обращение к значению переменной, интерпретатор сначала интересуется, нет ли для этого символа связи в текущем лексическом контексте, а если у текущего контекста есть контекст объемлющий — то нет ли соответствующей связи там (и так пока цепочка вложенных контекстов не кончится), и лишь только в случае, если в контекстах так ничего и не нашлось, обращается к динамическому (глобальному) значению, которое, как мы помним, хранится в самом символе.

Следует подчеркнуть, что такая конструкция с поиском по таблицам лексических контекстов приведена нами лишь для иллюстрации семантики. Если действительно (буквально) реализовать такую схему, интерпретатор будет работать недопустимо медленно, ведь ему придётся выполнять линейный поиск символа в контекстах каждый раз при обращении к переменной (любой!). Как на самом деле реализуются лексические значения переменных — вопрос сложный и заведомо выходящий за рамки нашей книги. Нам достаточно знать, что работает это всё семантически точно так же, как если бы

контексты и впрямь представлялись таблицами лексических связей — только намного быстрее.

Теперь, когда переменная должна получить локальное значение, нет никакого резона что-то делать со значением, хранимым внутри объекта символа: вместо этого в контекст добавляется лексическая связь для данной переменной.

Надо отметить, что объекты лексических связей используются не только при обращении к переменной, но и при присваиваниях: например, `setq` при наличии данного символа в активном лексическом контексте изменит именно значение внутри контекста, а не то значение, которое в самом символе.

Проиллюстрировать наличие у символа «более чем одного значения» — точнее, значения, которое не совпадает с хранимым в самом символе — можно на следующем примере. Вспомним, что к той ссылке, которая хранится в символе, можно обратиться с помощью функции `symbol-value`. Напишем теперь функцию, которая будет использовать символ `var` в роли формального параметра и создавать точечную пару из *действующего* значения этого символа и *динамического* значения того же символа:

```
(defun var-values (var)
    (cons var (symbol-value 'var))
)
```

Теперь для большей наглядности вне функций занесём в эту переменную значение `"global"`, а потом вызовем функцию `var-values` с параметром `"local"`:

```
(setq var "global")
(var-values "local")
```

Результат обращения к функции будет такой: `("local" . "global")`. SBCL при выполнении `setq` выдаст предупреждение, но вы можете его благополучно проигнорировать; GCL и ECL предупреждения не выдадут.

Введя лексические контексты и более-менее сообразив, как они будут работать, мы, можно надеяться, готовы к объяснению одного из самых нетривиальных понятий функционального программирования — ***замыканий*** (англ. *closures*). По сути замыкание — это тело функции, снабжённое лексическим контекстом, или, иначе говоря, набором лексических связей, действовавших на момент создания функции. **Коль скоро реализация Лиспа поддерживает лексическое связывание, каждый объект функции в такой реализации на самом деле представляет собой замыкание**. Это касается в том числе и встроенных функций, просто их контекст пустой — как и контекст функций, введённых с помощью `defun` на верхнем уровне; но пустой контекст не означает *отсутствия* контекста.

При выполнении любой функции перед передачей управления её телу интерпретатор сначала делает её контекст текущим (активным), запомнив при этом, какой контекст был активным раньше, чтобы по окончании выполнения функции вернуть контексты обратно. Это касается в том числе и пустых контекстов; следовательно, **функция (если это именно функция, а не макрос и не спецформа) не может никаким способом получить доступ к локальным переменным того, кто её вызвал: у неё свой контекст, у вызывающего — свой**. На спецформы это правило не распространяется, они ведь не функции и своего контекста не имеют, да и вообще они реализованы не на Лиспе, а на том языке, на котором написан интерпретатор, и являются частью ядра интерпретатора; спецформы, в частности, сами могут манипулировать контекстами — создавать их, изменять и переключать.

Макросы мы не рассматриваем, но на всякий случай — к примеру, если читатель захочет освоить их самостоятельно — отметим, что свои контексты у них есть; при этом тело макроса (которому, напомним, параметры переданы невычисленными) работает в *своём* контексте, но результат его вычисления (т. е. макрорасширения) — полученный код, который как раз и должен выполниться — будет выполнен уже в контексте вызывающего.

Форма `function`, обычно обозначаемая символом `#'`, на самом деле создаёт именно замыкание, а не что-то иное; впрочем, то же самое можно сказать и про `defun`. Это очевидным образом объясняет, как работают те же «прибавлялки» из примеров, приводившихся выше. Напомним, что мы создавали их с помощью функции

```
(defun make-adder (thevar)
    #'(lambda (x) (+ thevar x))
)
```

Если, например, вызвать её с параметром `33`, то при вычислении её тела форма `function` (`#'`) создаст замыкание, состоящее из, во-первых, тела лямбда-списка (включая список параметров) и, во-вторых, из контекста, в котором имеется лексическая связь (биндинг) символа `thevar` со значением `33`. Когда полученную таким образом «прибавлялку» (то есть функцию-замыкание) мы применяем, скажем, к числу `27`, то интерпретатор сначала делает активным контекст из нашего замыкания, а затем создаёт вложенный контекст для связи формальных параметров с фактическими значениями — в нашем случае здесь будет связь переменной `X` со значением `27`. После этого уже в новом контексте вычисляется единственная форма из тела лямбда-списка — `(+ thevar X)`; значения обоих символов интерпретатор возьмёт из контекста — точнее, `x` из активного контекста, а `thevar` — из объемлющего.

Сказанное позволяет продолжить начатый в предыдущем параграфе разговор о том, почему функция не может сама вычислять свои собственные параметры. Оказывается, даже если бы она каким-то образом могла узнать, когда их нужно вычислять, а когда — нет, всё равно бы ничего не получилось, ведь *для вычисления значений фактических параметров нужен доступ к лексическому контексту вызывающего*, а этого доступа у функции, оказывается, нет, ведь во время вычисления тела функции активен контекст этой функции, а не какой-то другой.

Правила смены лексических контекстов приходится учитывать также, например, и при использовании функции `eval`. Поскольку формально это именно функция, а не спецформа и не макрос, доступа к контексту вызывающего у неё нет. Поэтому, например, следующая функция

```
(defun demo-eval-symb (x) (eval 'x))
```

вернёт глобальное значение `x`, если таковое есть, если же его нет — произойдёт ошибка; при этом значение параметра, с которым мы вызвали функцию, вообще ни на что не повлияет.

Ещё один хороший иллюстративный пример на эту тему — функция `set`; в отличие от `setq`, она свой первый аргумент не экранирует, что позволяет, например, уже во время исполнения определить, какому символу присваивать значение. С другой стороны, `set` — это функция, а не спецформа; поэтому изменить локальное (лексическое) значение переменной она не может, её область применения ограничена глобальными переменными.

Для использования в дальнейших примерах нам желательно рассмотреть спецформу `let`, позволяющую вводить дополнительные локальные переменные; фактически эта форма создаёт вложенный лексический контекст. Первый элемент этой формы — естественно, символ `let`; второй элемент представляет собой список вводимых локальных переменных, причём этот список может содержать элементы двух видов: символы (в этом случае символ вводится в качестве локальной переменной без начального значения) и *списки из двух элементов* — символ и начальное значение для него. Остальные элементы формы `let` представляют собой вычислимые формы, которые будут одна за другой вычислены в новом лексическом контексте; результат вычисления последней из них, как водится в таких случаях, объявляется результатом всей формы `let`.

Например, в каком-нибудь контексте, где `thelist` есть некий список чисел, следующая форма вычислит за один проход сумму и произведение его элементов и вернёт составленную из результатов точечную пару:

```
(let ((s 0) (p 1))
    (mapcar
        #'(lambda (x) (setq s (+ s x)) (setq p (* p x)))
        thelist
    )
    (cons s p)
)
```

То же самое можно сделать чуть по-другому — не инициализировать переменные, а присвоить им начальные значения:

```
(let (s p)
    (setq s 0)
    (setq p 1)
    (mapcar
 ; ...
```

Начальные значения локальных переменных можно задать, естественно, не только константами, но и произвольными вычислимыми формами. Интересно, что **начальные значения для переменных в форме `let` вычисляются в объемлющем форму лексическом контексте**, так что «новые» переменные во время этого вычисления недоступны; в частности, если какая-либо из переменных, вводимых в качестве локальных формой `let`, имеет значение в объемлющем контексте, то при вычислении начальных значений используется её значение из объемлющего контекста. Новый контекст, содержащий все нововведённые переменные, создаётся и активируется лишь после того, как вычисления всех инициализирующих форм будут окончены.

Лисп также предусматривает форму с именем `let*`, которая действует иначе: переменные из списка обрабатываются последовательно — для каждой из них вычисляется форма, задающая начальное значение, после чего переменная заносится в контекст, а начальное значение следующей переменной вычисляется уже в рамках этого контекста. В Scheme есть варианты ещё более интересные.

Кроме того, локальными могут быть не только значения, но и функции, и даже макросы; для их введения используются формы `flet` и `macrolet`. Их мы рассматривать не будем.

Имея в своём распоряжении форму `let`, мы сможем проиллюстрировать одну важную особенность замыканий, а именно тот факт, что **при создании замыканий контексты и связи не копируются**. Это означает, что, коль скоро две (или больше) функции созданы в одном и том же контексте (или в контекстах, вложенных в один и тот же объемлющий контекст), у полученных замыканий могут быть общие (*физически общие*!) связи.

Для примера создадим с помощью `let` лексический контекст, состоящий из одной переменной `counter`, и в нём построим две функции, которые будут с этой переменной работать. При этом воспользуемся тем, что в Лиспе `defun` не обязана быть формой верхнего уровня:

```
(let ((counter 0))
    (defun seq-next () (setq counter (+ 1 counter)))
    (defun seq-reset () (setq counter 0))
)
```

Итак, теперь у нас есть две функции — `seq-next` и `seq-reset`, обе вызываются без параметров. Первая, как видим, увеличивает переменную `counter` на единицу и возвращает полученное значение (напомним, `setq` возвращает последнее из присвоенных значений, а если оно всего одно, как чаще всего и бывает — то это одно); вторая функция заносит в `counter` ноль. Теперь у нас может состояться следующий диалог с интерпретатором (символ `>` здесь — приглашение интерпретатора):

```
>(seq-next)
1
>(seq-next)
2
>(seq-next)
3
>(seq-next)
4
>(seq-next)
5
>(seq-reset)
0
>(seq-next)
1
>(seq-next)
2
>(seq-next)
3
```

Этот пример наглядно показывает, что функции `seq-next` и `seq-reset` используют одну и ту же (физически!) связь переменной `counter` с её значением.

Для нашего разговора о парадигмах этот пример интересен тем, что получившаяся сущность сильно напоминает объект с двумя методами. На эту тему автор этих строк от одного из своих студентов даже слышал как-то раз некую притчу в восточном стиле:

> *Как-то раз к Мастеру пришёл его ученик и сказал: «Учитель, я изучил объектно-ориентированное программирование! Мне кажется, объекты — это прекрасный инструмент, который позволит моему искусству программирования стать ещё лучше!» — на что Мастер стукнул его палкой и недовольно ответил: «Сколько раз тебе говорить, пустая голова, что объекты — это просто недоделанные замыкания!»*
>
> *Ученик вернулся в свою келью и долго занимался в одиночестве, пока не решил, что готов к следующей беседе с наставником. Тогда он снова пришёл к Мастеру и сказал: «Учитель, я изучил функциональное программирование, и мне кажется, что замыкания — это одна из самых изящных концепций, что я знаю!» — на что Мастер стукнул его палкой и недовольно пробурчал: «Сколько раз тебе говорить, пустая голова, что замыкания — это просто недоделанные объекты!»*
>
> *После этого ученик достиг просветления.*

Оригиналом этого «коана», по-видимому, послужило письмо автора Anton van Straaten в списке рассылки `ll1-discuss` (`https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html`). На семинаре «Парадигмы программирования» его в вольном переводе на русский пересказал В. Мелешкин, которого автор хотел бы за это поблагодарить. Тот текст, что приведён здесь, на самом деле совсем не похож ни на английский оригинал, ни на то, что прозвучало на семинаре — осталась лишь основная идея про учителя, ученика и замыкания с объектами, но восточные истории часто изменяются подобным образом: автору попадалось больше десятка версий одной только притчи про самурая и мастера чайной церемонии. Заодно автор хотел бы поблагодарить И. Костарева, который помог найти ссылку на английский оригинал.

11.1.10. О фунарг-проблеме в других языках

Хотя фунарг-проблема была впервые обнаружена именно в Лиспе, она для этого языка никоим образом не специфична. Для её возникновения достаточно всего двух условий: в языке должно быть можно работать с подпрограммами как со значениями (например, указателей на функции, как в Си, вполне хватит), а сами подпрограммы делать вложенными (друг в друга или в какую-то иную сущность) — так, чтобы они имели доступ к локальным переменным объемлющего контекста, каковой контекст имеет свойство возникать и исчезать.

В функциональных языках переменные и обращения к ним обычно изначально реализуются с учётом фунарг-проблемы; при этом переменная становится достаточно сложным объектом — во всяком случае, про переменные того же Лиспа никак нельзя сказать, что это просто имена, которые идентифицируют области памяти.

Совершенно иначе обстоят дела с фоннеймановскими языками, парадигматические свойства которых близки к свойствам языка ассемблера — с такими, например, как Паскаль и Си. Здесь именованные переменные бывают ровно двух видов — глобальные (их имена есть то же самое, что метка в языках ассемблеров, то есть глобальная переменная есть именованная область памяти в соответствующем сегменте) и локальные (как мы знаем, имена локальных переменных в Паскале и Си обозначают постоянные смещения относительно реперной точки стекового фрейма). Эти языки допускают работу с указателями, не скрывая того факта, что указатель хранит адрес области памяти; всякая переменная имеет адрес, который можно узнать и поместить в указатель, и т. д. Иначе говоря, семантика этих языков описана так, что переменная может быть только областью памяти и более ничем. В такой ситуации нет очевидных способов для реализации замыканий; но если упомянутые выше два условия окажутся выполненными, реализация замыканий потребуется. Язык Паскаль допускает вложенные процедуры и функции; рассказывая о Паскале в первом томе, мы не упоминали эту возможность, и сейчас читатель, возможно, поймёт, почему.

В спецификации Паскаля, исходно предложенной Никлаусом Виртом, функции и процедуры ни в каком виде не были «первоклассными» или вообще какими бы то ни было объектами, которые можно обрабатывать. Надо сказать, что даже при этом реализация вложенных подпрограмм резко усложняет работу со стеком, ведь подпрограмма, вложенная в другую подпрограмму, требует доступа не только к своему стековому фрейму, но и к фреймам всех подпрограмм, в которые она вложена — для обращения к их локальным переменным. Пространство стека при этом перестаёт быть простой последовательностью фреймов, поскольку фреймы вслед за подпрограммами получают возможность вложенности друг в друга. Кстати, именно для орга-

низации таких вот «фреймов во фреймах» предназначен загадочный второй параметр машинной команды `enter` из системы команд i386.

Стандарт Паскаля ввёл «процедурные параметры», то есть возможность передать процедуру или функцию через параметр другой процедуре или функции. Создатели стандарта поосторожничали и не стали вводить *переменные* процедурного типа, да и вообще процедурный тип как таковой; в частности, процедуру или функцию согласно этой спецификации можно было передать в подпрограмму параметром, но нельзя было вернуть из функции — ни через возвращаемое значение, ни через var-параметр. Это открывало возможность для возникновения нисходящей фунарг-проблемы, но полностью исключало появление проблемы в её восходящем варианте.

Впрочем, всё это осторожничанье длилось недолго: авторы реализаций Паскаля обобщили возможность, введённую в стандарте — в реализациях появились «процедурный» и «функциональный» тип переменных. В первом томе мы их не рассматривали, поскольку трудно придумать задачу, в которой они потребуются, при этом достаточно простую, чтобы её мог понять начинающий программист — а первый том был рассчитан именно на этот уровень. Так или иначе, в Паскале такие переменные есть и устроены буквально так же, как указатели на функции в Си — то есть физически это указатели, содержащие адреса подпрограмм в памяти.

Надо сказать, что нисходящая фунарг-проблема, если рассматривать её в контексте компиляции фоннеймановских языков, не столь страшна, как восходящая: на момент вызова подпрограммы — пусть даже это вложенная подпрограмма, переданная в другую подпрограмму, и пусть даже в её теле есть обращения к локальным переменным объемлющего контекста — *всё ещё существует стековый фрейм, содержащий значения всех этих локальных переменных контекста.* Если вложенная подпрограмма передаётся параметром в другую подпрограмму, вложенную в тот же контекст, то замыкание в явном виде можно не строить, а проблемы доступа к объемлющему контексту решаются точно так же, как и при обычных вызовах вложенных подпрограмм; впрочем, нельзя сказать, что это очень просто — но как-то это сделать всё же можно.

Совсем иначе обстоят дела, когда локальную подпрограмму незадачливый программист пытается вернуть из функции (или из процедуры через var-параметр, легче от этого не станет), а сама эта подпрограмма использует локальные переменные объемлющей подпрограммы — той, откуда её возвращают. Здесь уже, как говорится, мёртвому припарками не поможешь: при возврате из подпрограммы её стековый фрейм исчезает вместе со всеми локальными переменными. На этом месте иногда слушатели спрашивают, почему бы не выстроить аналог замыкания — но позвольте, *где* его выстраивать? Создать его в стеке не получится, мы ведь *возвращаемся*, то есть стек сейчас должен убывать, а не расти. Ну а динамическую память компиляторы статических языков, включая Паскаль, традиционно никогда не применяют для своих нужд. В Си, как мы знаем, компилятор вообще не знает о существовании динамической памяти, но и Паскаль сам по себе устроен так, что компилятору динамическая память не нужна, она задействуется только по явно выраженной воле программиста. Короче говоря, попытка решить восходящую фунарг-проблему в статически компилируемых языках противоречит их сути.

В реально существующих реализациях Паскаля эту трудность обошли просто и цинично: вложенные подпрограммы невозможно использовать в роли значения процедурного или функционального типа, компилятор при такой попытке выдаёт ошибку. В результате их невозможно ни вернуть из подпрограммы, ни передать параметром в другую подпрограмму. «Объектами первого класса» выступают только подпрограммы верхнего уровня, ну а у них контекст всегда один и тот же — глобальный, так что «замыкать» подпрограммы, присоединяя к телу контекст, не нужно, контекст и так известен.

Создатели языка Си с самого начала поступили проще: указатели на функции там присутствовали всегда, но зато не было вложенных функций, так что любое вхождение свободной переменной в функции могло подразумевать только обращение к глобальной переменной, более ничего. Как и в случае с подпрограммами верхнего уровня в Паскале, контекст тут один на всех, а замыкания оказываются заведомо не нужны. Однако, по-видимому, мир устроен так, что при наличии в нём неких граблей всегда находятся желающие по этим граблям прогуляться. Компилятор `gcc`, который мы использовали при изучении Си и Си++, поддерживает ряд «расширений GNU» (*GNU extensions*), в число которых входят вложенные функции. На всякий случай: **никогда, ни для чего, ни за что не используйте вложенные функции в Си, даже если ваш компилятор это позволяет**! Если сомневаетесь, сейчас мы попытаемся вас убедить.

Рассмотрим для начала простенький пример — функцию, которая подсчитывает сумму и произведение элементов заданного целочисленного массива. Возвращать оба значения эта функция будет через параметры; итого у неё будет четыре параметра — адрес начала массива, длина массива и два указателя — на переменные, куда нужно записать сумму и произведение. Задача вроде бы простая и может быть решена очевидным способом через цикл или чуть менее очевидным способом через рекурсию, но мы воспользуемся методом совсем не очевидным: организуем просмотр массива как редукцию последовательности, причём «для пущей ценности» для хранения «затравки» и текущего результата редукции воспользуемся переменной, «глобальной» по отношению как к функции, выполняющей редукцию, так и к функциям, из неё вызываемым (вычисляющим сумму и произведение). На самом деле переменная, конечно, будет не глобальная, а локальная — в объемлющей функции:

```
void sumprod(int *vec, int n, int *sum, int *prod)
{
    int accum;
    int plus(int a) { return accum + a; }
    int times(int a) { return accum * a; }
    void reduce(int *vec, int len, int (*op)(int))
    {
        if(!len)
            return;
        accum = op(*vec);
        reduce(vec+1, len-1, op);
    }
    accum = 0; reduce(vec, n, &plus); *sum = accum;
    accum = 1; reduce(vec, n, &times); *prod = accum;
}
```

Компилятор gcc такую функцию вполне примет (конечно, если не запретить GNU extensions соответствующим флагом командной строки), и она даже будет работать — вопрос в том, какой ценой это достигается. Если вложенная функция не обращается к локальным переменным объемлющей функции, то её работоспособность достигается «бесплатно», поскольку по своей работе она ничем от обычной (не-вложенной) функции не отличается; но это не наш случай, у нас все три функции обращаются к переменной `accum`. Это уже сложнее, но тоже не фатально: например, можно было бы воспользоваться теми самыми вложенными фреймами, при этом каждая функция знала бы, какое положение занимает объемлющий фрейм относительно её собственного вложенного. Впрочем, `gcc` поступает проще: при вызове вложенной функции непосредственно из объемлющей передаёт ей адрес объемлющего фрейма через регистр (в конвенции CDECL используется регистр `ECX`).

Настоящие проблемы начинаются, лишь когда кто-то берёт *адрес* вложенной функции. В этот момент компилятор утрачивает контроль за тем, кто и когда функцию будет вызывать; но ведь перед тем, как её тело получит управление, нужно, чтобы в регистре `ECX` оказалось правильное значение. Решается этот вопрос весьма нетривиальным способом. Там, где в тексте программы встречено выражение, подразумевающее взятие адреса вложенной функции, компилятор вставляет машинный код, создающий *в стеке* (!) фрагмент машинного кода (!), обязанность которого — занести правильное значение в `ECX` (на других архитектурах — в какой-то другой регистр), после чего передать управление на начало тела вызываемой функции. Что касается выражения с взятием адреса, то оно вместо настоящего адреса тела функции возвращает адрес этого вот фрагмента в стеке; сам этот фрагмент называется ***трамплином*** (*trampoline*). Именно такие трамплины создаются при компиляции нашего примера, когда компилятор обрабатывает выражения `&plus` и `&times`.

Основной момент, который следует осознать относительно трамплинов, состоит в том, что они создаются *в стековом фрейме той функции, где произошло взятие адреса вложенной функции* — и, как следствие, существуют лишь до момента её завершения, после чего исчезают вместе с её фреймом точно так же, как и её локальные переменные. В этом несложно убедиться, остановив компилятор на стадии генерации ассемблерного кода (напомним, для `gcc` это достигается флажком `-S`). На просторах Интернета вы можете

встретить утверждение, будто `gcc` написан столь изощрённо, что умеет решать фунарг-проблему для вложенных функций в обе стороны, но на самом деле это не просто не так, всё гораздо хуже: если вы попытаетесь вернуть из функции адрес её вложенной функции, никто вам не выдаст даже предупреждения, и, более того, при некоторых условиях эта конструкция даже будет «работать» — но лишь по той причине, что исчезновение стекового фрейма само по себе не ведёт к очистке занимаемой им памяти.

Например, следующая программа почти наверняка отработает без аварий и выдаст ожидавшееся число `13`:

```
#include <stdio.h>
typedef int (*funcptr)(int);
funcptr make_adder(int a)
{
    int adder(int t) { return t+a; }
    return &adder;
}
int main()
{
    funcptr a3 = make_adder(3);
    printf("%d\n", (*a3)(10));
    return 0;
}
```

Чтобы узнать, насколько эта программа далека от того, чтобы быть корректной *на самом деле*, достаточно прогнать её под управлением `valgrind`. На консоли вы при этом увидите целую простыню предупреждений об обращении к освобождённой области стека и о том, что в коде принимаются решения об условных переходах на основании неинициализированных значений. Можно поступить нагляднее: создать в `main` ещё одну «добавлялку», вот так:

```
int main()
{
    funcptr a3 = make_adder(3);
    funcptr a7 = make_adder(7);
    printf("%d %d\n", (*a3)(10), (*a7)(10));
    return 0;
}
```

Эта программа тоже успешно откомпилируется и отработает без аварий, вот только вместо ожидавшихся чисел `13 17` она, скорее всего[15], напечатает `17 17`. Происходит это по весьма, если подумать, простой причине: когда `make_adder` работает второй раз, стековый фрейм от её первого вызова как раз только что уничтожен, так что её новый фрейм располагается точно на том же месте, и на том же месте создаётся трамплин, вот только в переменной `a` — которая тоже расположена на том же месте, что и раньше — теперь уже `7`, а не `3`. Указатели `a3` и `a7` содержат, очевидно, один и тот же адрес — тот адрес в ныне неиспользуемой (!) области стека, где во время обоих вызовов `make_adder` располагались трамплины.

Чтобы вся эта шаткая конструкция рухнула, достаточно ввести какую-нибудь промежуточную функцию, которая благополучно затрёт своими локальными переменными несчастный трамплин, например, так:

```
void eraser(funcptr ff)
{
    char c[50];
    int i;
    for(i = 0; i < sizeof(c); i++)
        c[i] = i;
    printf("%d\n", (*ff)(10));
}
int main()
{
    funcptr a3 = make_adder(3);
    eraser(a3);
    return 0;
}
```

У автора этих строк сей вариант свалился с диагностикой «`Illegal instruction (core dumped)`»; у вас он может упасть как-нибудь иначе, но отработать корректно у этой программы шансов нет никаких от слова «совсем».

Итак, восходящую фунарг-проблему `gcc` не решает и никакие аналоги замыканий создавать не умеет, это миф — и, что особенно неприятно, компилятор никак не сопротивляется попыткам вернуть из функции адрес локального трамплина, обречённого на исчезновение (при том что попытка вернуть адрес локальной *переменной* всё же вызывает предупреждение). Но даже если не пытаться вернуть из функции адрес трамплина, как мы только что сделали, а использовать трамплин лишь до тех пор, пока функция, породившая его, остаётся активна (то есть стековый фрейм, содержащий трамплин, продолжает существовать), всё равно остаётся одна проблема. **Разрешать исполнение кода в области стека — неудачная идея с точки зрения безопасности**, и в современных условиях это довольно часто оказывается запрещено. Последние версии `gcc` предлагают альтернативную (и ещё более замороченную) технику, основанную на неких «дескрипторах функций», при этом сами создатели `gcc` подтверждают, что эта техника совместима далеко не с любым кодом.

Вывод из нашего обсуждения напрашивается довольно простой: **работая на статически компилируемых (фоннеймановских) языках, забудьте о возможности вложенных подпрограмм**. Эта история, что называется, из другой сказки.

Довольно забавное проявление фунарг-проблемы можно наблюдать в Си++ с методами классов. Если временно абстрагироваться от нашего знания о параметре `this`, можно заметить, что методы классов *вложены в класс* в том числе и как в *контекст* — им доступны поля объекта, при этом объекты имеют свойство появляться и исчезать. Продолжая игнорировать `this` (что в целом полностью аналогично тому, как мы игнорировали локальные переменные функции, пытаясь при этом взять адрес вложенной функции), мы можем, пожалуй, захотеть взять адрес функции-метода и присвоить его указателю на функцию с таким же профилем, как у этого метода (опять же, без учёта существования `this`). Пока мы вызываем одни методы из других, мы остаёмся в контексте одного и того же объекта, но если мы отдадим адрес метода куда-то ещё, и где-то он будет вызван, то откуда, спрашивается, при этом возьмётся нужный методу контекст (т. е. объект)?!

В Си++ эта проблема решена самым лобовым из возможных способов: взятие адреса метода (если только он не статический — но у статических методов нет объекта, для которого они вызываются) даёт адресное выражение, тип которого не имеет ничего общего с указателем на обычную функцию. Полученная сущность называется «указатель на функцию-член» (*member function pointer*), и правила работы с ней совершенно иные, нежели для обычных функциональных указателей; вызов метода через указатель на метод производится специальной операцией `.*`, предполагающей указание (явное) объекта, для которого вызывается метод. Мы не рассматривали эту сущность, поскольку она крайне редко бывает нужна; заинтересованный читатель легко найдёт нужную информацию по указанным ключевым словам. Так или иначе, ни о каких замыканиях здесь тоже речи не идёт: контекст (в данном случае — объект) программисту приходится указывать самому.

11.1.11. Редукция списков

Редукцию (свёртку) последовательностей мы уже обсуждали как пример возможностей рекурсии (см. §9.2.3), но пока мы были вынуждены обходиться возможностями чистого Си, разбираемые примеры оставались неубедительными. Позже, применив для редукции шаблоны Си++, мы смогли привести примеры более эффектные, но и это была скорее демонстрация возможностей шаблонов, а не свёртки как приёма. Всё-таки свёртка относится скорее к области функционального программирования, и в полной мере её возможности раскрываются в программах на соответствующих языках. Теперь, зная Лисп, мы можем наконец продемонстрировать свёртку в её, если можно так выразиться, естественной среде обитания.

Для начала напишем функции, реализующие левую и правую редукцию. В Common Lisp есть функция `reduce`, способная делать и то и другое, но её мы трогать не будем; свои функции назовём `lreduce` и `rreduce` — таких имён спецификация Common Lisp не предусматривает. Итак:

[15] Оборот «скорее всего» тут использован по той причине, что всерьёз предсказывать результаты выполнения столь вопиюще ошибочной программы было бы не совсем корректно — формально говоря, её результат неопределён.

```
; reduce.lsp
(defun lreduce (fun lst init)
    (cond
        ((null lst) init)
        (t
            (let ((next (funcall fun init (car lst))))
                (lreduce fun (cdr lst) next)
            )
        )
    )
)
(defun rreduce (fun lst init)
    (cond
        ((null lst) init)
        (t (funcall fun (car lst) (rreduce fun (cdr lst) init)))
    )
)
```

Сразу же стоит обратить внимание, что реализация `lreduce` — хвостовая, а реализация `rreduce` — нет; как следствие, при прочих равных сто́ит правой свёртке предпочесть левую. Эту разницу между левой и правой свёрткой мы отметили ещё при обсуждении примеров свёртки на чистом Си (см. стр. 20); она не зависит от языка реализации.

Начнём с простых примеров использования. Результатом вычисления формы

```
(rreduce #'cons '(1 2 3) ())
```

станет *копия* редуцируемого списка — т. е. список `(1 2 3)`, физически составленный из новых точечных пар, указывающих на те же самые элементы `1`, `2` и `3`. На всякий случай попробуем свёртку «в другую сторону»: вычисление формы

```
(lreduce #'cons '(1 2 3) ())
```

даст результат `(((nil . 1) . 2) . 3)`. Порядком поднадоевшие арифметические примеры тоже прекрасно сработают: применение редукции к списку `(1 2 3 4)` и функции сложения (`#'+`) вернёт `10`, а функции умножения (`#'*`) — `24`, причём, естественно, как для левой, так и для правой редукции; по правде говоря, в Лиспе это делать нет никакого смысла, ведь обе функции умеют работать с произвольным количеством аргументов, достаточно задействовать `APPLY`, и всё получится без всякой свёртки.

Попробуем что-нибудь посложнее. С помощью безымянной функции, создающей точечную пару «наоборот» (второй аргумент в качестве `car`, первый — в роли `cdr`), и левой редукции можно «развернуть» список, точнее — создать новый список, состоящий из тех же элементов, но расставленных в обратном порядке: так, вычисление

```
(lreduce #'(lambda (a b) (cons b a)) '(1 2 3) ())
```

приведёт к результату `(3 2 1)`. Для полноты картины попробуем правую редукцию:

```
(rreduce #'(lambda (a b) (cons b a)) '(1 2 3) ())
```

вычислится в `(((nil . 3) . 2) . 1)`.

Имея функции для свёртки, мы можем поиск максимального элемента в списке реализовать так:

```
(defun max_in_list (ls)
    (lreduce #'(lambda (x y) (if (< x y) y x)) (cdr ls) (car ls))
)
```

Для следующего эксперимента нам потребуется функция, вставляющая новый элемент (число) в список, который уже отсортирован по возрастанию. Поскольку эта функция довольно «тяжёлая», опишем её обычным `defun`'ом, не пытаясь запихнуть в лямбда-список:

```
(defun ins2list (i list)
    (cond
        ((null list) (cons i ()))
        ((< i (car list)) (cons i list))
        (t (cons (car list) (ins2list i (cdr list))))
    )
)
```

Теперь мы можем — если, конечно, не будем сильно переживать по поводу эффективности — написать весьма компактную реализацию сортировки списка по возрастанию:

```
(defun redsort (lst) (rreduce #'ins2list lst ()))
```

Вспомним теперь, что левая редукция эффективнее правой. Перейти к левой редукции здесь очень просто, надо будет только переставить местами аргументы вызываемой функции:

```
(defun redsort2 (lst)
    (lreduce #'(lambda (ls el) (ins2list el ls)) lst ())
)
```

Попробуем теперь реализовать что-то вроде `mapcar`, но поскромнее — для функций одного аргумента:

```
(defun redmap (fun lst)
    (rreduce #'(lambda (x res) (cons (funcall fun x) res)) lst ())
)
```

Например, результатом `(redmap #'- '(1 2 3))` будет `(-1 -2 -3)`, а результатом

```
(redmap #'(lambda (x) (* 10 x)) '(1 2 3))
```

станет `(10 20 30)`.

В завершение нашей последовательности примеров напишем функцию, которая по исходному списку строит список из элементов, удовлетворяющих заданному условию (предикату):

```
(defun select (lst pred)
    (rreduce
        #'(lambda (x ls) (if (funcall pred x) (cons x ls) ls))
        lst ()
    )
)
```

Результатом вычисления `(select '(1 2 "abc" t 15) #'numberp)` станет список `(1 2 15)`, элементы `"abc"` и `T` будут проигнорированы, поскольку не являются числами (не удовлетворяют предикату `numberp`). Выражение `(select lst #'(lambda (x) (> x 0)))` выберет из списка чисел `lst` только строго положительные элементы и т. д.

11.1.12. Груз устаревших парадигм

В исторических версиях Лиспа не предусматривалось отдельного типа атомов для изображения строк — привычных нам литералов в двойных кавычках. В роли слов использовались символы, в роли фраз — списки символов, что-то вроде `(CATS EAT MICE)`. Та эпоха давно прошла; всё-таки идентификатор — это скорее инструмент программиста, а строковые константы относятся к *предметной области*, для которой написана программа — попросту говоря, их так или иначе видит пользователь. В эпоху, когда компьютеры были большими, а программы — маленькими, пользователь и программист практически всегда были едины в одном лице, то есть с машинами работали *только* программисты, конечных пользователей в современном понимании не существовало, так что идея *разделять* «пользовательское» и «программистское» никому не приходила в голову; но с тех пор, как говорится, много воды утекло.

Во всех версиях Лиспа, начиная с 1970-х и позже, строковые литералы как отдельная сущность, конечно же, присутствовали; но отголоски «бесстрочной» эпохи можно найти в Лиспе и сейчас. Например, на первый взгляд Лисп вроде бы не чувствителен к регистру букв в именах символов: `CAR`, `car`, `CaR` или `cAr` — это разная запись одного и того же имени, которое интерпретатор для себя обозначает буквами верхнего регистра — `CAR`; пользователей ранних диалектов Лиспа это вполне устраивало, к тому же многие старые компьютеры вообще не умели работать с буквами нижнего регистра. Когда же кому-то потребовалось (исходя из требований решаемой задачи — напри-

мер, при обработке каких-нибудь документов) использовать в строках буквы обоих регистров, разработчики очередного интерпретатора Лиспа не нашли ничего лучше, чем предусмотреть специальные средства для внедрения в имена символов букв нижнего регистра и знаков препинания, которые в норме в имени символа встречаться не должны. Это мракобесие пролезло в том числе и в Common Lisp: в нём имя символа можно записать между двумя вертикальными палочками, заставив содержать любую ерунду. Например, `|John|` — это символ с именем «`John`», содержащим три буквы нижнего регистра, и это совсем не то же самое, что `JOHN` (а без «палочек» получилось бы именно это). Сами «палочки» в имя не входят, они играют здесь примерно такую же роль, как кавычки в обычных строковых литералах. С их помощью в имя символа можно загнать и пробелы, и всевозможные скобки — вообще практически что угодно, вопрос тут лишь в том, *зачем* так делать; но ответ на этот вопрос станет очевиден, если представить себе, что у нас отобрали обычные строки, записываемые в кавычках, и оставили нам только символы.

Больше того, когда строковые константы всё-таки появились, вместе с ними в язык «проползли» средства преобразования символов в строки и обратно; в Common Lisp имя символа (в виде строки) можно получить с помощью функции `symbol-name`, а функция `intern` позволяет создать (во время исполнения программы!) символ, задав его строковое имя; точнее, новый символ создаётся, если символа с таким именем в системе пока нет, в противном случае возвращается существующий символ.

В современных условиях подобные выкрутасы выглядят, мягко говоря, странно. Идентификаторы — это личная кухня программиста, он может выбирать их по своему усмотрению и на свой вкус, тогда как строки в большинстве случаев обусловлены решаемой задачей; по идее, переименование идентификаторов вообще не должно никак влиять на видимое (пользователю) поведение программы. Работа с символами как со строками этому явно противоречит. Можно надеяться, что знание истории появления в Лиспе таких возможностей позволит читателю воздержаться от их использования.

Символы в роли строк — это, безусловно, не единственная устаревшая группа возможностей Common Lisp. Например, сейчас мало кто помнит про так называемые *списки свойств символов* и функцию `get`, позволяющую с ними работать; можно привести и другие примеры. «Символы как строки» здесь примечательны именно тем, что это *парадигма*, то есть подход к осмыслению возможностей инструмента, и при этом парадигма *устаревшая*, показавшая свою несостоятельность.

11.1.13. Возможности, которых лучше бы не было

Устаревшими парадигмами негативные особенности Common Lisp не исчерпываются. Мы уже упоминали в качестве основного недостатка этой спецификации её огромный объём: книга «Common Lisp the Language» содержит больше тысячи страниц; для подобных случаев в английской программистской лексике в последние годы начало активно использоваться слово *bloat*[16] и его производные, такие как *bloatware*. Популярный тезис, что добавлением новых возможностей в программу, язык или ещё куда-то якобы нельзя ничего испортить, постепенно уходит в историю — хотя, как мы знаем (в особенности на примере Си++), авторы стандартов предпочитают этого не замечать.

Бóльшую часть объёма спецификации Common Lisp занимает описание встроенных функций, т. е., если можно так выразиться, аналога стандартной библиотеки. С этим ещё можно было бы смириться: если представить себе *правильную* открытую реализацию, то она вполне могла бы (а если подумать — просто обязана) позволять выкинуть все ненужные/неиспользуемые функции, причём, что важно, вместе с обозначающими их объектами символов, которые в Лиспе тоже занимают память. Для компилируемого исполнения это означало бы, что в итоговый исполняемый файл попадает только код используемых функций и объекты нужных символов, а для интерпретируемого, видимо, потребовалась бы возможность быстрой пересборки интерпретатора или его исходно модульная структура (например, с выбором модулей параметрами командной строки); так или иначе, это можно устроить.

Намного хуже ситуация с такими возможностями, которые требуют поддержки со стороны базового интерпретатора — их выкинуть невозможно. Одним из самых одиозных примеров этого может служить предусмотренный в Common Lisp «общий вид» списка параметров функции — как обычной именованной, так и заданной лямбда-списком. Обычный список символов — имён формальных параметров, такой, какой использовался во всех наших примерах — показался создателям Common Lisp слишком примитивным, и они ввели для него возможность использования целого ряда ключевых слов: `&optional` для необязательных параметров, `&rest` для обозначения формального параметра, с которым связывается список «лишних» фактических параметров, `&key` для «ключевых» параметров (эти параметры при вызове функции снабжаются ещё одним видом ключевых слов — символами, имена которых начинаются с двоеточия), да вдобавок ещё и `&aux` — этим символом обозначаются формальные параметры, которые вообще не являются параметрами, их вводят в роли простых локальных переменных. В довершение список параметров можно снабдить ещё и декларациями типов, которые транслятор имеет право игнорировать (и все так и делают), но кто-то когда-то будто бы с их помощью повышал эффективность исполнения.

Одно только описание всего этого великолепия требует изрядных усилий, чтобы просто понять, чтó имеется в виду и как всем этим пользоваться. Естественно, в большинстве программ эти возможности остаются незадействованными, но ведь транслятор-то (если, конечно, для него декларируется соответствие спецификации Common Lisp) вынужден их поддерживать — а если учесть существование функции `eval`, то всю эту поддержку в любом случае придётся таскать за собой на случай, если подобное лямбда-выражение поступит `eval`'у на вход, т. е. даже в том случае, если программа компилируется и при этом в ней соответствующие ключевые слова не встречаются. Отметим, что усложняется не только анализ списков параметров (в спецформе `defun` и лямбда-выражениях), но и общий вид анализа формы, вызывающей функцию.

На фоне всех этих сложностей совершенно теряется один в действительности важный момент: коль скоро транслятор завязан на использование *ключевых* символов, все эти символы (как объекты, располагающиеся в памяти) должны присутствовать в среде выполнения. Этот эффект проявляется всякий раз, когда для управления работой той или иной подсистемы, спецформы или даже простой функции используются объекты символов в роли *значений*. Прекрасным примером этого может служить функция `coerce`, предназначенная для преобразования произвольных S-выражений в выражения другого типа. Она принимает два параметра: собственно исходное выражение и *тип*, к которому выражение следует привести, причём последний задаётся либо символом вроде `string`, `vector`, `float`, либо списком, соответствующим сложному типу, например `(vector integer)`. Всего здесь задействовано *более сорока* символов, обозначающих типы; сколь бы *правильной* ни была реализация, если вы хотя бы раз применили в программе функцию `coerce`, ваша среда исполнения будет неизбежно содержать их все.

Такой же громоздкостью грешит и рассматривавшаяся выше форма `setf`, и многие другие элементы спецификации Common Lisp. Так, аналогичные проблемы связаны со спецформой (точнее, формально говоря, *встроенным макросом*) `loop`, предназначенной для организации циклов. Для начала заметим, что этой форме в книге «Common Lisp the Language» посвящена *целая глава*, состоящая из двенадцати параграфов. Чтобы составить общее впечатление о том, как всё это выглядит, приведём пример, позаимствовав его напрямую из книги (§26.3; мы сохранили отступы, как в оригинале примера; никогда так не пишите!):

```
(loop for i from 1 to (compute-top-value)
      while (not (unacceptable i))
      collect (square i)
      do (format t "Working on ~D now" i)
      when (evenp i)
        do (format t "~D is a non-odd number" i)
      finally (format t "About to exit!"))
```

Все эти `for`, `from`, `to`, `while`, `collect`, `do`, `when`, `finally` — не что иное как символы, с помощью которых программист сообщает реализации `loop`, чего он от неё хочет добиться. Перечисленными символами дело не ограничивается; общее их количество довольно сложно сосчитать, поскольку они рассеяны по всему тексту главы, посвящённой форме `loop`, но можно уверенно заявить, что их там не меньше пятидесяти, и с каждым связаны те или иные возможности, предоставляемые реализацией `loop`. Всё это выглядит особенно прелестно на фоне того, что вообще-то циклы в Лиспе, как мы знаем, со-

[16] Буквальный перевод этого слова как глагола — «раздуваться», «пухнуть», «надувать(ся)»; есть и переносные значения — «толстеть» (о человеке), «коптить» (рыбу). Что особенно забавно, в качестве существительного слово *bloat* на английском жаргоне обозначает утопленника, а на американском — пьяницу.

вершенно не в почёте. Некоторые из этих символов (тот же `when`) обладают своей собственной ролью помимо той, в которой их использует `loop`, но на общую ситуацию это никак не влияет.

Пожалуй, теперь мы можем сформулировать главный, если угодно, системный недостаток Common Lisp и других подобных спецификаций, чаще всего связанных с интерпретируемыми или полуинтерпретируемыми языками. **Такие спецификации не позволяют — исходно не предусматривают возможности — сократить среду исполнения, даже если пользователю (автору программы) бóльшая её часть низачем не нужна**.

Как мы увидим позже при обсуждении функций ввода-вывода, Common Lisp сопротивляется не только гипотетическим попыткам сократить среду исполнения (т. е. урезать набор возможностей, включаемых в интерпретатор), но и робким поползновениям программиста решить, что некоторые части спецификации ему не нравятся, и ограничиться изучением того или иного её подмножества. Временами создаётся ощущение, что авторы Common Lisp задались целью попросту *заставить* каждого, кто этот инструмент возьмёт в руки, изучить все его возможности до единой.

11.1.14. Ввод-вывод

На обсуждение модели ввода-вывода, предлагаемой в спецификации Common Lisp, нам потребуется довольно много времени; обойти стороной эту тему мы никак не можем — в противном случае останется непонятно, как же на *этом* писать программы, а не отдельные и никому не нужные функции. К сожалению, мы вынуждены предупредить читателя, что, скорее всего, материал этого параграфа вообще отобьёт у него всякое желание иметь дело с Common Lisp; и тем не менее в контексте разговора о парадигмах этот материал полезен хотя бы как пример того, как делать не надо, и, главное, *почему* так делать не надо.

Изучая Си и Си++, мы могли привыкнуть (и наверняка привыкли) к определённому восприятию операций ввода-вывода. В основе модели, которую сейчас можно считать общепринятой, лежит понятие ***потока*** ввода или вывода (а в некоторых случаях — того и другого сразу), при этом потоки ввода имеют фундаментальное свойство *заканчиваться*, т. е. сигнализировать о наступлении ситуации «конец файла»; кроме того, любая операция ввода-вывода (в особенности операция создания нового потока, в том числе путём открытия файла на чтение или запись) может стать причиной ошибки, и эта ошибка с точки зрения программы совершенно не обязана быть фатальной, то есть у работающей программы должна быть возможность о происшедших ошибках узнавать и обрабатывать их так, как хочет её автор, а не кто-то другой (например, создатель транслятора).

Эта модель при всей её привычности на самом деле нетривиальна и даже может, пожалуй, рассматриваться как *парадигма*; и, конечно же, она сложилась не сразу и не вдруг, и тем более отнюдь не сразу стала общепринятой. Язык Лисп сам по себе много старше, он появился, ещё когда не существовало понятия файловой системы — да что там, в 1950-е годы не было операционных систем как явления. В давние времена сложился и набор основных встроенных функций, и этот набор исходно никак не учитывал, например, феномен ситуации конца файла. Делить ошибки, как это обычно происходит сейчас, на *программистские* (на которые может и должен среагировать транслятор) и *пользовательские* (на которые реагировать должна выполняющаяся программа) в то время тоже никому не приходило в голову по причинам, которые мы уже обсуждали в §11.1.12.

Спецификация Common Lisp создавалась позднее, к этому времени общий подход к вводу-выводу успел «устаканиться», но то, как именно были изменены (дополнены) описания встроенных функций для соответствия новым реалиям, вызывает некое недоумение. Так, если мы хотим обрабатывать ситуацию «конец файла» — а мы, естественно, этого хотим, как вообще может быть иначе, — то нам придётся каждой функции ввода передавать *три* дополнительных аргумента: один из них — логическое значение `nil` (ложь) — указывает, что функция ввода не должна «конец файла» обрабатывать сама (как? очень просто: если им этого не запретить, они «конец файла» приравнивают к ошибкам, при этом диагностику выдают сами, а в довершение запускают отладчик), другой задаёт то значение, которое нужно вернуть из функции, если «конец файла» всё-таки случился, а третий (или скорее *первый*, если иметь в виду его расположение) — это идентификатор потока; его можно было бы опустить, когда ввод идёт из стандартного потока, да только он, увы, должен стоять *перед* теми двумя, так что теперь его опустить не получится.

Ситуация станет ещё забавнее, как только нам придёт в голову *открыть файл*. В «Common Lisp the Language» есть глава, посвящённая потокам (она имеет номер 21 и так и называется «Streams»), а в ней параграф, озаглавленный «Creating New Streams»; функции с именем `open` в этом параграфе нет, так что может сложиться впечатление, что её нет вообще. На самом деле она всё-таки есть, только надо как следует поискать — и тогда обнаружится, что самый привычный и очевидный способ создания новых потоков ввода-вывода в этой книге отнесён не к области «создания новых потоков», а к области «интерфейса к файловой системе». Надо сказать, что это тоже отдельная глава, описывающая, как выглядит файловая система (имена файлов, их свойства и т. п.) в той странной альтернативной реальности, которую зачем-то придумали создатели Common Lisp. Отыскав нужную функцию, мы рискуем не понять описание параметров, которые она принимает:

```
open filename &key :direction :element-type :if-exists
                              :if-does-not-exist :external-format
```

Со времён появления Лиспа как такового до выхода спецификации Common Lisp у публики более-менее оформилось понимание, как надо работать с вводом-выводом, однако возникла другая проблема, которую создатели Common Lisp, разумеется, не считали проблемой: в спецификацию просочились всевозможные прелести из тех, что мы обсуждали в предыдущем параграфе, в том числе *ключевые параметры* функций. Появились и в принципе полезные возможности вроде обработки исключений — но полезные, пожалуй, лишь до тех пор, пока нам их не навязывают.

До определённого момента можно просто игнорировать все эти выверты комитетского мышления; но вот нам потребовалось открыть файл, и в описании функции, которая это делает, мы видим пресловутые ключевые параметры, причём всё устроено так, что без их применения мы вообще не сможем работать — например, открыть файл на чтение без них ещё можно, а вот на запись уже не получится. С обработкой ошибок тоже весёлая ситуация: или мы покорно идём осваивать встроенные в Common Lisp исключения (в документации они называются *conditions*, т. е. «условия»; час от часу не легче) и в довесок к ним ещё объектно-ориентированную подсистему, которая называется CLOS (*Common Lisp Object System*) — или нам придётся смириться с тем, что, к примеру, программа, которой не хватило полномочий для открытия файла, аварийно завершается с невнятной диагностикой, а то и вовсе запускает отладчик.

Судя по всему, авторы Common Lisp так и не преодолели наследие той (неоднократно уже упоминавшейся) эпохи, когда не было конечных пользователей, а потому никому не приходило в голову разделять «программистское» и «пользовательское». Такой вывод уже напрашивается, когда мы видим, сколь трудно взять в свои руки заботу о выдаваемой диагностике, не довольствуясь (почему бы это) сообщениями интерпретатора. Но есть и другие признаки. Так, в тексте «Common Lisp the Language» бóльшую часть главы о вводе-выводе занимает описание функции `read`, позволяющей читать произвольные S-выражения, представленные в виде текста. Да, при этом задействуются лексический и синтаксический анализаторы, имеющиеся в интерпретаторе, то есть вообще-то инструменты сугубо программистские; да, при этом диагностика выдаётся тоже исключительно программистская, обычный пользователь всего этого просто не поймёт; при этом сами анализаторы можно очень гибко перенастраивать, и для поддержки этой настройки спецификация предусматривает целый ворох сложных инструментов. Между тем *всё это принципиально не годится для программ, передаваемых конечному пользователю*, поскольку вроде бы очевидно, что конечный пользователь не обязан знать Лисп и уметь, например, вводить информацию с клавиатуры в виде S-выражений, соблюдая их синтаксис. Иначе говоря, интерпретатор Common Lisp содержит в себе (а книга — долго и упорно расписывает) прорву таких возможностей, которые в силу фундаментальных причин вообще нельзя применять, если у программы предполагаются какие-то ещё пользователи, кроме её автора.

Постараемся для начала объяснить, как организовать в Common Lisp обыкновенный и привычный файлово-потоковый ввод-вывод, и при этом чтобы ситуация «конец файла» рассматривалась как штатная, а не как ошибоч-

ная. К более тяжёлым средствам обработки ошибок мы вернёмся позже.

Начнём с замечания о стандартных потоках ввода-вывода. Для их обозначения Common Lisp предусматривает глобальные переменные `*standard-input*`, `*standard-output*` и `*error-output*`, которые *по идее* должны соответствовать привычным нам потокам ввода, вывода и диагностики. Как водится, это только начало истории. Кроме этих трёх переменных, спецификация предусматривает ещё `*trace-output*`, `*query-io*`, `*debug-io*` и `*terminal-io*`, причём последние три должны допускать как ввод, так и вывод. Согласно спецификации, `*query-io*` предназначается для диалога с пользователем (якобы даже в случае, если стандартные потоки ввода и вывода перенаправлены), `*terminal-io*` должен всегда позволять работу с терминалом (в обе стороны), `*trace-output*` и `*debug-output*` тоже имеют свои особые роли.

Интересно, что вообще-то реализовать все эти особенности поведения *возможно*. Интерпретатор мог бы, например, открывать устройство `/dev/tty`, чтобы добраться до управляющего терминала, если стандартные потоки перенаправлены; в некоторых случаях можно было бы перепрограммировать драйвер терминала и всё такое прочее. В реальности этого никто не делает. Если написать простенький скрипт, который будет печатать одну фразу в `*standard-output*`, другую — в `*terminal-io*`, третью — в `*query-io*`, а потом перенаправить выдачу такого скрипта в файл, то все три фразы окажутся в этом файле, и это верно для всех трёх рассматриваемых нами интерпретаторов; прямо скажем, заморачиваться их авторы не стали. Дальше всех пошёл `gcl`: при работе с потоком `*error-output*` он физически производит вывод всё в тот же `stdout` (дескриптор `1`), тогда как `sbcl` и `ecl` хотя бы знают о существовании дескриптора `2` (`stderr`).

В большинстве случаев поток можно не указывать либо указать вместо потока `nil` или `t`. При этом (если буквально следовать спецификации) отсутствие явного указания потока либо указание `nil` означает для функций ввода работу с `*standard-input*`, для функций вывода — работу с `*standard-output*`; указание `t` в обоих случаях означает использование `*terminal-io*`. Это, конечно, *очень* важно, если учесть, что во всех трёх рассматриваемых нами (и, возможно, вообще во всех существующих) интерпретаторах между этими потоками нет никакой разницы. Так или иначе, если поток предполагается стандартный, но его почему-то обязательно надо указать, обычно используют `t` — видимо, потому что так короче.

Для *вывода* информации в классических диалектах Лиспа предусматривались четыре похожие друг на друга функции: `print`, `prin1`, `princ` и `pprint`. Как часто бывает, от функции с наиболее понятным именем — `print` — как раз меньше всего толку, но об этом чуть позже. Все четыре функции принимают один или два аргумента: обязательный — что́ нужно напечатать, и опциональный — поток, в который это надо печатать; как можно догадаться, в классических диалектах этого опционального параметра не было, он появился позже. Например, `(prin1 x)` выдаст текущее значение переменной `x` в стандартный поток вывода — `*standard-output*`; `(prin1 x t)` выдаст значение уже в поток `*terminal-io*`, но, как мы знаем, физически при этом произойдёт то же самое; а вот выражение `(prin1 x *error-output*)` должно печатать уже в диагностический поток, и в двух из трёх наших интерпретаторов это действительно произойдёт; стыд и позор авторам `gcl`, где этот поток не отличается от стандартного вывода.

Различаются эти функции тем, что́ конкретно они выдают, получив параметром то или иное значение. Все, кроме `princ`, печатают *текстовое представление полученного S-выражения, пригодное для «чтения обратно»*, то есть соответствующее синтаксическим правилам Лиспа; например, если попытаться напечатать число, то число и будет напечатано, а вот если попробовать напечатать *строку*, то на печать будет выведено представление строки в Лиспе, *включая кавычки*. Если попытаться напечатать какой-нибудь особый символ, в потоке мы увидим его лисповское обозначение, например `(prin1 (code-char 65))` напечатает «`#\A`» (65 — это ASCII-код заглавной латинской буквы `A`), а `(prin1 (code-char 10))` напечатает громоздкое `#\Newline`.

Функция `prin1` ограничится только печатью представления заданного выражения; функция `pprint` сначала выведет перевод строки, и только после него — представление выражения; `print` сделает то же самое, только ещё добавит в конце пробел. Здесь может возникнуть вполне резонный вопрос, зачем для этого было придумывать отдельную функцию; дело в том, что, разумеется, функция `print` появилась первой, остальные были придуманы позже. В те времена, когда их придумывали, принцип «делай что-то одно» ещё не стал общепризнанным.

Функция `princ` отличается от остальных тем, что печатает строки без кавычек, а символы (которые *chars*) выдаёт, если можно так выразиться, в виде символов, а не в виде их лисповских имён. Например, `(princ #\Newline)` выдаст символ перевода строки, то есть переведёт строку на печати; `(princ "Hello world")` напечатает `Hello world` — без всяких кавычек. Очевидно, что эта функция из всех четырёх — самая полезная; по большому счёту, в программе, предназначенной для конечных пользователей, остальные три функции вообще было бы достаточно странно встретить.

Перевод строки можно выдать ещё с помощью `terpri`, которая либо вызывается без аргументов, либо с одним аргументом, задающим поток вывода.

Апологеты Common Lisp часто рекомендуют для вывода использовать функцию `format`, по своему принципу действия напоминающую `fprintf` из стандартной библиотеки Си — она тоже принимает аргументами идентификатор потока, ***форматную строку*** (только в ней используются не «процентики», а символ `~`) и далее произвольное число аргументов, предназначенных к выдаче; рассказывать про неё подробно мы не будем, сэкономим место. Отметим только одно её любопытное свойство: если в качестве потока указать `nil`, она превращается в аналог `sprintf`, то есть формирует *строку*, которую иначе выдала бы на печать, и *возвращает* эту строку как своё значение.

С вводом ситуация несколько сложнее. Выше мы уже упоминали функцию `read`, которая позволяет задействовать лексический и синтаксический анализаторы, используемые интерпретатором для разбора текста программы. Возьмём на себя смелость посоветовать читателю забыть про саму возможность использования этой функции. Иногда возникает соблазн задействовать её, чтобы, например, прочитать число с клавиатуры; не делайте этого! А если сомневаетесь, напишите коротенькую программу, которая, например, запрашивает число и печатает указанное количество звёздочек, или делает ещё что-нибудь столь же примитивное, запустите её и, представив себя на месте пользователя, введите «по ошибке» открывающую круглую скобку — и после этого попытайтесь выбраться оттуда, куда попали. Рискуя испортить вам сюрприз, отметим, что интерпретатор будет при этом ждать, чтобы ему ввели парную закрывающую скобку, чтобы сформировать корректный список, а всевозможные привычные нам `Ctrl-C` и `Ctrl-D` в большинстве случаев отправят вас в режим отладки. Учтите, пользователь в норме не имеет никакого понятия ни о Лиспе, ни о том, что ваша программа читает какое-то там «S-выражение», в котором, видите ли, баланс скобок должен соблюдаться: его просили ввести число, он ошибся и ввёл скобку, но при чём тут теперь какой-то там баланс?! Ну а когда вместо ожидаемого завершения пользователь, загнанный в состояние «выпустите меня отсюда», увидит приглашение отладчика, которому предшествует простыня непонятных сообщений, можете быть уверены, что автора такой программы он помянет отнюдь не добрым словом.

Вообще говоря, подсистема, называемая обычно *lisp reader* и заключённая в функции `read`, в некоторых случаях даже может оказаться полезной; например, в статье [16] приведён целый ряд примеров, как приспособить её для чтения файлов в разнообразных форматах, включая, например, JSON. Здесь имеется одно серьёзное ограничение. Пока речь идёт об анализе файлов, в том или ином смысле считающихся *частью вашей программы* — всё в порядке, с диагностикой интерпретатора вы сами как-нибудь справитесь, а конечному пользователю предоставите готовые «вылизанные» файлы, на которые reader ругаться уже не будет. Но если файл, который вы собрались анализировать, по смыслу задачи вам должен предоставить пользователь или кто-то ещё, но не вы сами, то гарантировать его корректность заведомо невозможно — и, стало быть, пользователю потенциально придётся иметь дело с диагностикой, выдаваемой интерпретатором. Допускать такое, попросту говоря, *неприлично*.

Коль скоро мы запретили себе использовать lisp reader, всё, что нам остаётся — это старый добрый посимвольный ввод, для которого предусмотрена функция `read-char`. Эта функция имеет четыре необязательных аргумента, из которых нам обязательно (пардон за невольный каламбур) потребуются первые три: поток ввода, указание не считать конец файла ошибкой и значение, которое следует возвратить вместо очередного символа, когда ситуация конца файла настанет; как мы уже отмечали на стр. 128, при чтении из стандартного потока ввода мы могли бы не указывать соответствующий аргумент, но беда в том, что он стоит первым, а два последующих мы указать просто-таки обязаны — мы же не хотим, чтобы при обнаружении конца файла интерпретатор оглушил нашего пользователя кучей диагностики, которую

пользователь заведомо не поймёт, тем более что никакая это не ошибка. Итак, для чтения символа с клавиатуры придётся воспользоваться вызовом вроде `(read-char t nil 'eof)` — в этом случае для индикации конца файла будет возвращён символ (на этот раз *symbol*, не *character*) `eof`. Можно, впрочем, поступить проще: `(read-char t nil nil)`, и пусть в ответ на конец файла возвращается привычное для всяких особых случаев `nil`.

Например, хорошо знакомая нам по Паскалю и Си задача «читать текст из стандартного потока ввода, пока он не кончится, и в ответ на каждую строку печатать её длину» на Лиспе решается так:

```
#!/usr/bin/sbcl --script
; textbychar.lsp
(defun main (len)
    (let ((c (read-char t nil nil)))
        (cond
            ((eq c nil) t)
            ((eq c #\Newline) (prin1 len) (terpri) (main 0))
            (t (main (+ 1 len)))
        )
    )
)
(main 0)
```

Работать эта программа будет с любым из наших интерпретаторов, достаточно заменить соответствующим образом первую строчку (см. §11.1.2).

Если соберётесь этот пример попробовать, заодно можете полюбоваться на эффект от нажатия Ctrl-C; для тех, кто не хочет пробовать, скажем, что все три наших интерпретатора считают своим долгом рассматривать это как настоящую катастрофу — sbcl вываливает простыню диагностики, но хотя бы всё-таки завершается, тогда как ecl и gcl переходят в режим отладки, как будто их цель — довести пользователя до инфаркта. Чтобы справиться с этим маразмом, потребуется обработка исключений; мы вернёмся к этой проблеме ближе к концу параграфа.

Во многих случаях может показаться привлекательной также функция `read-line`, которая, подобно сишной `fgets`, читает из заданного потока строку до символа конца строки. Параметры у неё точно такие же, как и у `read-char`. Проблемы с этой функцией начинаются, когда вы хотите отличить случай полностью прочитанной строки (чтение завершено из-за получения символа перевода строки) от не совсем корректного случая, когда строка была прервана наступлением ситуации «конец файла». Понять, что произошло, возможно, но для этого потребуется освоить *многозначные* (то есть возвращающие больше одного значения) функции. Это, в принципе, не очень сложно; если у читателя возникнет такое желание, порекомендуем ему обратиться к документации и прочитать описания спецформ `values` и `multiple-value-list`, этого будет достаточно.

К этому времени у читателя наверняка созрел вопрос, как же всё-таки создать новый поток. Из материала второго тома мы знаем, что в ОС Unix имеется довольно богатый ассортимент потоков: файлы, именованные и неименованные каналы, сокеты... увы, создатели Common Lisp до всего этого не снизошли, так что, перефразируя известную юмористическую песенку Высоцкого, кроме файлов — никаких чудес; ни каналы, ни сокеты спецификация даже не упоминает, хорошо хоть есть возможность открыть файл. Функцию `open` мы уже начали обсуждать на стр. 128, напомним её официальное описание:

```
open filename &key :direction :element-type :if-exists
                              :if-does-not-exist :external-format
```

Как можно ожидать, эта функция открывает файл и возвращает созданный поток ввода или вывода. В простейшем случае — если не использовать ключевые параметры — `open` откроет на чтение файл, имя которого указано единственным аргументом. При этом если хоть что-нибудь пойдёт не так (а читатель прекрасно знает, сколькими способами что-то может пойти не так при попытке открыть файл), интерпретатор в своей обычной манере ударится в панику.

Отметим ещё один странный момент. Имя файла может быть задано строкой, объектом специального типа *pathname* (для работы с ними имеется особый набор функций) или даже другим (существующим) потоком. Как часто говорят американцы, вы не хотите знать об этом больше.

Чтобы добиться от `open` хоть сколько-нибудь осмысленного поведения, придётся всё-таки разобраться, как использовать ключевые параметры при вызове функций. Странные идентификаторы, начинающиеся с двоеточия, соответствуют каждый своему параметру, у каждого такого параметра есть значение по умолчанию на случай, если при вызове его не зададут, а чтобы его задать, нужно в форме вызова написать сначала имя ключевого параметра, а затем нужное значение для него. Например, если какая-нибудь функция `f` принимает ключевой параметр `:weirdnum`, то задать ему значение можно так: `(f :weirdnum 13)`.

Происходящее ещё больше запутывается тем, что в роли *значений* для ключевых параметров функция `open` использует всё те же ключевые слова, т. е. идентификаторы, начинающиеся с двоеточия. Вообще-то это вполне нормально, если учесть, что в действительности любой такой идентификатор — это просто константа, равная самой себе и больше ничему; к сожалению, одни функции в Common Lisp используют в этой роли ключевые слова, другие (вроде той же `coerce`) — обычные символы, и целостности восприятия происходящего всё это отнюдь не способствует.

Вернёмся к функции `open`. Чтобы открыть некий файл на запись, можно попробовать сделать что-то вроде следующего:

```
(setq f (open "file.txt" :direction :output))
```

Здесь *значением* для параметра `:direction` указана константа `:output`. По умолчанию, как можно догадаться, используется `:input`, а ещё предусмотрены `:io` — открыть для работы в обоих направлениях (кто б ещё объяснил, как это можно применить в той альтернативной реальности, которую придумали создатели спецификации) и `:probe` — вообще не открывать файл, просто узнать, существует ли он.

Здесь невозможно удержаться от ещё одного замечания. В тексте спецификации явным образом сказано, что при этом файл открывается и сразу же закрывается. По-видимому, предполагалось использовать это примерно так: посмотреть, существует ли файл, и если да, то тогда уж его открывать, чтобы избежать ошибки. Проблема в том, что подобная последовательность действий порождает прямо-таки канонический race condition (напомним, что официальный русский термин — ***ситуация гонок (состязаний)***; см. т. 2, §§5.3.4, 7.1.1): никто не может гарантировать, что файл не будет удалён кем-нибудь другим за время, которое пройдёт между вызовом open в варианте с :probe и вызовом, открывающим файл «по-настоящему».

Если всё пройдёт хорошо и файл откроется, то с переменной `f` будет связан объект потока вывода, который можно будет передавать во всякие функции вроде `princ`, `terpri`, `format` и т. д. Но, как уже говорилось, если хоть что-то пойдёт не так, интерпретатор окажется в состоянии истерики, а пользователь — в предынфарктном.

Частично снять остроту творящегося маразма помогут ещё два ключевых параметра — `:if-does-not-exist` и `:if-exists`. Начнём с `:if-exists`, который имеет смысл только при открытии файла на запись, а во всех остальных случаях игнорируется. Его значение по умолчанию — `:error`, оно подразумевает как раз такое поведение, которое допускать не следует (выдать собственную диагностику интерпретатора, вывалиться в режим отладки и всё такое прочее). Среди прочих возможных значений, каковых довольно много, можно порекомендовать `:overwrite` (файл перезаписывается с нуля), `:append` (запись будет производиться в конец существующего файла) и `nil` (файл не откроется, `open` вернёт значение `nil`). Пример, приведённый выше, следует с учётом этого скорректировать, скажем, вот так:

```
(setq f
    (open "file.txt" :direction :output :if-exists :overwrite)
)
```

Параметр `:if-does-not-exist`, как легко догадаться, задаёт поведение на случай, если файла с заданным именем нет. Здесь предусмотрено всего три возможных значения: `:error` (см. выше), `:create` (создать файл) и `nil` (ничего не открывать, вернуть `nil`). К счастью, при открытии файла на запись умолчанием выступает `:create`, а не `:error`, так что этот параметр всерьёз нужен только при открытии на чтение; зато уж в этом случае он становится просто-таки необходимым, и единственное осмысленное значение для него — именно `nil`, так что открывать файлы на чтение надо примерно так:

```
(setq f (open "file.txt" :if-does-not-exist nil))
```

На всякий случай подчеркнём ещё раз, что все более «сложные» ошибки вроде нехватки полномочий, отсутствия нужной директории и т. п. без вариантов вызывают ошибку с точки зрения интерпретатора, а чтобы её перехватить, потребуется освоить ещё и механизм обработки исключений.

К счастью, с *закрытием* потоков всё несколько проще: для этого есть функция `close`, принимающая один аргумент — собственно поток — и закрывающая его. Впрочем, Common Lisp и здесь не может обойтись без сюрприза — у `close` есть дополнительный ключевой параметр, который называется `:abort`. Мы не станем тратить на него время.

Отметим ещё один интересный казус. Теоретически Common Lisp поддерживает не только текстовые потоки, но и бинарные; чтобы открыть файл как бинарный, нужно добавить в вызов `open` ещё один ключевой параметр — `:element-type` со значением `signed-byte` или `unsigned-byte`. Да, в этот раз значения предполагаются без двоеточия, так что это обычные символы, и надо не забыть квотировать их апострофом, чтобы не дать им вычислиться, примерно так:

```
(setq f
    (open "file.txt"
        :direction :output
        :if-does-not-exist nil
        :element-type 'unsigned-byte
    )
)
```

Набор операций для работы с бинарными потоками, прямо скажем, не впечатляет своим разнообразием: предусмотрены всего две операции — чтение байта и запись байта. Запись производится функцией `write-byte`, которая принимает два параметра: целое число, соответствующее значению байта, который надо записать, и поток, куда этот байт записывать. Функция для чтения байта называется `read-byte`, параметры у неё в целом те же, что и у `read-char` — поток, затем `nil`, чтобы не считать «конец файла» ошибкой, и значение, которое нужно вернуть, если «конец файла» всё же настал. Параметр, задающий поток, тут обязателен (в отличие от `read-char`, где его вроде бы можно опустить), но мы в любом случае договорились указывать его всегда.

Попробуем теперь понять, как же всё-таки обработать возможные ошибки самостоятельно, не допуская к этому назойливый интерпретатор. Для этого, как уже говорилось, потребуется хотя бы минимальное представление о том, как в Common Lisp устроена обработка исключений.

Создатели Common Lisp называют исключения словом *conditions* (то есть «условие», или, если угодно, «обстоятельство»). Сколько-нибудь подробный рассказ об этой подсистеме имел бы такой объём, что к концу мы бы забыли, зачем вообще начали всю эту канитель; поэтому мы опустим обсуждение того, как такие исключения создавать и возбуждать, как их анализировать и т. п., и тем более не будем рассказывать, как в интерактивном режиме что-то исправить и вернуть управление в точку, где исключение было выброшено (да, Common Lisp и такое позволяет); ограничимся лишь рассмотрением базового средства для перехвата исключений — спецформой `handler-case`.

Первым аргументом для `handler-case` служит выражение, которое нужно вычислить; в принципе здесь может стоять что угодно, но обычно это всё же список — обращение к функции или спецформе, и достаточно часто здесь используется форма `progn`, позволяющая последовательно вычислить свои аргументы — аналог составного оператора императивных языков. Далее в форме `handler-case` можно указать произвольное количество *обработчиков* для разных видов исключительных ситуаций. Всё это вместе идеологически несколько напоминает знакомую нам по Си++ конструкцию `try/catch`, вплоть до того, что в каждом обработчике указывается *имя переменной*, значением которой на время выполнения обработчика станет *объект исключения*. Всё вместе выглядит примерно так:

```
(handler-case
    (my-suspicious-func arg1 arg2 ...)
    ;;
    (file-error (er) (form1 ...) (form2 ...))
)
```

Здесь мы *пытаемся* вызвать функцию `my-syspicious-func`, передав ей какие-то аргументы; если произойдёт исключительная ситуация типа `file-error`, то объект исключительной ситуации будет связан с переменной `er`, после чего будут вычислены формы `(form1 ...)`, `(form2 ...)` и так далее. Таких обработчиков в одном `handler-case` может быть несколько для разных типов исключений.

В роли объекта исключения выступает атомарное S-выражение специального типа; об этих выражениях тоже придётся сказать несколько слов. Спецификация Common Lisp подразумевает объектно-ориентированный подход к классификации пресловутых *conditions*, подобный тому, что мы рассматривали в §10.6.15 для Си++. Сам по себе этот подход, позволяющий построить *иерархию* особых ситуаций, ничем не плох; но, что бы ни говорили апологеты Common Lisp, объектно-ориентированная парадигма для Лиспа чужеродна. В Common Lisp объектно-ориентированное программирование представлено подсистемой CLOS (*Common Lisp object system*), которая сама по себе достаточно сложна. С наличием этой подсистемы можно было бы смириться, если бы программисту хотя бы оставили возможность её игнорировать, коль скоро у него не возникает желания заниматься ООП на Лиспе; увы, комитет X3J13 эту возможность ликвидировал, приняв (явным образом!) решение намертво связать между собой CLOS и систему обработки особых ситуаций.

CLOS мы обсуждать всё-таки не станем, это слишком долго, тем более в плане обработки ошибок ввода-вывода это нам практически ничего не даст; всё, чего мы можем добиться от объекта исключения, имеющего тип `file-error` (а при невозможности открыть файл исключение будет именно такое) — это извлечь из него с помощью функции `file-error-pathname` имя файла, ставшего причиной ошибки — как будто мы сами не знаем, какой файл открывали. Логично было бы ожидать, что в объекте найдётся какая-то ещё информация, в том числе самое для нас интересное — что, собственно говоря, произошло; вот только спецификация Common Lisp, в иных местах подробная до занудства, этот момент обошла стороной, так что каждая реализация делает всё по-своему, и не сказать чтобы удачно — а если говорить прямо, то просто уродливо. Начнём с SBCL, который включает в объект типа `file-error` информацию, как надо (по мнению SBCL) печатать сообщение о происшедшей ошибке; сюда входит, помимо прочего, текстовое описание самой ошибки (например, `"Permission denied"`). Извлечь его можно так (здесь `ex` — переменная, значением которой является пойманный объект исключения):

```
(third (slot-value ex 'sb-kernel::format-arguments))
```

Попробовать это можно, например, попытавшись открыть на чтение файл `/etc/shadow` (никому, кроме пользователя `root`, это сделать не удастся, получится как раз `"Permission denied"`):

```
* (defvar *ex*)
*EX*
* (handler-case (open "/etc/shadow")
          (file-error (er) (setq *ex* er)))
#<SB-INT:SIMPLE-FILE-ERROR "~@<~?: ~2I~_~A~:>" {1003A44113}>
* (third (slot-value *ex* 'sb-kernel::format-arguments))
"Permission denied"
*
```

Надо сказать, что заклинание `sb-kernel::format-arguments` (имя слота объекта исключения) нигде не документировано, как и то, что значение в этом слоте представляет собой список, а нужное нам описание ошибки находится в его третьем элементе. Чтобы сие раскопать, у автора книги ушло около часа; в других интерпретаторах всё это, естественно, работать не будет, и более того, никто не гарантирует, что найденное заклинание сработает в другой версии того же SBCL. При этом **сам текст описания изменится, если в системе установлен другой язык**, что окончательно делает невозможным программный анализ причин ошибки по строке, которую мы с таким трудом извлекли; ну а никакой другой информации — например, кода ошибки — в объекте просто нет.

Впрочем, случай SBCL ещё не самый худший. В ECL объект исключения для `file-error` не содержит даже этого, ну а GCL вообще не знает формы с именем `handler-case` — точнее, механизм автодополнения о её существовании знает, но попытка к ней обратиться вызывает ошибку. Эксперименты с GCL можно на этом закончить, окончательно списав этот интерпретатор в категорию лабораторных игрушек, непригодных для практической работы.

Вообще-то *выдать пользователю* сообщение, содержащее причины возникшей ошибки, можно (согласно спецификации) вполне документированным способом, который по идее (которая, к сожалению, весьма далека от реальности) должен работать во всех интерпретаторах Common Lisp: просто-напросто *напечатать* объект исключения, воспользовавшись для этого функцией `princ` (или `format` с директивой `~A`). Так, SBCL напечатает что-то вроде

```
error opening #P"/etc/shadow": Permission denied
```

— а при включённой русской локализации мы получим следующее:

```
error opening #P"/etc/shadow": Отказано в доступе
```

Кто б сомневался, что переведено окажется не всё; локали вообще средство весьма сомнительное. Впрочем, ECL не может и этого — в аналогичной ситуации он печатает следующее:

```
> (princ *ex*)
Filesystem error with pathname "/etc/shadow".
Either
 1) the file does not exist, or
 2) we are not allowed to access the file, or
 3) the pathname points to a broken symbolic link.
#<a FILE-ERROR>
```

Как видим, здесь вообще нет никакой информации о том, что же за ошибка случилась. Но и с тем, что выдаёт SBCL, имеется по меньшей мере две проблемы. Во-первых, программа самостоятельно не может предпринять никаких действий по исправлению ошибки. Представьте себе, например, что при том же самом отказе в доступе мы захотели бы попытаться изменить права доступа к файлу и потом снова его открыть — если файл наш, это вполне получится; при попытке создать файл в несуществующей директории мы могли бы попытаться создать эту директорию и т. д.; но Common Lisp нам ничего подобного не позволит, начисто закрыв программе доступ к нужной для этого информации. Во-вторых, *автор программы не может сам решить, как должно выглядеть сообщение об ошибке* — его конкретный вид навязан интерпретатором.

Попробуем всё же написать программу, работающую с файлами, приблизившись, насколько возможно, к нашему пониманию корректной работы. Пример мы для этого выберем совсем простенький — решим задачу, которую раньше решали для потока стандартного вывода: читать из заданного файла текст, выдавая в поток стандартного вывода длины прочитанных строк, и так пока файл не кончится.

Из рассматривавшихся нами трёх интерпретаторов только один — SBCL — позволяет здесь сделать хоть что-то не совсем позорное, им мы и ограничимся; программу оформим, как водится, в виде скрипта (см. §11.1.2), начав её со следующих строк:

```
#!/usr/bin/sbcl --script
; strlens_file.lsp
```

Код программы начнём с функции `do_it`, которая будет отличаться от функции `main` примера, разобранного на стр. 130, наличием второго параметра — потока, из которого следует читать:

```
(defun do_it (file len)
    (let ((c (read-char file nil nil)))
        (cond
            ((eq c nil) t)
            ((eql c #\Newline) (prin1 len) (terpri) (do_it file 0))
            (t (do_it file (+ 1 len)))
        )
    )
)
```

Наша программа будет получать имя файла через параметр командной строки, если же параметра нет — использовать поток стандартного ввода. Для этого напишем две вспомогательные функции, которые будут вызывать функцию `do_it`, предварительно всё для этого подготовив. Для случая использования стандартного ввода готовить особо нечего, так что функция получается тривиальная:

```
(defun process_stdin ()
    (do_it *standard-input* 0)
)
```

Со случаем, когда нужно открывать файл, всё несколько сложнее. Как уже упоминалось выше, создатели Common Lisp настоятельно советуют не использовать `open` и `close` напрямую. Связано это всё с тем же подходом к обработке ошибок ввода-вывода (исключение на любой чих); попробуем пояснить, что имеется в виду.

Допустим, в вашей программе организована корректная обработка ошибок ввода-вывода, основанная на исключениях. Если теперь вы в какой-нибудь из своих функций успешно откроете файл, но за этим вскоре последует какая-нибудь исключительная ситуация — хотя бы даже просто какой-то другой файл не откроется — то из функции ваша программа, очевидно, вылетит по исключению, *так и не закрыв тот файл, который открылся успешно*.

Бороться с этим предлагается с помощью конструкции `with-open-file`; здесь вы сразу же указываете, какой файл открыть и в каком режиме (собственно говоря, все параметры как для `open`), а также с какой переменной связать созданный поток и какие затем формы следует вычислить (естественно, имя вашей файловой переменной может в них использоваться). Файл автоматически закрывается по завершении вычисления всех форм, а равно и при досрочном прекращении их вычисления из-за возникшего исключения; это ровно то, ради чего предлагается отказаться от `open` в пользу `with-open-file`. Конечно, это имеет смысл лишь в том случае, если ваша программа корректно обрабатывает исключительные ситуации.

Попробуем применить этот подход в нашей программе. Функция, которая открывает файл по заданному имени и затем для полученного потока вызывает `do_it`, получится такая:

```
(defun process_file (fname)
    (with-open-file
        (f fname :direction :input :if-does-not-exist nil)
        (if (null f)
            (progn
                (princ "Couldn't open the file" *error-output*)
                (terpri *error-output*)
                nil
            )
            (do_it f 0)
        )
    )
)
```

Теперь нам остаётся лишь написать главную программу — форму, вычисление которой, собственно говоря, и сделает то, ради чего написана программа. Напомним, что в SBCL параметры командной строки доступны (в виде списка строк) через глобальную переменную `*posix-argv*`. Если этот список состоит из одного элемента — значит, имени файла нам при запуске не указали и можно вызвать `process_stdin`, в противном случае нужно применить `process_file` ко второму элементу списка. Всё это можно организовать простой формой `if`, а чтобы обработать исключительные ситуации, мы её форму поместим в первый аргумент формы `handler-case`. Первый обработчик сомнений не вызывает — его мы напишем для исключений типа `file-error`. Но прежде чем завершить написание вызова `handler-case`, вспомним, к каким противным последствиям приводит нажатие во время работы программы комбинации `Ctrl-C`, столь привычной любому пользователю Unix. Интерпретатор это тоже относит к исключительным ситуациям, так что давайте заодно справимся и с этим; в SBCL тип исключения для этого — `sb-sys:interactive-interrupt` (да, спецификация Common Lisp и тут предпочла промолчать, поэтому имя такое странное — используется символ из «пакета» `sb-sys`). Из объекта исключения такого типа нам вообще ничего не извлечь, так что имя переменной мы в его обработчике опустим. В итоге получится вот что:

```
(handler-case
    (if (cdr *posix-argv*)
        (process_file (second *posix-argv*))
        (process_stdin)
    )
    ;;
    (file-error (er)
        (princ er *error-output*)
        (terpri *error-output*)
    )
    (sb-sys:interactive-interrupt () nil)
)
```

Программа, написанная таким образом, будет при трудностях с открытием файла печатать лаконичное сообщение об ошибке и завершаться, а при нажатии `Ctrl-C` — просто завершаться, ничего не говоря, как, собственно, и должны поступать все порядочные программы. Читателю, дочитавшему досюда и не сломавшемуся раньше, предложим принять наше восхищение, а заодно оценить объём получившегося у нас параграфа; а ведь бóльшая его часть посвящена, скажем начистоту, борьбе с мракобесием создателей спецификации Common Lisp и конкретных её реализаций. Мы вынуждены были всем этим заниматься *просто чтобы получить нормальную программу* — то, что при работе с другими языками воспринимается как само собой разумеющееся.

11.1.15. Неутешительное заключение

К сожалению, из последних трёх параграфов следует совершенно неизбежный вывод: все реально существующие реализации «стандарта» Common Lisp категорически не годятся для практического использования, даже если допустить, что сам этот стандарт хоть чем-то полезен. Скажем больше: они не особенно подходят даже в роли учебных пособий для изучения Лиспа, но, увы, ничего другого просто нет, все альтернативные диалекты Лиспа либо умерли, либо существуют в роли предметно-ориентированных языков (выше мы уже упоминали AutoLisp и Emacs Lisp) и вдобавок оказываются слишком куцыми.

Автор книги продолжает надеяться на лучшее, ведь если отбросить всё мракобесие, придуманное создателями Common Lisp, то оставшийся Лисп «как таковой» — это одна из самых красивых концепций в истории развития языков программирования. Возможно, когда-нибудь мы ещё увидим правильные и пригодные для работы диалекты этого языка, создатели которых найдут в себе силы наплевать на «авторитет» комитета X3J13, породившего бастарда по имени Common Lisp и фактически уничтожившего Лисп как явление.

Так или иначе, в контексте нашего разговора о парадигмах, то есть о том, как и с каких точек зрения можно осмысливать программирование, полезно попытаться сформулировать — по возможности кратко и ясно — некий принцип, который следует соблюдать, но который нарушили создатели Common Lisp. Принцип очень прост: **поведение программы, написанной на языке программирования общего назначения, должно целиком и полностью определяться текстом этой программы, а не особенностями языка программирования или применяемого транслятора**.

11.2. Scheme: Лисп, но не совсем

История появления языка Scheme нам уже знакома (см. §11.1.1); мы неоднократно упоминали одно из основных достоинств Scheme — *лаконичность* спецификации. Создатели R6RS попытались эту лаконичность нарушить, но последовавшие за этим события показали, что в IT ещё остались люди, сохраняющие остатки здравого смысла.

Эта глава нашей книги будет короче, чем предыдущая, поскольку во многом Scheme всё же остаётся Лиспом, так что нам нужно лишь показать её ключевые отличия от «традиционных» диалектов Лиспа.

11.2.1. Chicken Scheme

По традиции предложим читателю один из вариантов реализации Scheme, чтобы было на чём попробовать описываемые возможности; для этого мы выберем одну из самых популярных сейчас свободных реализаций — Chicken Scheme.

После установки в системе соответствующего пакета (в системах семейства Debian пакет называется `chicken-bin`) вы получите в своё распоряжение две команды: `csi` — интерактивный интерпретатор языка Scheme и `csc` — компилятор[17]. Названия команд представляют собой сокращения от *Chicken Scheme Interpreter* и *Chicken Scheme Compiler*.

Компилятор `chicken` устроен довольно неожиданным образом: он переводит исходный текст, написанный на языке Scheme, *в текст на языке Си*. Для Scheme это самый распространённый способ трансляции. Выглядит этот текст так, что читателю вряд ли захочется в нём разбираться; компилятор Си используется здесь как промежуточное звено, но об удобстве чтения сгенерированного кода никто, по-видимому, не задумывался. Впрочем, увидеть этот код можно лишь если специально этого добиваться, а в обычном режиме команда `csc`, прогнав компилятор `chicken` и получив файл на Си, тут же применяет к нему компилятор Си (хорошо известный нам `gcc`), а все временные файлы стирает.

Сразу же отметим одну особенность Chicken Scheme, серьёзно противоречащую как спецификациям, так и привычкам многих сторонников лиспоподобных языков: по умолчанию эта реализация чувствительна к регистру букв в именах символов, примерно как в Си, тогда как традиционно в лиспоподобных языках регистр в именах символов всегда игнорировался. Если вам это не нравится, и компилятор, и интерпретатор позволяют эту чувствительность отключить, применив флажок `-i`.

Если интерпретатор `csi` запустить без параметров, он войдёт в традиционный интерактивный режим, в котором с клавиатуры читаются S-выражения и сразу же вычисляются (выполняются), а результат печатается — уже известный нам ***REPL*** (см. стр. 110). Средствами редактирования вводимой строки интерпретатор не оснащён, так что есть смысл запускать его через утилиту `rlwrap` (см. стр. 110), работать так будет удобнее. Если при запуске `csi` указать в аргументах командной строки один или несколько файлов, содержащих текст на Scheme, эти файлы будут загружены в интерпретатор, содержащиеся в них формы верхнего уровня — вычислены, после чего интерпретатор войдёт в режим REPL, и в нём будут доступны функции, описанные в файлах. Загрузить файл в интерпретатор можно и после его запуска — с помощью псевдофункции `load` (например, `(load "myfile.scm")`) или *команды* `,l` (строка вроде «`,l myfile.scm`» вводится без скобок, поэтому `,l` считается именно командой, а не чем-то ещё).

Интерпретатор `csi` можно использовать для создания традиционных скриптов, причём для этого он предусматривает два флага командной строки, работающих по-разному. В простом варианте можно воспользоваться флажком `-s` или, что то же самое, `--script`. Например, сакраментальное «Hello, world» может выглядеть так:

```
#!/usr/bin/csi -s
(display "Hello, world!")
(newline)
```

В отличие от классических лиспов, Scheme в строковых литералах понимает обозначения символов, традиционные для Си, так что можно написать и короче:

```
#!/usr/bin/csi -s
(display "Hello, world!\n")
```

Примечательно, что такой скрипт можно без малейших проблем откомпилировать:

```
avst@host:~/scheme$ cat h.scm
#!/usr/bin/csi -s
(display "Hello, world!\n")
avst@host:~/scheme$ csc h.scm
avst@host:~/scheme$ ./h
Hello, world!
avst@host:~/scheme$
```

Подробное рассмотрение функций для ввода-вывода мы оставим для отдельного параграфа (см. §11.2.3). Для доступа к параметрам командной строки Chicken Scheme предусматривает функцию `command-line-arguments` — параметров она не принимает, а возвращает список строк — элементов командной строки *без* `argv[0]`, т. е. имени программы. Если имя программы всё же понадобится, его можно получить с помощью другой функции — `program-name`. Надо сказать, что эти функции не входят в спецификации R^{n}RS и являются особенностью именно Chicken; к примеру, в Gambit Scheme таких функций нет, зато есть функция `command-line`, которая тоже возвращает список строк, но этот список включает все аргументы командной строки, в том числе имя программы.

Доступность функций, которые авторы Chicken относят к «расширениям», в том числе и функций доступа к аргументам командной строки, зависит от того, какую версию Chicken вы будете использовать. На момент написания этой книги последние версии имели номера `5.*`, но в дистрибутивах Linux (и вообще много где) ещё встречаются и довольно долго будут встречаться установки Chicken предыдущих, четвёртых (`4.*`) версий. Если вы используете четвёртую версию, функция `command-line-arguments` будет доступна без дополнительных ухищрений, а вот если у вас уже пятая версия, то, чтобы получить эту функцию в своё распоряжение, в начало программы придётся добавить строку `(import (chicken process-context))`. Например, команду `echo` можно на Chicken Scheme версий 4.* реализовать так:

```
#!/usr/bin/csi -s
(map
    (lambda (s) (display s) (display " "))
    (command-line-arguments)
)
(newline)
```

Для пятой версии пример будет выглядеть чуть иначе:

```
#!/usr/bin/csi -s
(import (chicken process-context))
(map
    (lambda (s) (display s) (display " "))
    (command-line-arguments)
)
(newline)
```

Вполне возможно, что к тому времени, когда вы это будете читать, выйдут ещё более новые версии Chicken Scheme, и как в них будет обстоять дело, сейчас никто предсказать не возьмётся; во всяком случае, к обратной совместимости создатели Chicken относятся, как видим, не слишком серьёзно.

Ещё в Chicken Scheme предусмотрена функция, которая так и называется `argv`, но она работает по-разному для скриптов и для откомпилированных исполняемых файлов, что делает её довольно неудобной.

Более сложный способ создания скриптов на Chicken Scheme требует флага `-ss` (в отличие от `-s`). В этом случае ваша программа (скрипт) должна содержать функцию с именем `main` и одним параметром; интерпретатор вызовет эту функцию, передав ей в качестве фактического параметра список аргументов командной строки — ровно такой, как возвращает функция `command-line-arguments`. «Hello, world» с использованием этого метода будет выглядеть так:

```
#!/usr/bin/csi -ss
(define (main args)
    (display "Hello, world!\n")
)
```

[17] Строго говоря, компилятор как таковой называется просто `chicken`, а `csc` — это программа, запускающая сначала его, а затем хорошо известный нам `gcc`.

а команду `echo` можно реализовать так:

```
#!/usr/bin/csi -ss
(define (main args)
    (map (lambda (s) (display s) (display " ")) args)
    (newline)
)
```

Программы (скрипты), оформленные с помощью флага `-ss`, удобны тем, что их можно загрузить в интерактивный интерпретатор (указав в командной строке с помощью функции `load` или команды `,l`), при этом не запуская программу; это позволит попробовать в работе отдельные функции из неё.

Откомпилировать такой скрипт несколько сложнее, чем обычный (который с флагом `-s` и без функции `main`). При запуске компилятора `csc` нужно добавить в командную строку `-postlude "(main (command-line-arguments))"`, или просто `-postlude "(main '())"`, если ваша функция `main` не пользуется своим параметром.

Получающиеся с помощью `csc` исполняемые файлы довольно скромны по своим размерам — всего несколько десятков килобайт, но они при этом зависят от разделяемых (динамических) библиотек, а сами эти библиотеки довольно объёмны. Если требуется выполнять полученную программу на машине, на которой нет Chicken Scheme, можно воспользоваться флагом `-static`, чтобы `csc` создал полностью автономный исполняемый файл, не зависящий ни от каких библиотек. Особенно приятна возможность убрать при этом изрядную часть стандартных функций, чтобы сократить размер исполняемого файла. По умолчанию `csc` подключает модули `library` и `eval`; без первого не обойдётся даже самая простенькая программа, тогда как второй оказывается не нужен, если ваша программа не использует примитив `eval`. Флаг `-x` отменяет использование обоих модулей, а параметр `-uses library` явным образом подключает `library` обратно. Наконец, флаг `-strip` убирает из готового исполняемого файла отладочную информацию. Итоговый вызов компилятора может выглядеть так:

```
csc -static -strip -x -uses library echo.scm \
    -postlude '(main (command-line-arguments))'
```

На машине вашего покорного слуги результат такой компиляции программы `echo` «весил» чуть больше 2 MB — весьма неплохо на фоне монстров, порождаемых разными реализациями Common Lisp, но, конечно, многовато, если вспомнить, что это всего лишь `echo`.

11.2.2. Видимые отличия Scheme от обычного Лиспа

Для начала перечислим наиболее очевидные отличия, которые при ближайшем рассмотрении оказываются сугубо косметическими. В традиционных диалектах Лиспа символ с именем `nil`, как мы видели, существенно перегружен — это и просто символ, и пустой список (так что `()` приходится считать синонимом `nil`), и логическая ложь. В Scheme пустой список обозначается только пустыми круглыми скобками, и больше никак; для логических выражений введены специальные атомарные выражения (заметим, не являющиеся символами!) — `#t` (истина) и `#f` (ложь); истиной, впрочем, считается всё, что не `#f`. Использовать символ с именем `nil`, конечно, можно, но никакой особой роли для него не предусмотрено, это просто обычный символ.

К косметическим отличиям следует отнести также подход к именованию встроенных (если угодно, библиотечных) функций. В Scheme все предикаты, то есть такие функции, которые проверяют некое условие и возвращают логическое значение, имеют имена с вопросительным знаком на конце: `null?`, `eql?`, `atom?`, `number?` и т. п.; разрушающие примитивы обозначены восклицательным знаком: `set!`, `set-car!`, `set-cdr!`, `vector-fill!`. Аналога громоздкой функции `coerce` из Common Lisp (см. стр. 127) здесь нет, вместо этого предусмотрен целый ряд функций преобразования, таких как `string->list`, `number->string`, `char->integer` и т. п. Функции сравнения для чисел выглядят вполне привычно: `=`, `<`, `>`, `<=`, `>=` (при этом, как ни странно, отсутствует функция «не равно»); предусмотрены операции сравнения и для нечисловых типов, но выглядят они несколько непривычно: для символов это `char=?`, `char<=?`, для строк — `string>?` и т. п.

При переходе с Лиспа на Scheme начинающие часто ищут знакомую функцию `mapcar`; она здесь есть, но называется просто `map`. Кроме того, присутствует ещё функция `for-each`, предназначенная для вызова ради побочного эффекта; она работает точно так же, как `map`, но не формирует списка из результатов вызовов, экономя немного времени, если такой список не нужен.

Ещё один довольно забавный момент состоит в том, что в Scheme функции не всегда возвращают значение; точнее говоря, **в Scheme значение для некоторых встроенных функций, форм и макросов не определено**. Классический пример этого — форма `set!`; вспомним, что её лисповский аналог — `setq` — возвращает присвоенное значение (точнее, последнее из присвоенных значений, поскольку их может быть больше одного); точно так же не возвращают никакого значения форма `define`, функции `display`, `for-each` и некоторые другие примитивы. Что несколько странно, форма `if`, в которой отсутствует третий аргумент (ветка *else*), при истинном условном выражении вернёт значение, полученное как результат выполнения второго аргумента, а при ложном условии — ничего не вернёт; из этого следует, что `if` с отсутствующим выражением для *else* вообще нельзя использовать как составную часть более сложных выражений.

Выглядело бы намного логичнее, если бы `if`, будучи вызванной в неполном виде, не возвращала никакого значения вне зависимости от результата условия. Такого не сделали, потому что это потребовало бы лишних движений. Как мы помним, в Scheme выполнение оптимизации «хвостовых» вызовов является одним из ключевых требований к реализации. Как следствие, вычисление любой формы, как функциональной, так и специальной, может завершиться не фиксацией возвращаемого значения, а выдачей выражения, которое нужно вычислить для получения результата. Это как раз и позволяет ликвидировать стековый фрейм текущей функции *до* того, как будет вычислена её последняя форма, в которой может быть рекурсивный вызов (но не обязательно).

Вычисление `if` обычно реализуется совершенно очевидным способом: эта форма сначала вычисляет условие, а затем в зависимости от полученного результата отдаёт интерпретатору свой второй или третий аргумент (соответственно ветку *then* или ветку *else*) с указанием, что это выражение нужно ещё вычислить. Если форма `if` вызвана в неполном виде и при вычислении условия получилась ложь, форма немедленно возвращает управление, указав интерпретатору, что результат не определён (*unspecified value*), но если получилась истина, `if` действует обычным для себя способом, не дожидаясь окончания вычисления своего второго аргумента.

Функция, введённая с помощью `define` или как результат лямбда-списка, тоже может не возвращать значения. Очевидный способ этого добиться — завершить её тело какой-то формой, которая сама ничего не возвращает; но есть и средство специально для этого — псевдофункция `void`: выражение `(void)` специально предназначено, чтобы порождать сакраментальное *unspecified value*, которое в действительности очень даже «определено».

Есть и ещё один, менее тривиальный способ не вернуть значение из функции. В Scheme (как, впрочем, и в Common Lisp) есть средства для создания функций, возвращающих несколько значений; такую функцию нужно вызывать специальным образом — с помощью `multiple-value-list` в Common Lisp или `call-with-values` в Scheme. В Common Lisp функцию, возвращающую несколько значений, можно вызвать и обычным путём, тогда все значения, кроме первого, будут проигнорированы; в Scheme результат такого вызова не определён. Интересно, что примитив для собственно *возврата* нескольких значений в обоих диалектах называется одинаково: `values`; это встроенная функция, принимающая произвольное количество параметров, которые и станут в итоге значениями. Вполне логично, что вызов `values` без параметров приводит к возвращению *нуля* параметров, то есть функция в итоге ничего не возвращает.

Между прочим, вызвать `values` без параметров можно и в Лиспе, притом с тем же эффектом, но там это выглядит чужеродным, поскольку все встроенные функции, формы и макросы там хоть что-нибудь да возвращают.

На этом косметические отличия заканчиваются и начинается то, чем Scheme *действительно* отличается от традиционных диалектов.

Как ни странно, **едва ли не основное и далеко идущее отличие состоит в том, что в Scheme у символов нет «ассоциированной функции» как отдельной сущности**; если символ служит именем для функции (неважно, встроенной или введённой программистом), то объект функции является для этого символа *значением*. Это может показаться не столь серьёзной разницей, но торопиться не стоит; сейчас мы рассмотрим одно за другим вытекающие отсюда следствия, и к концу этого рассмотрения читатель, можно надеяться, более не будет полагать концепцию «функция как значение символа» незначительной.

Итак, следствие первое. Как мы помним, интерпретатору при вычислении формы сначала нужно понять, не обозначает ли её первый элемент нечто специфическое (для традиционных лиспов — спецформу или макрос). Если первый элемент списка задаёт обычную функцию — будучи либо символом, именующим функцию, либо лямбда-списком — то в традиционных диалектах Лиспа следующим шагом будет вычисление всех элементов формы, *кроме* первого, а из первого элемента функциональный объект извлекается

(или строится) с помощью того же механизма, который задействует спецформа `function` (напомним, что её вызов может быть сокращённо записан в виде `#'`; из символов она извлекает объект ассоциированной функции, из лямбда-списков — строит новое замыкание). Так вот, **в Scheme вычисляются все элементы формы вызова функции, включая первый**; никакие специальные средства к первому элементу формы не применяются, если только он не связан со спецформой или макросом.

Из этого вытекают ещё два следствия. Во-первых, в Scheme ***лямбда-список*** (заметим, строящийся практически по тем же правилам, что и в традиционных лиспах) не представляет собой ничего волшебного, это просто *спецформа*, которая *вычисляется*, а полученное замыкание возвращает в качестве своего значения; ничего особенного не представляет собой и сам символ `lambda`, он просто именует спецформу — так же, как `if` или `define`. Во-вторых, в Scheme отсутствует за ненадобностью какой бы то ни было аналог лисповской формы `function` и её сокращения `#'` — оба режима её работы в Лиспе (извлечение функционального объекта из символа и создание нового замыкания из лямбда-списка) превращаются в Scheme в простое вычисление.

Отсутствует в Scheme также и примитив `funcall` — он просто не нужен, ведь в ситуации, где в Лиспе мы должны были написать что-то вроде `(funcall f a b c)`, в Scheme мы можем написать просто `(f a b c)`, поскольку функция есть *значение* символа, а первый элемент формы *вычисляется*. Отметим, что функция `apply` в Scheme есть, и работает она точно так же, как в традиционных лиспах.

Коль скоро значение и ассоциированная функция для символа не различаются, вполне логично выглядит и то, что для их задания в Scheme используется одна и та же спецформа `define`. Так,

```
(define myvar 275)
```

определяет переменную `myvar` с начальным значением `275`, а

```
(define (plus3 x) (+ 3 x))
```

— функцию `plus3`, которая прибавляет `3` к своему аргументу. Здесь наблюдается ещё одно косметическое различие: в Лиспе форма `defun` подразумевает, что первым аргументом идёт имя функции, тогда как здесь имя и формальные параметры составляют единый список. Так сделано, чтобы этот вариант можно было отличить от предыдущего, где задаётся начальное значение. В принципе мы могли бы то же самое написать и иначе:

```
(define plus3 (lambda (x) (+ 3 x)))
```

Вариант со списком в первом параметре считается сокращённой формой такой записи.

Раз уж мы затеяли обсуждение формы `define`, отметим ещё две её особенности. Во-первых, эта форма может выступать либо формой верхнего уровня (*top-level form*), т. е. находиться на верхнем уровне программы, не будучи ни во что вложенной, либо её можно поставить *в начале блока* — то есть в начале тела функции либо тела одной из форм семейства `let`. В первом случае переменная объявляется глобальная, во втором — локальная (так что если в объемлющем или глобальном контексте есть переменная с таким же именем, новая переменная её *затеняет*, как это обычно бывает с локальными переменными). Расположить вызов `define` где-то ещё считается ошибкой, хотя не все реализации это проверяют.

Во-вторых, список, служащий первым аргументом `define`, может быть *неправильным* (точечным). Это используется для объявления ***вариадических функций***. Например,

```
(define (myvarfunc x y . rest)
    ; ...
)
```

вводит функцию `myvarfunc`, которая принимает *не менее двух* параметров, то есть два, три, пятнадцать, пятьсот — сколько угодно, лишь бы не меньше двух. Первые два фактических параметра связываются с переменными `x` и `y`, из остальных формируется список, который связывается с переменной `rest`.

В Common Lisp такое тоже возможно:

```
(defun myvarfunc (x y &rest rest)
    ; ...
)
```

Возвращаясь к рассказу о ключевых словах Common Lisp (см. стр. 127), заметим, что вариадические функции — это единственное реально полезное применение всего этого кошмарного «хозяйства». Создатели Scheme, как видим, эту единственную нужную возможность предоставили, используя точечные списки, которые и так есть, и избежали введения громоздких и некрасивых ключевых слов или каких-либо их аналогов.

Заодно можно заметить, что в Scheme нет вообще ничего из того, что мы упоминали в качестве нежелательных элементов Common Lisp в §11.1.13: ни аналога функции `coerce`, ни обобщённого присваивания вроде `setf`, ни управляющих конструкций, завязанных на множество разнообразных символов, таких как Common Lisp'овский `loop`.

К сожалению, от груза устаревших парадигм, которые мы упоминали в 11.1.12, Scheme никоим образом не избавлена, что и понятно: в середине 1970-х, когда она появилась, эти парадигмы устаревшими не были. Получить строковое имя для данного символа можно с помощью функции `symbol->string`, а найти (или создать, если найти не удалось) символ по заданному имени — функцией `string->symbol`. Символы с произвольными знаками в именах делаются в Scheme точно так же, как в Common Lisp — заключением имени символа между «вертикальными палочками», что-то вроде `|I'm A SyMbOL, too|`.

С символами и их значениями в Scheme связаны ещё два момента, которые следует учитывать. Во-первых, **в Scheme изначально существовало только лексическое связывание** (если вы не уверены, что понимаете, о чём идёт речь, перечитайте §11.1.9). Во-вторых, согласно спецификации символ должен быть сначала определён с помощью `define` или какого-то из вариантов `let`, и лишь после этого его можно присваивать с помощью `set!`, а до тех пор символа как бы не существует. Впрочем, это ограничение отслеживают далеко не все реализации; в частности, Chicken никакого возмущения при присваивании значения новому символу не высказывает.

R7RS предусматривает аналог динамического связывания: формы `fluid-let`, `make-parameter` и `parameterize`; Chicken всё это поддерживает, хотя в R5RS этих средств не было. В сочетании с континуациями динамическое связывание превращается в материю довольно странную — в частности, *динамическое время существования* (англ. *dynamic extent*) перестаёт быть промежутком времени между входом в некую процедуру и выходом из неё, поскольку к выполнению этой процедуры позднее возможен возврат. Мы не будем рассматривать динамическое связывание в Scheme: для этого языка оно чужеродно.

В качестве фирменных особенностей Scheme в добавок к её лаконичности обычно называют, во-первых, систему гигиенических макросов, и, во-вторых, так называемые ***континуации*** (англ. *continuations*; в ряде русскоязычных источников этот термин переведён просто как «продолжения»). Макросы мы рассматривать не будем точно так же, как не рассматривали их для Common Lisp; что касается континуаций, то они заслуживают подробного обсуждения, которому мы чуть позже посвятим отдельный параграф.

11.2.3. Ввод-вывод в Scheme

Рассказывая в §11.2.1 о Chicken Scheme, мы в примерах неоднократно использовали функцию `display`, которая, будучи вызванной с одним параметром, выдаёт этот свой параметр в поток стандартного вывода; там же встречаются и вызовы функции `newline`, которая выдаёт перевод строки.

Кроме функции `display`, для вывода в Scheme предусмотрена функция `write`. Эти две функции аналогичны функциям `princ` и `prin1` из Лиспа (см. стр. 129): когда речь идёт о выводе символов и строк, `display` выводит сами заданные строки и символы, тогда как `write` печатает их *представление* по правилам Scheme — заключает строки в двойные кавычки, а для одиночных символов выводит их обозначения, такие как `#\A`, `#\newline` и т. п.; выражения остальных типов эти функции выдают одинаково. Как и в Лиспе, в Scheme предусмотрена функция `read`, позволяющая прочитать S-выражение (в том числе ранее выданное с помощью `write`, но не `display`); и, как и ранее при обсуждении Лиспа, мы настоятельно рекомендуем сразу же выкинуть эту возможность из головы; см. обсуждение этого вопроса и комментарий на стр. 129.

Отказавшись от `read`, мы, как и для Лиспа, оставляем себе только возможность посимвольного ввода; в Scheme для этого предусмотрена функция, называющаяся, как и в Лиспе, `read-char`, и здесь нас ожидает приятный сюрприз: в отличие от Лиспа, где нам пришлось изрядно попотеть, чтобы функция не ломалась на ситуации конца файла, в Scheme всё изначально сделано правильно: `read-char` возвращает либо прочитанный символ, либо некий специальный объект, символизирующий наступление конца файла. Что

это за объект, спецификации умалчивают; таких объектов теоретически даже может быть больше одного. Чтобы опознать конец файла, следует воспользоваться функцией `eof-object?`, которая получает один аргумент и возвращает истину (`#t`), если её аргумент — это «объект конца файла», в противном случае возвращает ложь (`#f`).

Отметим, что в Scheme предусмотрена также и функция `write-char` для вывода отдельно взятого символа, хотя `display` тоже позволяет это сделать. Строго говоря, согласно спецификации `write-char` — это *процедура*, а `display` — *библиотечная процедура*; таким термином в R5RS обозначаются функции, которые могут быть написаны на Scheme и в определённой степени избыточны (введены просто для удобства), тогда как простые «процедуры», без слова «библиотечная», представляют собой примитивы, обеспечивающие функциональную полноту: если бы их не было, то написать их было бы нельзя, во всяком случае, без использования кода на языках, отличных от Scheme.

Любопытно наличие в Scheme функций `peek-char` и `char-ready?`. Первая позволяет узнать, какой символ будет прочитан следующим, без собственно чтения символа; вторая сообщает нашей программе, есть ли уже в заданном потоке («порте») символ, готовый к прочтению. Впрочем, радоваться рано: согласно документации Chicken Scheme, `char-ready?` всегда возвращает `#t` (истину) для терминальных портов, так что услуг по перепрограммированию терминала (см. т. 2, §5.4.3) нам Chicken Scheme не предоставляет. Кстати, это решение, пожалуй, следует признать правильным — не дело языка программирования заниматься такими вещами.

Пока мы ограничиваемся в работе стандартным вводом и выводом, всё более-менее просто, но необходимость работы с файлами, как водится, картину резко усложняет. Спецификация R5RS вообще не предусматривает никакой обработки ошибок при открытии файла, в её тексте сказано только, что «возникает ошибка» (*an error is signaled*). Между прочим, там же говорится, что при открытии файла на запись, если файл при этом уже существует, результат *неопределён* (*the effect is unspecified*), и не спрашивайте, какие вещества создатели этой спецификации употребляли внутрь.

R7RS предусматривает обработку исключений; создатели Chicken Scheme в своей документации не упоминают R7RS, но обработку исключительных ситуаций, очень похожую на ту, что описана в R7RS, всё же реализовали, так что *перехватить* ошибку, не позволив реализации обрабатывать её по-своему, в Chicken Scheme можно; при этом построить собственную полноценную обработку ошибок, увы, практически невозможно. Остаток параграфа мы посвятим тому, как сделать в этом плане хотя бы то, что в наших силах.

Отметим прежде всего, что в терминологии, принятой в описаниях Scheme, потоки ввода-вывода почему-то называются *портами* (*ports*). Зачем так обращаться с терминами — вопрос, видимо, к создателям ранних спецификаций Scheme. Интересно, что эти «порты» могут работать либо на ввод, либо на вывод, но двунаправленных портов спецификации не предусматривают.

Получить объекты портов, соответствующие стандартным потокам, можно с помощью функций `current-input-port` и `current-output-port`; создатели Chicken Scheme добавили ещё функцию `current-error-port`, которая возвращает объект порта, соответствующего диагностическому потоку (`stderr`). Подчеркнём, что эти `current-...-port` — именно функции, а не что-то другое, и их, чтобы получить нужный объект, следует *вызвать* (без аргументов). Например, выдать диагностическое сообщение в Chicken Scheme можно так (обратите внимание на скобки вокруг `current-error-port`):

```
(display "We're on fire\n" (current-error-port))
```

Слово *current* в именах функций, которое переводится как «текущий», наводит на мысль, что эти порты должно быть возможно как-то поменять. Как ни странно, автору не удалось понять, как это сделать.

Объект порта указывается при вызове функций ввода и вывода дополнительным аргументом — для `newline` и `read-char` он будет единственным, для `display`, `write` и `write-char` — вторым.

Открыть файл на чтение или запись можно с помощью функций `open-input-file` и `open-output-file`. Каждая из них принимает ровно один аргумент — имя файла в виде строки; значение, которое эти функции возвращают, всегда представляет собой объект открытого порта. Если произошла ошибка любого рода, функции управление не возвращают, а система выполнения — будь то интерпретатор или библиотека времени исполнения для случая откомпилированной программы — ударяется в панику (ну а как ещё это всё назвать?)

Прежде чем пытаться с этой паникой справиться, отметим ещё одну довольно странную особенность Scheme. Для закрытия портов ввода и портов вывода предусмотрены *разные* функции: `close-input-port` и `close-output-port`. Из каких соображений их разделили, остаётся непонятным.

Вернёмся к исключениям, возникающим в результате ошибки при открытии файла. Chicken Scheme позволяет обрабатывать исключения в соответствии со спецификацией srfi-12[18]. Самый простой примитив для перехвата исключений называется `with-exception-handler`. Он вызывается с двумя аргументами. Первый аргумент представляет собой функцию-обработчик, то есть ту функцию, которая должна быть вызвана, если исключение возникнет. Эта функция должна иметь ровно один параметр, с которым будет связан объект исключения. Второй аргумент представляет собой, опять же, *функцию* (этот момент часто служит источником непонимания), только уже с пустым списком параметров. Примитив `with-exception-handler` *применяет* свой второй аргумент к пустому списку фактических параметров, при этом на всё время его вычисления *первый* аргумент устанавливается в качестве обработчика исключений.

Обычно обоими аргументами `with-exception-handler` служат lambda-выражения. Довольно неочевидным оказывается следующий момент: *обработчик исключения не должен возвращать управление обычным путём*, в противном случае то же самое исключение возникнет снова. Для выхода из него можно воспользоваться механизмом континуаций, который будет рассмотрен в следующем параграфе; в нашем примере мы поступим проще — завершим выполнение программы с помощью `exit`. Всё вместе должно выглядеть примерно так:

```
(with-exception-handler
    (lambda (ex)  ; функция-обработчик
        ; что делать, если произошла ошибка
        ; ex -- объект исключения
        (exit 1)
    )
    (lambda ()    ; функция, содержащая основной код
        ; здесь пишется код, в котором может
        ; возникнуть исключительная ситуация
    )
)
```

Отдельное происшествие — пресловутый «объект исключения». Увы, с этим всё плохо, но после Common Lisp нас вряд ли можно чем-то удивить. Так или иначе, ничего полезного — во всяком случае, для ошибки, связанной с открытием файла — этот объект не содержит, из него можно разве что вытащить строковое сообщение, которое, как можно догадаться, самостоятельно проанализировать наша программа всё равно не сможет, поскольку оно зависит от текущей локали. Заинтересованному читателю мы предложим применить к объекту исключения функцию `condition->list` и полюбоваться на получившийся развесистый список, содержащий массу всякой ерунды, включая трассу вызовов функций с указанием их имён и номеров строк файла исходного текста программы, и не содержащий ничего из того, что нужно.

Попробуем переписать на Chicken Scheme программу, которую мы раньше писали на Common Lisp — читающую текстовый файл и печатающую длины прочитанных строк (см. стр. 132). Как мы знаем, программу на Chicken Scheme можно и компилировать, и выполнять интерпретатором (как скрипт). Чтобы сделать скриптовое выполнение возможным, начнём программу (файл `strlens_file.scm`) со строки

```
#!/usr/bin/csi -s
```

Компилировать программу нам это не помешает — компилятор `csc` эту строку просто проигнорирует.

Дальнейшее зависит от того, какую версию Chicken вы будете использовать (см. замечание на стр. 133). Если вы используете четвёртую версию, две следующие строки вам не потребуются, а вот если у вас уже пятая версия, то, чтобы получить в своё распоряжение два «расширения» Chicken — а именно форму `with-exception-handler` и функцию `exit` — придётся сделать так:

```
(import (chicken condition))
(import (chicken process-context))
```

[18] Тексты серии SRFI представляют собой предложения по тому, какие ещё возможности хорошо бы включить в язык; аббревиатура расшифровывается как *Scheme Request For Implementation*, т. е. «запрос на реализацию в Scheme».

Перепишем теперь на Scheme функцию `do_it`:

```
(define (do_it port len)
    (let ((c (read-char port)))
        (cond
            ((eof-object? c) #t)
            ((eqv? c #\newline)
                (display len)
                (newline)
                (do_it port 0)
            )
            (else (do_it port (+ 1 len)))
        )
    )
)
```

Теперь настаёт черёд неприятной чёрной магии — формы `with-exception-handler`. Обработчик мы сделаем примитивным: заставим его напечатать список, полученный из объекта исключения с помощью `condition->list`, затем переведём строку и выведем уже своё собственное сообщение об ошибке; всё это направим в диагностический поток. Что касается основной функции, то, чтобы не писать несколько раз громоздкое `(command-line-arguments)`, мы с помощью `let` свяжем полученный от неё список параметров с переменной, а затем, если список пустой, вызовем `do_it` для стандартного потока ввода, если же список не пуст, воспользуемся первым его элементом в качестве имени файла, откроем этот файл на чтение, вызовем `do_it`, затем закроем файл. Получится примерно следующее:

```
(with-exception-handler
    (lambda (ex)
        (write (condition->list ex) (current-error-port))
        (newline (current-error-port))
        (display "something wrong (failed to open?)\n"
                 (current-error-port))
        (exit 1) ; without this, endless loop occurs
    )
    (lambda ()
        (let ((args (command-line-arguments)))
            (if (null? args)
                (do_it (current-input-port) 0)
                (let ((port (open-input-file (car args))))
                    (do_it port 0)
                    (close-input-port port)
                )
            )
        )
    )
)
```

Как ни странно, весь этот откровенный *бред* — ещё не приговор языку Scheme. Благодаря трансляции «через Си» программы на Scheme могут легко обращаться к функциям, написанным на Си, и наоборот; это позволяет всё взаимодействие с внешним миром (то, с чем у Scheme, как видим, просто беда) возложить на модули, написанные на Си, а для кода на Scheme оставить всевозможную нетривиальную логику. Возможно, отчасти поэтому Scheme в наше время применяется на практике намного чаще, чем Common Lisp.

11.2.4. Континуации (продолжения)

Континуацию, или ***продолжение*** (англ. *continuation*), проще всего воспринимать как некий «список того, что осталось сделать, чтобы закончить вычисления». Конечно, одной фразой объяснить этот термин не получится, поэтому попробуем конкретизировать вводные. Представьте себе, что ваша программа выполняет тело некоторой функции. Всё, что ещё успеет произойти до момента, когда функция вернёт управление, нас (условно) не волнует; но когда-то она всё же вернёт управление тому, кто её вызвал — и (за исключением некоторых особых ситуаций) вернёт при этом значение.

Тот, кто нашу функцию вызвал, получит это значение, после чего (возможно, сразу, а может быть, и очень сильно не сразу) завершит вычисление своего тела и тоже вернёт значение; *его* вызывающий, в свою очередь, должен будет завершить выполнение своего тела и т. д., и так пока не закончится программа. Вся эта последовательность действий во всём её великолепии — то есть те действия, которые будут выполнены с момента возврата из текущей функции до окончания выполнения программы — и есть, собственно говоря, континуация; обычно говорят о *континуации текущей функции*, или просто о *текущей континуации*.

Здесь следует обратить внимание на то, что **работа континуации зависит от того, какое значение вернёт текущая функция**. Этот факт понадобится нам несколькими абзацами ниже, пока просто запомним его.

К этому моменту у читателя могло сложиться впечатление, что континуация представляет собой некую чисто умозрительную абстракцию, но это впечатление обманчиво. Припомнив, как устроен аппаратный стек и как его используют программы, написанные на компилируемых фоннеймановских языках, таких как Паскаль, Си или Си++, мы можем заметить, что континуация имеет достаточно очевидное представление в виде конкретной информации, хранящейся в памяти. Так, если бы мы попытались реализовать континуации в чистом Си, то нам хватило бы содержимого стека от его «подножия» — адреса, где начинается секция стека в пространстве нашего процесса — до того места, где располагалась вершина стека перед вызовом текущей функции. В самом деле, в стеке хранятся *адреса возврата* в тела всех функций, которые к настоящему моменту были вызваны, но ещё не завершились, так что именно содержимое стека определяет, какие ещё действия будут выполнены в каждой из этих функций, прежде чем она вернёт управление. Конечно, сами эти действия определяются скорее машинным кодом, расположенным начиная с адресов возврата, но ведь машинный код в ходе работы программы не меняется.

Выполнение программы, написанной на Scheme и других нефоннеймановских языках, может быть организовано совершенно не так, как выполнение программы на процедурном языке; в частности, аппаратный стек может либо не использоваться вовсе, либо использоваться, но не так, как мы могли бы ожидать (к этому вопросу в применении к Scheme мы ещё вернёмся); тут важно другое: континуация не представляет собой ничего такого, что было бы слишком сложно *описать в виде данных*, ведь это всего лишь набор контекстов вычисления тех функций, которые к настоящему моменту начались, но не закончились.

В Scheme континуация представлена *первоклассным объектом*, то есть её можно, как и любое обычное значение, присваивать переменным, делать частью более сложных выражений, передавать в функции как параметр и возвращать из функций в качестве возвращаемого значения. При этом предусмотрена лишь одна операция над континуацией, вскрывающая её суть: **объект континуации можно «вызвать» как обычную функцию одного аргумента** или, если угодно, *применить* к одному параметру — точно так же, как можно применить к параметру функцию одного аргумента.

Этот аргумент континуации нуждается в дополнительных пояснениях. В любой момент всё оставшееся выполнение программы можно разделить на то, что произойдёт *до* возврата из текущей функции и то, что произойдёт *после* (собственно континуацию). Эти два периода вычислений связаны между собой *значением, которое вернёт функция*, то есть работа континуации начинается с того, что она получает (на вход!) значение, возвращённое породившей её функцией. Именно это значение и передаётся единственным параметром при вызове континуации как функции.

Итак, получается, что мы можем представить в виде некоторого объекта всю ту *последовательность действий, которые произойдут после возврата из текущей функции*. Надо отметить, что время жизни этого объекта никак не зависит от времени вычисления текущей функции, объект может быть доступен как в функциях, вызванных из текущей функции (то есть хронологически до момента её завершения), так и после завершения текущей функции. От самой текущей функции объекту континуации нужно только значение; но это значение, как мы уже поняли, не обязательно получать традиционным путём, то есть дождавшись момента завершения текущей функции: в любой момент континуацию можно применить к любому значению, полученному откуда угодно и каким угодно способом. Дальнейший ход вычислений при этом выглядит так, как если бы это (переданное континуации как параметр) значение было только что возвращено той функцией, чья континуация вызвана. Вызывая континуацию когда-то потом (*позже*, чем она была законсервирована в объекте), мы тем самым возвращаем программу в состояние, в котором она находилась в момент создания объекта континуации, и заставляем её начать сначала всю последовательность вычислений, составляющих континуацию — возможно, с другим значением, нежели то, что было возвращено в

эту континуацию функцией (если такое значение вообще существовало, что, как мы увидим из дальнейших примеров, совершенно не факт).

К нынешнему моменту у читателя в мыслях может образоваться полный бардак — и это, увы, нормально, поскольку континуации — материя для постижения весьма сложная, а мы пока не только не приводили примеров, но даже не рассказали, как вообще «законсервировать» континуацию. Попытаемся исправиться.

Единственный способ оформить континуацию в виде объекта — вызвать встроенную функцию, которая изначально называлась `call-with-current-continuation`; писать столь длинное название людям не хотелось, так что имя функции вскоре сократилось до лаконичного `call/cc`, хотя старое имя при желании тоже можно использовать. Функция `call/cc` получает один параметр, который сам должен быть функцией от одного аргумента; `call/cc` вызовет эту функцию (свой собственный параметр), передав ей параметром объект континуации.

Обычно параметр для `call/cc` получают вычислением лямбда-списка, то есть делают что-то вроде

```
(call/cc (lambda (cc) ... ))
```

Объект континуации оказывается связан с параметром `cc`, который доступен в теле лямбда-списка. Отдельный вопрос, континуацию *какой функции* `call/cc` передаст параметром в вызываемую им функцию, или, иначе говоря, действия *начиная с какого* войдут в эту континуацию. Ответ на этот вопрос довольно прост: это будет континуация *самого* `call/cc`, то есть она будет содержать все действия, начиная с самого первого *после* обращения к `call/cc`.

Из этого следует, что вызов континуации, «законсервированной» таким способом, в применении к произвольному параметру `x` сделает всё так, будто это значение `x` *было возвращено формой вызова* `call/cc`, то есть как будто бы это само обращение к `call/cc` вернуло значение.

Форма обращения к `call/cc` может вернуть значение и «более штатным» способом: если функция, переданная в `call/cc`, завершится и вернёт значение, то это значение станет возвращаемым значением и для самого `call/cc`; но во многих случаях до этого не доходит: программисты часто предпочитают возвращать управление изнутри `call/cc` путём вызова континуации. Например, выражение

```
(call/cc (lambda (cc) 25))
```

вернёт, естественно, число 25, поскольку это число станет результатом функции, полученной из лямбда-списка; выражение

```
(call/cc (lambda (cc) (cc 25)))
```

тоже вернёт 25, но в этом случае возврат происходит через обращение к континуации (применение объекта континуации к значению 25), ну а сама по себе функция, которая здесь получена из `lambda`, вообще не возвращает управления — она *прекращает* выполняться, когда внутри неё происходит обращение к континуации.

Несомненно, оба приведённых примера абсолютно бессмысленны на практике; мы привели их только для иллюстрации того, как работает `call/cc`.

Континуации при внимательном рассмотрении оказываются своего рода обобщением нескольких знакомых нам инструментов, связанных с передачей управления. С их помощью можно сделать аналог оператора `return`, знакомого нам по Си и Си++; можно организовать обработку исключительных ситуаций, весьма напоминающую то, что под этим термином понимается в Си++; в довершение всего континуации можно использовать для простой передачи управления вроде той, что делает столь нелюбимый программистами оператор `goto` (хотя, конечно, `goto` только передаёт управление, тогда как континуации меняют всё состояние вычисления программы). Но на самом деле возможности континуаций ещё шире — есть такие способы организации работы программы, которые могут быть сделаны только на континуациях.

Начнём с простого — покажем, как с помощью континуаций организовать досрочный выход из тела функции (и вообще любой управляющей конструкции). Тело функции или другой фрагмент программы, из которого хочется иметь возможность немедленно «выпрыгнуть», нужно «завернуть» в связку `call/cc` и `lambda` следующим образом:

```
(call/cc (lambda (return)
    ;
    ; произвольная последовательность форм
    ;
))
```

В любом месте оформленного таким образом фрагмента кода мы можем вставить вызов `(return` *val*`)` (в качестве *val* может выступать произвольное значение) — и это приведёт к немедленному завершению выполнения нашего фрагмента, после чего вся эта форма (имеющая `call/cc` на своём верхнем уровне) вернёт значение *val*. Конечно, имя `return` мы выбрали из соображений наглядности, оно тут ни на что не влияет — это просто переменная, с которой связывается значение континуации, и она могла бы называться как угодно иначе.

Пусть, например, у нас имеется некий (достаточно длинный) список и мы хотим отыскать в нём первый элемент, удовлетворяющий некоторому условию, оформленному в виде функции-предиката. Такой поиск можно оформить в виде функции двух аргументов, первый будет называться `wanted?` и задавать предикат, через второй (`lst`) будет передаваться список. Для просмотра списка воспользуемся встроенной функцией `for-each`, и если очередной элемент обратил наш предикат в истину, вернём этот элемент, воспользовавшись механизмом управления континуациями:

```
(define (search wanted? lst)
    (call/cc (lambda (return)
        (for-each
            (lambda (element)
                (if (wanted? element) (return element))
            )
            lst
        )
        #f    ; на случай, если так ничего и не нашлось
    ))
)
```

В отличие от оператора `return`, привычного нам по Си/Си++, *такой* возврат управления можно проделать не только непосредственно из фрагмента программы, который нужно досрочно завершить (как это сделано в нашем примере), но и из любой функции, которую этот фрагмент прямо или косвенно вызывает — лишь бы в ней оказался доступен нужный объект континуации. Пусть, к примеру, у нас *уже есть* функция, которая просматривает список и для всех элементов, удовлетворяющих определённому условию, вызывает заданную функцию (*callback*, см. §9.4.1):

```
(define (traverse lst wanted? callback)
    (for-each
        (lambda (x) (if (wanted? x) (callback x)))
        lst
    )
)
```

Задача поиска первого элемента, удовлетворяющего заданному условию — та, которую мы решали в предыдущем примере — благодаря механизму континуаций может быть решена с помощью этой функции, что для людей, незнакомых с континуациями, выглядит несколько неожиданно. Для этого нам потребуется «законсервировать» континуацию и передать её — вот прямо саму континуацию! — функции `traverse` в роли *callback*-функции:

```
(define (search2 wanted? lst)
    (call/cc (lambda (return)
        (traverse lst wanted? return)
    ))
)
```

Глядя на этот пример, легко догадаться, как с помощью континуаций организовать ***обработку исключений***. Все функции, которые по нашей задумке должны быть способны (прямо или косвенно) выбросить исключение, надо снабдить дополнительным параметром. Через этот параметр будет передаваться объект континуации с того уровня вложенности вызовов, где располагается обработчик исключений. Функции, способные бросать исключения, могут выглядеть примерно так:

```
; quad_eq.scm
(define (quad-eq a b c throw)
    (if (= a 0) (throw 'NotAQuadraticEq))
    (let ((discrim (- (* b b) (* 4 a c))))
        (if (< discrim 0) (throw 'NoRealRoots))
        (let
            (
                (mb2a (/ (- b) (* 2 a)))
                (dq2a (/ (sqrt discrim) (* 2 a)))
            )
            (list (- mb2a dq2a) (+ mb2a dq2a))
        )
    )
)
```

Слово `throw` мы, естественно, выбрали для большей наглядности — это (как и `return` в примерах, приведённых выше) просто параметр, через который передаётся континуация. Как видим, функция решает квадратное уравнение (в действительных числах), при этом в обоих хорошо известных особых случаях — если уравнение на самом деле не квадратное или если оно не имеет действительных корней — заставляет переданную параметром континуацию немедленно вернуть специальное значение (символ `NotAQuadraticEq` или `NoRealRoots`).

Вызвать функцию, оформленную таким образом, мы можем либо из такого места, где мы готовы обрабатывать особые случаи, либо из другой функции, которая оформлена так же, то есть имеет параметр, через который передаётся объект континуации-обработчика. Первый вариант иллюстрируется следующим диалогом с интерпретатором `csi`:

```
#;1> (load "quad_eq.scm")
; loading quad_eq.scm ...
#;2> (call/cc (lambda (cc) (quad-eq 1 -2 1 cc)))
(1.0 1.0)
#;3> (call/cc (lambda (cc) (quad-eq 1 1 -20 cc)))
(-5.0 4.0)
#;4> (call/cc (lambda (cc) (quad-eq 0 1 -20 cc)))
NotAQuadraticEq
#;5> (call/cc (lambda (cc) (quad-eq 1 1 10 cc)))
NoRealRoots
```

Здесь не происходит ничего из того, что мы обычно ожидаем от обработки исключений: конструкция с `call/cc` возвращает либо список корней, либо символ, который наша функция выбросила, а должным образом среагировать на специальные значения — забота того, кто эту конструкцию вызывает, т. е. работа с нашей функцией строится так же, как если бы она просто *возвращала* специальные значения как свои собственные, а не использовала для них континуацию. Намного интереснее будет, если где-то (например, в главной функции нашей программы) предусмотреть обработчик, оформленный примерно так:

```
(define (main)
    (let
        ((caught (call/cc (lambda (throw)
            ;
            ; здесь вызываем произвольные функции,
            ; имеющие параметр throw для исключений
            ;
        ))))
        (cond
            ((eq? caught 'ok) #t)
            ((eq? caught 'NotAQuadraticEq)
                (display "Malformed quadratic eq.\n")
            )
            ((eq? caught 'NoRealRoots)
                (display "Encountered quad. eq. without roots\n")
            )
            ;
            ; здесь обработчики других особых случаев
            ;
        )
    )
)
```

Тело лямбда-списка здесь играет роль try-блока, а альтернативы формы `cond` заменяют catch-блоки (см. §10.5.4). Единственное существенное отличие от Си++ состоит в том, что значение континуации (`throw`) нам придётся передавать в функции в явном виде. Но общая идея обработки исключений оказывается в полной мере реализована: мы можем из нашего аналога try-блока вызвать какую-нибудь функцию `f`, из неё — функцию `g`, из неё — функцию `h`, которая будет решать квадратное уравнение с помощью `quad-eq`, и обычным результатом — списком вычисленных корней — сможет воспользоваться сама, тогда как если возникнет одна из двух особых ситуаций (`quad-eq` выполнит одну из веток с `throw`, по смыслу аналогичных оператору `throw` из Си++), выполнение функций `h`, `g` и `f` (и самой `quad-eq`, естественно, тоже) немедленно прекратится, а управление получит соответствующий обработчик.

Пожалуй, теперь самое время вернуться к комментарию на стр. 61, где мы отмечали, что обработка исключений не имеет никакого отношения к объектно-ориентированному программированию. Можно надеяться, что теперь сомнений в этом у читателя не осталось.

Для контраста рассмотрим ещё один пример[19]; в отличие от всего рассмотренного выше, здесь обращение к континуации будет происходить уже после того, как «законсервировавший» её вызов `call/cc` завершился. Для начала определим функцию `right-now`, которая будет возвращать текущую континуацию:

```
(define (right-now) (call/cc (lambda (cc) (cc cc))))
```

Как видим, здесь из тела лямбда-списка сразу же производится обращение к полученной континуации, при этом она же сама служит параметром, так что форма, вызывающая `call/cc`, возвращает ту самую континуацию, которую представила в виде объекта. Название *right now* оправдывается тем, что функция как бы фиксирует (в виде объекта) текущий момент вычислений.

Вторая функция, которую мы опишем, будет называться `go-when`; она будет производить возврат к моменту, ранее зафиксированному с помощью `right-now`. Естественно, это приведёт к тому, что форма, содержащая вызов `right-now`, «вернёт» ещё раз; чтобы она всегда возвращала одно и то же значение — а именно всё тот же объект континуации — мы при обращении к континуации (которое как раз и вернёт нас обратно к ранее помеченному моменту) воспользуемся самой континуацией в качестве параметра:

```
(define (go-when then) (then then))
```

Теперь можно в любой момент вызвать `right-now`, запомнить возвращённое ею значение, а затем вызвать `go-when`, чтобы вернуться назад, туда-тогда, где вызывалась `right-now`. Например, вычисление конструкции

```
(let ((the-beginning (right-now)))
    (display "Hello, world\n")
    (go-when the-beginning)
)
```

приведёт к бесконечному циклу, печатающему сакраментальное `Hello, world`. Цикл, впрочем, не обязан быть бесконечным; конструкция

```
(let ((rest 5))
    (let ((the-beginning (right-now)))
        (display "Hello, world\n")
        (set! rest (- rest 1))
        (if (> rest 0) (go-when the-beginning))
    )
)
```

напечатает `Hello, world` ровно пять раз. Совершенно не факт, что так стоит делать на практике (фактически здесь в обоих примерах создаётся цикл с помощью безусловного перехода назад), но эти примеры позволяют составить первое впечатление о возможностях континуаций.

На деле континуации — инструмент намного более мощный, чем можно увидеть на простеньких примерах. Заинтересованному читателю порекомендуем статью Мэтта Майта «Continuations by example» [17], в которой разобраны примеры намного более сложные и красивые: автор статьи эмулирует на континуациях так называемые сопрограммы, организует поиск решения с возвратами из тупиковых версий, решает задачу обхода дерева с выдачей результатов по одному (что-то подобное мы делали в §9.2.1, но на континуациях это выглядит намного изящнее) и приводит много других интересных примеров.

[19] Пример взят из статьи [17]; автор — Мэтью Майт (*Mattew Might*).

11.2.5. (*) Continuation-passing style

Коль скоро в нашем распоряжении оказались continuations, реализованные с достаточной степенью эффективности — то есть так, чтобы мы могли применять их, не опасаясь чрезмерных потерь по памяти и времени выполнения — становится возможным качественно иной (в сравнении со всем, к чему мы привыкли до сих пор) подход к организации передачи управления между подпрограммами. Вместо того, чтобы *возвращать управление вызвавшему*, подпрограммы получают на вход лишний параметр — continuation, означающую, *что должно происходить после их завершения*, и, закончив свои вычисления, отдают управление через continuation, не задумываясь о том, *кому* управление будет передано.

Для иллюстрации приведём пример, заимствованный из английской Википедии[20]. В этом примере длина гипотенузы прямоугольного треугольника вычисляется по теореме Пифагора на основании заданных длин двух катетов. В обычных условиях для этого будет применена примерно такая функция:

```
(define (pyth x y)
    (sqrt (+ (* x x) (* y y)))
)
```

Чтобы переделать это решение в соответствии со стилем передачи continuations, для начала снабдим себя нужными примитивами; нам потребуются соответствующим образом оформленные функции умножения, сложения и извлечения квадратного корня, принимающие continuation дополнительным параметром. Написать такие функции несложно:

```
(define (+& x y k) (k (+ x y)))
(define (*& x y k) (k (* x y)))
(define (sqrt& x k) (k (sqrt x)))
```

(здесь и далее версии функций, оформленные в соответствии с continuation-passing style, обозначены символом амперсанда «&», а параметр, через который передаётся continuation, называется `k`). Теперь мы можем переписать функцию `pyth` следующим образом:

```
(define (pyth& x y k)
    (*& x x
        (lambda (x2) (*& y y
            (lambda (y2) (+& x2 y2
                (lambda (x2py2) (sqrt& x2py2 k))
            ))
        ))
    )
)
```

Несомненно, человек в трезвом уме и твёрдой памяти так писать не станет, это лишь иллюстрация концепции; для реального программирования в стиле передачи continuations нужна специальная поддержка со стороны используемого языка, которой, заметим, нигде нет; та же статья английской Википедии упоминает некую библиотеку для языка Хаскель, но на этом, похоже, всё.

Практическое применение этот стиль нашёл в области реализации некоторых языков программирования, и прежде всего — самого языка Scheme. Многие трансляторы Scheme, включая рассмотренный нами Chicken Scheme, переводят программу с языка Scheme на язык Си, но получающаяся программа организована довольно экзотически: функции в этой программе *никогда не возвращают управление* обычным способом, вместо этого они вызывают всё новые и новые функции, так что аппаратный стек только растёт; в нём располагаются все порождаемые функциями структуры данных, так что куча оказывается до поры до времени не нужна. Когда размер стека достигает некоторого заранее заданного предела, включается *сборщик мусора*, который переносит структуры данных, всё ещё доступные программе, из стека в кучу, а те, что уже не доступны (ушли в мусор), просто игнорирует; после завершения работы сборщика стек сбрасывается в исходное положение и снова начинает расти — до следующего включения сборщика. Подробно эта техника изложена в статье [18].

Заинтересованному читателю мы можем посоветовать статью Джона Рейнолдса (John C. Reynolds) «The Discoveries of Continuations» [19]. В этой статье приведён обзор истории возникновения continuations как таковых и стиля передачи continuations как подхода к организации передачи управления между частями программы. Полный текст статьи можно найти в Интернете с помощью поисковых сервисов.

11.3. Об оформлении кода на лиспоподобных языках

Конечно, при создании программ на Лиспе и Scheme применение структурных отступов становится насущной необходимостью, причём даже в большей степени, чем при работе на других языках — иначе, пожалуй, в этих скоплениях скобок разобраться было бы совершенно невозможно. Стиль, применявшийся нами в предшествующих параграфах, существенно отличается от того, который «исторически сложился» вокруг Лиспа и с которым вы, скорее всего, столкнётесь при изучении текстов других авторов. Надо сказать, что этот стиль вызывает некоторое недоумение. Рассмотрим для примера типичную функцию на Лиспе, написав её в «традиционном» для любителей Лиспа стиле:

```
(defun isomorphic (tree1 tree2)
       (cond ((atom tree1) (atom tree2))
             ((atom tree2) NIL)
             (t (and (isomorphic (car tree1) (car tree2))
                     (isomorphic (cdr tree1) (cdr tree2))))))
```

Обращают на себя внимание два важных момента. Во-первых, размер отступа явно «плавает»: форма `cond` сдвинута относительно объемлющей формы на семь пробелов, отдельные предложения этого `cond`'а сдвинуты уже на шесть пробелов, а второй аргумент `and` сдвинут на восемь пробелов правее предыдущего уровня отступа. Обусловлено это главенствующим подходом к форматированию списка: если список умещается в строку, то его записывают на одной строке, если же он не умещается, то на одной строке пишут его начало — первый элемент, который чаще представляет собой имя функции или формы, и второй элемент, то есть первый аргумент функции или формы; остальные аргументы располагают под первым в столбик. Если первый аргумент не поместился в строку, его разбивают аналогичным образом, причём его начало на следующую строку не переносят (именно так в приведённом примере получилось с вызовом `and` после символа `t`). В результате размер сдвига определяется длиной имени символа, стоящего в форме первым: слово `defun` вместе со скобкой и пробелом даёт семь символов (отсюда семь пробелов на первом уровне отступа), `cond` вместе со скобкой и пробелом даёт шесть символов — это размер второго уровня отступа, ну а `t`, `and`, две скобки и два пробела — это искомые восемь пробелов на третьем уровне.

Форма `defun` (или `define`, если речь идёт о Scheme) представляет собой одно из нескольких исключений из общего правила: здесь стараются на одной строке разместить само имя формы, затем имя функции и список параметров, а тело функции обычно сносят на следующую строку, даже если оно уместилось бы в одной строке с «заголовком».

Вторая характерная особенность, показанная в примере — это группа из шести закрывающих скобок в конце. Почему поклонники Лиспа предпочитают закрывать вложенные списки именно таким вот образом, не вполне понятно, ведь скобки приходится *считать*, они сливаются друг с другом, очевидно становясь источником труднообнаружимых ошибок. Тем не менее такой стиль настолько популярен, что существуют даже диалекты Лиспа, в которых есть специальный символ, обозначающий столько закрывающих скобок, сколько есть незакрытых списков; так, в диалекте muLISP можно было записать наш пример следующим образом:

```
(defun isomorphic (tree1 tree2)
       (cond ((atom tree1) (atom tree2))
             ((atom tree2) NIL)
             (t (and (isomorphic (car tree1) (car tree2))
                     (isomorphic (cdr tree1) (cdr tree2]
```

Возможность закрыть сразу все открытые скобки удобна лишь на первый взгляд: используя её, мы сами себя лишаем средства проверки корректности структуры нашего списка.

Между тем никто не мешает писать на Лиспе так, как это делали мы в предыдущих главах — «по-паскалевски», с постоянным размером отступа и расположением закрывающей скобки точно под началом соответствующей конструкции; нужно только разрывать список, не поместившийся в строку, не после *второго* элемента, а после первого. Функция `isomorphic` теперь будет выглядеть так:

```
(defun isomorphic (tree1 tree2)
    (cond
        ((atom tree1) (atom tree2))
        ((atom tree2) NIL)
        (t
            (and
                (isomorphic (car tree1) (car tree2))
```

[20] Статья «Continuation-passing style», `https://en.wikipedia.org/wiki/Continuation-passing_style`

```
                    (isomorphic (cdr tree1) (cdr tree2))
                )
            )
        )
    )
```

Конечно, такой код занимает больше строк, но это окупается ясностью его структуры. В частности, расположение нескольких идущих подряд закрывающих круглых скобок вдоль диагонали экрана показывает, что баланс скобок, скорее всего, соблюдён, тогда как нарушение такого расположения однозначно свидетельствует о том, что где-то мы со скобками запутались.

В большинстве случаев синтаксическая структура в Лиспе — это просто список, который начинается и заканчивается скобкой, так что открывающую и закрывающую круглые скобки приходится располагать точно друг над другом (конечно, если они не остались на одной строке). Из этого правила есть несколько исключений, связанных с сокращённой записью квотирования (апостроф), функционального квотирования (комбинация `#'`), с синтаксисом вектора (символ `#`), полуквотированием (*semiquotation*) и его составляющими, которые используются в макросах, и так далее. Во всех этих случаях перед открывающей скобкой оказывается один или два символа, задающих иной смысл всей конструкции; при этом получается, что открывающая скобка — *не первый* символ конструкции, но закрывающая — по-прежнему *последний* символ. Естественно, если такая конструкция оказалась разнесена на несколько строк, следует располагать закрывающую скобку точно под *началом конструкции*, а не под открывающей скобкой:

```
(mapcar
    #'(lambda (str elem)
        (list ('rec str elem 'endrec))
    )
    list
    cycled-labels
)
```

В этом примере также видно, что лямбда-список представляет собой ещё одно исключение из правил — его оформление напоминает оформление формы `defun`; источник такого исключения вполне понятен, ведь лямбда-список — это безымянная функция, а для функции нам привычнее видеть заголовок, в который включён список параметров.

Ещё один важный особый случай, заслуживающий внимания — это форма `let`, а также всевозможные `let*`, `letrec`, `funlet`, `macrolet` и т. п. Если список локальных переменных, вводимых с помощью `let`, умещается на одной строке с самим словом `let`, следует там его и оставить:

```
(let ((s 0) (p 1))
    (mapcar
        #'(lambda (x) (setq s (+ s x)) (setq p (* p x)))
        lst
    )
)
```

Ситуация резко осложняется, если первый элемент формы `let` приходится расположить на нескольких строках, а такое требуется очень часто — достаточно в одном из инициализаторов появиться сложному выражению. «Правильного» решения для этой ситуации просто нет, все существующие варианты так или иначе оказываются плохи. Программисты в таких случаях изобретают самые разнообразные конструкции, например:

```
(let ((seq (build-sequencer 0))
      (colset '(red orange yellow green blue violet))
      (ls nil))
   (setq ls (mapcar #'list seq colset the-list))
   (store ls)
   (reverse ls))
```

Недостатки такого форматирования (к сожалению, традиционного для Лиспа) мы уже отмечали выше. Чтобы в такой ситуации соблюсти стиль с фиксированным размером отступа, нужно действовать совершенно иначе:

```
(let
    (
        (seq (build-sequencer 0))
        (colset '(red orange yellow green blue violet))
        (ls nil)
    ) ;;
    (setq ls (mapcar #'list seq colset the-list))
    (store ls)
    (reverse ls)
)
```

Обратите внимание на символ комментария в конце списка связываний; применять его, конечно, не обязательно, но он позволяет яснее отделить заголовок формы `let` от её тела. В принципе, можно найти и другой допустимый стиль, например:

```
(let (
    (seq (build-sequencer 0))
    (colset '(red orange yellow green blue violet))
    (ls nil)
) ;;
    (setq ls (mapcar #'list seq colset the-list))
    (store ls)
    (reverse ls)
)
```

К такому стилю придётся некоторое время привыкать, очень уж непривычно выглядят две закрывающие скобки на одном уровне в условиях отсутствия открывающей скобки; но, как ни странно, это всё же лучше, чем показанный выше «традиционный» вариант, в котором не соблюдается постоянство размера отступа и закрывающие скобки скапливаются в конце конструкции, образуя нечто неудобоваримое.

11.4. Логическое и декларативное программирование

Понятия логического и декларативного программирования часто употребляются в связке, но означают в действительности совершенно разные вещи. Как уже отмечалось (см. стр. 14), логическая программа формулируется в виде набора правил, описывающих предметную область, а исполнение состоит в попытке доказать или опровергнуть некое утверждение; иначе говоря, описание работы программы (т. е. сама программа, ведь это то же самое) строится из конструкций, характерных для формальной логики — предикатов, правил, логических связок и т. п. Напротив, ***декларативное программирование*** исходно вообще не имеет прямого отношения к элементам формальной логики: термин «декларативное программирование» сам по себе означает, что в программе тем или иным способом задаются характеристики информационного объекта, который должен быть найден или построен, а то, как именно его будут искать или строить — это уже проблемы исполнителя.

Естественно, формальная логика — это один из наиболее очевидных кандидатов на роль языка для такого «описания свойств объекта», и это приводит к возникновению своеобразного симбиоза логического программирования с декларативным; больше того, два наиболее известных языка логического программирования — Пролог и Дэйталог — оба являются одновременно и логическими, и декларативными. С другой стороны, язык SQL, как мы отмечали ранее, является декларативным, но никоим образом не логическим. Автору не попадались языки, которые не были бы декларативными, при этом будучи логическими, но он не рискнёт утверждать, что такое невозможно.

В §9.1.3 (стр. 14–15) мы обсудили парадигмы логического и декларативного программирования и даже привели простенькие примеры кода на Прологе, чтобы создать некое общее впечатление, так что сейчас мы можем себе позволить перейти непосредственно к практике; читателю, возможно, имеет смысл освежить в памяти приводившийся ранее материал.

11.4.1. Язык Пролог и его история

Исторически первым языком собственно логического программирования стал язык Пролог, и он же по сей день остаётся наиболее популярным. Предложен этот язык был в начале 1970-х во Франции исследовательской группой под руководством Алана Колмероэ (*Alain Colmerauer*); первую известную реализацию Пролога Колмероэ выполнил вдвоём с Филиппом Русселем (*Philippe Roussel*) в 1972 году.

До этого логическое программирование применялось в языке Плэннер, который, судя по всему, сыграл определённую роль в формировании основных идей Пролога; но Плэннер был скорее попыткой сформировать «продвинутый» диалект Лиспа, возможности логического вывода играли в нём роль пусть и заметную, но не главную. Синтаксис, изначально основанный на логических обозначениях, впервые был введён именно в Прологе.

Отдельного замечания заслуживает внутреннее устройство пролог-решателя. В мире довольно популярен миф о том, что якобы в реализации Пролога как-то используется метод резолюций из математической логики; в действительности это, мягко говоря, не совсем так. Семантику Пролога вообще нельзя адекватно описать в терминах логики, поскольку вычисление *целей* (из которых состоят предложения Пролога и в конечном счёте вся программа на этом языке) может порождать побочные эффекты. Кроме того, Пролог допускает такие предикаты (точнее, запросы к ним), множество решений которых бесконечно; при этом бесконечный ряд решений рассматривается в каком-то определённом порядке, одно решение за другим, и этот порядок может оказаться таким, что до некоторых (несомненно правильных) решений дело не дойдёт *никогда* — например, если всё множество решений состоит из двух и более последовательностей, каждая из которых бесконечна сама по себе, а нужное решение находится в любой из этих последовательностей, кроме самой первой.

До некоторой степени верным будет утверждение, что *если рассматриваемое предложение пролог-программы не содержит обращений ни к каким «хитрым» предикатам* — ни к тем, которые выдают бесконечное множество решений, ни к тем, что имеют побочный эффект, ни к тем, которые управляют самим процессом вычислений, ни, наконец, к тем, которые рассматривают некоторые или все свои аргументы в качестве арифметических выражений и вычисляют их, прежде чем продолжить работу — то в этом случае поведение пролог-решателя соответствует частному случаю SLD-резолюции. В действительности это утверждение уводит нас в сторону от понимания реальности; реализуя Пролог, его авторы совершенно не думали ни про какие методы резолюций. Намного правильнее рассматривать пролог-решатель как машинку (пусть и весьма продвинутую) для перебора возможных вариантов. «Продвинутость» тут заключается в весьма нетривиальной логике работы с контекстами, то есть с теми структурами данных, которые отвечают внутри пролог-решателя за связь имён переменных с их локальными (а никаких других в Прологе нет) значениями.

Так или иначе, далеко не во всех случаях можно понять работу программы на Прологе, пользуясь только декларативными и логическими терминами; очень часто (на самом деле, возможно, намного чаще, чем хотелось бы) приходится отвечать на вопрос, *как* (какими своими действиями) пролог-решатель решает поставленную задачу: в какой последовательности он рассматривает цели в предложении и предложения в составе процедуры, в каком порядке — отдельные решения из множества возможных, в какой момент переменные связываются со своими значениями и в какой момент эти связи исчезают, и т. п. Всё это называется ***процедурной семантикой*** Пролога; её противопоставляют семантике ***декларативной***, или ***логической***. В разговоре о парадигмах, то есть о том, как человек может воспринимать выполнение программы, логическая семантика для нас намного интереснее, но, к сожалению, приходится признать один простой факт: в терминах процедурной семантики работу Пролога описать можно всегда, а вот в терминах логической/декларативной — только если повезёт.

Принципы устройства ранних версий пролог-решателя изложили сами его авторы во главе с Аланом Колмероэ в статье [20], которая опубликована на французском языке, но переведена в том числе и на русский. К сожалению, в русском переводе диаграммы используемых структур данных были воспроизведены довольно посредственно, так что по нему трудно понять, как всё в действительности было устроено.

В 1983 году Дэвид Уоррен разработал формальный автомат, который так и называют ***абстрактной машиной Уоррена***. Описание этого автомата, использующего стек, кучу и так называемую «тропинку» (*trail*), хранящую информацию для отката к предыдущим состояниям, Уоррен привёл в своём техническом сообщении [21]; его текст столь скуп на объяснения, что в мире нашлось не так много людей, способных понять, как это всё устроено и как оно работает. Между тем программы на Прологе в наши дни обычно переводятся как раз в инструкции для машины Уоррена или её вариаций — это позволяет исполнять их с бóльшей эффективностью.

В 1991 году Хасан Айт-Каси (*Hassan Aït-Kaci*) опубликовал книгу «Warren's Abstract Machine: A Tutorial Reconstruction» («Абстрактная машина Уоррена: учебная реконструкция») [22], которая, если верить её предисловию, должна была донести принципы устройства машины Уоррена до более широких кругов публики. Объём этой книги превышает объём сообщения Уоррена почти в пять раз, но проблемы, если совсем честно, так и не решает: понять происходящее оказывается всё ещё слишком сложно. На русский язык ни сообщение Уоррена, ни книга Айт-Каси не переводились, что весьма способствует дальнейшему мифотворчеству относительно внутреннего устройства пролог-решателя.

11.4.2. SWI-Prolog

На момент написания этой книги единственной в мире активно поддерживаемой и реально «живой» реализацией Пролога следует, по-видимому, считать SWI-Prolog; его интерпретатор входит в большинство дистрибутивов Linux, так что вы сможете его установить на свой компьютер без лишних сложностей. Скорее всего, нужный вам пакет будет называться `swi-prolog`; исполняемый файл интерпретатора обычно имеет имя `swipl`, хотя в некоторых дистрибутивах это имя отличается. Если вы этот интерпретатор просто запустите без указания параметров, он вывалит на ваш терминал несколько строк информации о своей версии и правовом статусе и войдёт в интерактивный режим, в котором пользователю предлагается вводить запросы. В качестве приглашения к вводу здесь печатается знак `?-`. Интерпретатор использует то ли GNU Readline, то ли какой-то её аналог, так что вам сразу же доступно и редактирование вводимых запросов, и их история. Составить первое впечатление вам помогут несколько простых запросов, основанных на встроенных возможностях интерпретатора. Например, давайте для начала спросим у интерпретатора, сколько будет дважды два:

```
?- X is 2 * 2.
X = 4.

?-
```

Здесь «решение» для нашего запроса возможно только одно, причём предикат `is` в принципе не допускает множественности решений, так что всё совсем просто: интерпретатор выдал нам это единственное решение и сразу же, не дожидаясь дальшейших указаний, напечатал приглашение к новому вводу.

Давайте теперь попробуем более сложный случай, когда решений (хотя бы потенциально) может быть несколько. Для начала отметим, что в Прологе имеются гетерогенные списки, очень похожие на лисповские, только записываются они в квадратных скобках, а не в круглых, и элементы разделяются не пробелами, а запятыми. Предикат `member` встроен в Пролог и задаёт отношение, в котором находится любой элемент списка с самим списком, т. е. этот предикат «возвращает истину»[21], если первый его аргумент является элементом списка, заданного вторым аргументом. Давайте попробуем этот предикат в деле. Начнём с простого случая, когда заданные нами аргументы не находятся в том отношении, которое предполагает предикат:

```
?- member(100, [1, 2, 3]).
false.

?-
```

Здесь тоже всё просто, Пролог сказал нам, что мы неправы, и на этом успокоился. А вот если мы окажемся *правы*, всё будет выглядеть несколько иначе:

```
?- member(2, [1, 2, 3]).
true
```

[21] Как мы убедимся далее, о предикатах Пролога не следует думать как о «функциях, возвращающих истину или ложь» — формально это правильно, но уводит нас от надлежащего стиля мышления.

и... и всё, интерпретатор как будто бы завис. На самом деле, конечно, он вовсе не висит, он просто ждёт нашей реакции. Дело здесь в том, что *потенциально* запросы, основанные на предикате `member`, могут иметь больше одного решения, так что коль скоро одно решение найдено, то, возможно, стоит проверить, не найдётся ли ещё каких-нибудь решений; и сейчас Пролог как раз спрашивает нас, хотим ли мы это проверять. Вариантов у нас два: ввести символ точки («хватит, больше не надо») или же точки с запятой («хотим других решений»). В зависимости от того, какую кнопку мы нажмём, дальнейшее будет выглядеть по-разному:

```
?- member(1, [1, 2, 3]).
true ;
false.

?-
```

```
?- member(1, [1, 2, 3]).
true .

?-
```

Конечно, никакого «другого решения» интерпретатор не нашёл, но после ввода точки он сразу же успокоился и выдал нам очередное приглашение, тогда как после нажатия точки с запятой он сначала был вынужден довести до нашего сведения, что требуемых нами ответов на наш запрос, отличных от уже выданных, не имеется.

Чтобы понять, к чему всё это, давайте попросим интерпретатор *перечислить* элементы списка `[1, 2, 3]`:

```
?- member(X, [1, 2, 3]).
X = 1 ;
X = 2 ;
X = 3.

?-
```

Здесь каждый раз, когда мы вводили точку с запятой, требуя ещё решений, Пролог возвращался на шаг назад внутри реализации предиката `member` и на «развилке» пробовал пойти по новому пути, и так до тех пор, пока решения не иссякли.

Примечательно, что, выдав последнее решение, он завершил обработку нашего запроса, не дожидаясь никакой реакции. Это обусловлено реализацией предиката `member`; например, запрос `member(3, [1, 2, 3])` тоже будет обработан без каких-либо задержек, а наши вышеописанные сложности возникли лишь потому, что на момент нахождения заданного элемента (в нашем примере — элемента 2) список ещё не исчерпан.

Выйти из интерпретатора можно двумя способами: «запросив» встроенный предикат `halt` (не забудьте про точку, означающую конец запроса!) или устроив хорошо знакомый нам «конец файла», т. е. нажав `Ctrl-D`.

Среди довольно большого количества опций командной строки интерпретатора `swipl` следует сразу же выделить ключ `-s`. Пока вы пишете фрагменты вашей будущей программы или просто отдельные предикаты, этот ключ сэкономит вам изрядное количество времени. Дело тут в том, что, чтобы попробовать в деле свеженаписанный код на Прологе, его нужно *загрузить* в интерпретатор. Можно сделать это изнутри с помощью встроенного предиката `consult`, например, так:

```
?- consult('myprog.pl').
% myprog.pl compiled 0.00 sec, 11 clauses
true.

?-
```

Немного сэкономить время позволяет сокращённая запись обращения к предикату `consult` с помощью квадратных скобок:

```
?- ['myprog.pl'].
```

но — хотите верьте, хотите проверьте — намного быстрее будет указать имя файла в командной строке при запуске интерпретатора:

```
swipl -s myprog.pl
```

При этом интерпретатор запустится, как обычно, в интерактивном режиме, но уже будет «знать» все ваши предикаты, описанные в указанном файле (в нашем примере `myprog.pl`), и вы сможете сразу же использовать их в своих запросах.

Коль скоро в нашем обсуждении появились имена файлов, содержащих программы на Прологе, отметим, что авторы SWI-Пролога считают «правильным» для таких файлов суффикс (или, если угодно, «расширение») `.pl`. Этот выбор трудно назвать удачным, во всяком случае, в современных условиях, поскольку такой же суффикс привыкли использовать для своих скриптов любители языка Перл (Perl), а в повседневной жизни скрипты на Перле встречаются намного чаще, чем программы на Прологе.

Автор этих строк всегда предпочитал для Пролога суффикс `.plg`, пока в ходе экспериментов со SWI-Прологом не убедился, что в некоторых (пусть и довольно редких, но неприятных) случаях интерпретатор ожидает именно `.pl` и ни о чём другом слышать не желает — вплоть до того, что к имени вроде `program.plg` самовольно добавляет `.pl`, так что получается `program.plg.pl`, причём делает это совершенно втихаря, не выдавая никаких сообщений — в том числе и о неизбежно возникающей ошибке; что́ в действительности происходит, удалось понять лишь с помощью утилиты `strace`, о которой рассказано в приложении ко второму тому.

Здесь и далее в тексте нашей книги все имена файлов с кодом на Прологе будут заканчиваться на `.pl`; автор вынужден пойти на это, хотя и против собственной воли.

Самый простой способ сделать на Прологе полноценную программу — это, как водится, оформить её в виде скрипта. Для этого нужно будет начать ваш файл с «магической» строки `#!/usr/bin/swipl` (или где там ещё в вашей системе расположен интерпретатор SWI-Пролога); если этим и ограничиться, результат вас может несколько удивить: выполнив всё, что содержится в вашем файле, интерпретатор, вместо того чтобы успокоиться, войдёт в интерактивный режим.

Интерпретатор принимает на вход целый ряд опций, которые вроде бы должны помочь решить эту проблему, вот только тут есть одна сложность: как мы знаем, в файле скрипта интерпретатору можно передать лишь один аргумент (см. стр. 110), и как раз ничего подходящего для этой цели авторы SWI-Пролога не предусмотрели. Остаётся довольно странное решение: сделав всё, что хотели, принудительно завершить выполнение с помощью уже упоминавшегося `halt`. Например, сакраментальное «Hello, world» можно реализовать так:

```
#!/usr/bin/swipl
:- write('Hello, world!'), nl, halt.
```

или так:

```
#!/usr/bin/swipl
:- write('Hello, world!\n'), halt.
```

В обоих вариантах скрипт, *скорее всего*, будет работать так, как вы от него ожидаете — но, как ни странно, с уверенностью этого сказать нельзя. В *некоторых* (автор не возьмётся сказать, в каких конкретно) случаях `swipl` при запуске скрипта зачем-то выдаёт «информационное сообщение» о том, что файл был «откомпилирован» (читай — переведён во внутреннее представление) за такое-то время. Это можно рассматривать как повод потратить наш единственный аргумент командной строки, указав ключ «`-q`» (*quiet*), запрещающий выдачу лишних сообщений. Первая строка скрипта тогда будет выглядеть так:

```
#!/usr/bin/swipl -q
```

Интересно, что сами авторы SWI-Пролога рекомендуют вызывать интерпретатор из скрипта через стандартную программу `/usr/bin/env` вот так:

```
#!/usr/bin/env swipl
```

У этого подхода есть одно несомненное достоинство: он не привязан к местоположению `swipl` в вашей системе, а это местоположение может отличаться от `/usr/bin` — интерпретатор может оказаться в директории `/usr/local/bin`, `/usr/swipl/bin`, `/opt/swipl` и т. п., вариантов масса. Вот только флаг `-q` при этом указывать будет уже негде. Остаётся риторический вопрос, о чём думают люди, реализовавшие сложнейшую программу (а интерпретатор Пролога несомненно относится именно к таким) и при этом не давшие себе труда подумать, как этой программой потом пользоваться — а выдача непрошеного «информационного сообщения» однозначно об этом свидетельствует.

Как водится, всё это отнюдь не конец истории, а лишь её начало. Авторы SWI-Пролога указывают, что «правильнее» (судя по всему, с точки зрения переносимости, хотя переносить прологовские программы в сегодняшних условиях толком некуда) выделить в программе главную процедуру — её, не мудрствуя лукаво, обычно называют хорошо знакомым нам словом `main` — и указать интерпретатору на её наличие с помощью *директивы* `initialization` примерно так:

```
:- initialization(main).
```

Здесь есть два варианта. Вы можете оформить свою процедуру `main` как не имеющую параметров (т. е. написать процедуру `main/0`) либо как имеющую один параметр, который обычно называют `Argv`; на этот случай в документации говорится, что библиотека SWI-Пролога сама предоставляет процедуру `main/0`, которая уже вызывает процедуру `main/1`, передав ей единственным параметром список аргументов командной строки. Тут, опять же, есть один, как говорят, нюанс: зачем-то тот элемент, который мы привыкли называть `argv[0]` — т. е. имя, по которому запущена программа — из списка при этом удаляют (зачем? спросите у них). Воспользовавшись тем, что процедура `write` умеет печатать сложные термы, в том числе списки, мы можем опробовать сказанное в деле. Программа из трёх строк

```
#!/usr/bin/swipl -q
:- initialization(main).
main(Argv) :- write(Argv), nl.
```

— будучи запущенной с параметрами командной строки, напечатает их список:

```
avst@host:~$ ./echo_simple.pl abra schwabra kadabra
[abra,schwabra,kadabra]
```

Мы воспользовались здесь возможностью печати списка, чтобы не загромождать пример рекурсивной процедурой, в которой многое окажется непонятно — ведь изучение практически всего Пролога у нас ещё впереди. *Никогда так не делайте в реальных программах: ваш выбор инструмента (в данном случае языка Пролог) не должен влиять на то, что видит пользователь*, а в данном случае программа безосновательно навязывает пользователю прологовский синтаксис представления списков.

Есть ещё один способ доступа к аргументам командной строки: встроенная процедура `current_prolog_flag`, если первым из двух параметров ей передать атом `argv`, свяжет со своим вторым параметром всё тот же же список:

```
#!/usr/bin/swipl -q
:- initialization(main).
main :- current_prolog_flag(argv, Argv), write(Argv), nl, halt.
```

Внимательный читатель может обратить внимание, что в предыдущем примере мы не вызывали `halt` явно; дело в том, что там это за нас сделала библиотечная версия `main/0`, тогда как здесь мы сами написали `main/0`. Впрочем, добыть имя скрипта нам это не поможет[22] — из привычного `argv` SWI-Пролог по-прежнему будет убирать «нулевой» элемент.

Происходящее резко усложняется, если вам придёт в голову написать на Прологе программу, читающую что-то с клавиатуры. Здесь вы можете столкнуться сразу с двумя совершенно несуразными и крайне неприятными особенностями поведения интерпретатора. Во-первых, он за каким-то дьяволом упорно печатает приглашение к вводу (символы `|:`), хотя об этом его никто не просит. Во-вторых, программу оказывается невозможно прикончить привычной нам комбинацией `Ctrl-C` — вместо того, чтобы просто завершиться, интерпретатор «услужливо»[23] переходит в режим отладки, из которого даже программист не сразу выпутается, а о конечном пользователе и говорить нечего.

С навязчивым «приглашением» справиться оказывается довольно просто: нужно объяснить упрямому интерпретатору, что строка приглашения отныне пустая; это делается вычислением цели `prompt(_, '')`. С неработающим (точнее, работающим совсем не так, как надо) `Ctrl-C` чуть сложнее, придётся разобраться, что же происходит. Оказывается, интерпретатор SWI-Пролога из каких-то своих соображений перехватывает целый ряд сигналов[24], включая и тот, что генерируется по нажатию `Ctrl-C` — `SIGINT`. Отменить это можно параметрами командной строки, вот только из заголовочной строки скрипта мы их, увы, передать не можем. Остаётся разве что отменить уже состоявшийся перехват сигнала, вычислив цель `on_signal(int, _, default)`. Таким образом, если ваша программа должна что-то читать или вообще выполняться дольше, чем в течение нескольких сотых долей секунды, следует, по-видимому, начать её так:

```
main :- on_signal(int, _, default), prompt(_, ''), ...
```

и уже дальше написать свой текст. Между прочим, `on_signal` может почему-то не работать, если вы воспользуетесь вариантом, при котором ваш `main` имеет аргумент — во всяком случае, с версией SWI-Пролога 6.6.6 всё вышло именно так. Программа, начинавшаяся со строк

```
main(Argv) :-
    on_signal(int, _, default),
```

работать не хотела ни в какую, выдавая невнятную диагностику, но всё заработало, когда вместо этого было написано

```
main :-
    on_signal(int, _, default),
    current_prolog_flag(argv, Argv),
```

Теоретически `swipl` позволяет откомпилировать программу на Прологе, получив исполняемый файл. Найти информацию об этом удалось с большим трудом, причём сначала инструкция была обнаружена на каком-то форуме в Интернете, и лишь потом, уже точно зная, какие флаги нужно искать, ваш покорный слуга смог наконец найти нужное место в официальной документации. Здесь вас ожидает целый ряд сюрпризов.

Во-первых, это как раз тот случай, когда суффикс («расширение») исходного файла оказывается архиважным: если указать файл с суффиксом, отличным от `.pl`, `swipl` «ошибётся» (попросту не сделав ничего полезного или хотя бы осмысленного), но при этом *ни слова вам не скажет*.

Во-вторых, забудьте всё то, что говорилось про директиву `initialization` и библиотечную версию `main/0`; судя по всему, режим компиляции с ними несовместим. Вместо этого нужно написать свою процедуру `main/0`, а её имя указать горе-компилятору в командной строке как значение флага `--goal`.

Добраться до параметров командной строки теперь можно будет только через `current_prolog_flag(argv, Argv)`, как показано выше. И — о чудо — в этот раз с переменной `Argv` окажется связан список *всех* параметров, включая нулевой (имя программы). Кстати, не забудьте убрать из начала вашего исходника строку, превращавшую его в скрипт (ту, что начинается с `#!`) — она будет мешаться, а написать программу так, чтобы она могла служить и скриптом, и исходником для компиляции, автору не удалось (но, возможно, удастся вам; утверждать, что это совсем невозможно, оснований вроде бы нет).

Полностью командная строка для компиляции будет выглядеть так:

```
swipl -nodebug -g true -O --goal=main --stand_alone=true \
        -o myprog -c myprog.pl
```

Здесь предполагается, что файл с вашей программой называется `myprog.pl`, исполняемый файл вы хотите видеть под именем `myprog`. Основную «магическую» нагрузку несёт опция `--stand_alone=true`, это именно она указывает компилятору, что мы хотим получить исполняемый файл (да, *этому* компилятору надо об этом говорить). Флаг `--goal=main` указывает, как у вас в программе называется та процедура, которая выступает в роли точки входа, или, точнее говоря, с какой цели нужно начать выполнение программы — то есть здесь можно указать сложный терм с параметрами, а не просто имя. Загадочное `-g true` отключает печать текста о версии SWI-Пролога, попросту заменив «инициализационную» цель на вычисление лаконичного `true`. О назначении оставшихся двух ключей предложим читателю догадаться самостоятельно. Кстати, запретить перехват сигналов из командной строки компилятора вашему покорному слуге не удалось, хотя такая возможность в документации декларируется — так что не забудьте выполнить в начале своей программы приведённый выше запрос к `on_signal`.

Автор вынужден признать, что к концу рассказа о том, как же всё-таки победить `swipl`, у него сложилось чёткое ощущение, что создатели SWI-Пролога попросту издеваются.

11.4.3. Модель данных в Прологе

Как и в Лиспе, в Прологе используются гетерогенные динамические структуры данных, причём по своей сути они очень похожи на лисповские S-выражения. Любое корректное выражение Пролога называется ***термом***. Как и в Лиспе, в Прологе имеется несколько типов атомарных данных, причём, надо сказать, ассортимент этих типов может показаться очень бедным, особенно в сравнении с Лиспом; зато, как мы увидим позже, средства создания сложных структур данных на базе атомарных в Прологе несколько более «развесистые».

К атомарным объектам в Прологе относятся, во-первых, числовые константы — целые и с плавающей точкой. Здесь стоит отметить, что интер-

[22] Автор вынужден предположить, что это поведение может быть исправлено в будущих версиях SWI-Пролога, так что на момент прочтения книги читатель вполне может описанной проблемы не обнаружить.

[23] Здесь вспоминается притча, ставшая основой выражения «медвежья услуга».

[24] Механизм сигналов в Unix мы рассматривали во втором томе, см. §5.3.14.

претатор `swipl` в большинстве случаев компилируется с поддержкой «длинных» целых чисел, что означает отсутствие ограничения на их разрядность, но несколько снижает эффективность целочисленной арифметики; впрочем, предусмотрена возможность компиляции без этой поддержки, в этом случае целые будут 64-битными. Что касается «плавучих» чисел, то в документации декларируется использование формата, который на данной аппаратной платформе соответствует типу `double` языка Си (по-видимому, используется просто тип `double` того компилятора, которым была откомпилирована часть интерпретатора, написанная на Си).

Второй фундаментальный атомарный тип в Прологе — так называемые ***атомы***; этим термином обозначается сущность, очень похожая на лисповские *символы* (см. стр. 112). Атом в своём имени может содержать практически что угодно, только для этого его надо заключить в одиночные апострофы (да, строка `'Hello, world\n'` в одном из примеров выше — именно атом, как это ни печально). Если имя атома состоит только из букв, цифр, знака подчёркивания и при этом начинается со строчной («маленькой») буквы, апострофы можно опустить. Например, с точки зрения Пролога «`world`» и «`'world'`» — два варианта записи одного и того же, а вот если нам потребуется атом `'Hello'`, то опустить апострофы мы не сможем, поскольку слово здесь начинается с заглавной буквы (так в Прологе обозначаются переменные, об этом мы поговорим ниже). Есть и ещё один вариант атомов, которые можно записывать без заключения в апострофы: это любая последовательность из символов «`@`», «`#`», «`$`», «`^`», «`&`», «`*`», «`+`», «`-`», «`=`», «`<`», «`>`», «`.`», «`?`», «`/`», «`\`», «`:`» и «`~`»; возможно, есть и ещё какие-то символы, допустимые в именах атомов такого рода. В то же время символы «`!`», «`,`», «`%`», «`|`», а также круглые, квадратные и фигурные скобки, кавычки, апострофы в таких атомах использовать нельзя; возможность использования обратного апострофа («`‘`») зависит от версии интерпретатора и его настроек. Обычно такие комбинации символов используют для обозначения всевозможных математических, логических и чёрт знает ещё каких операций. Отметим заодно, что символ «\» внутри апострофов имеет специальное значение, примерно так же, как в Си.

Атомы в Прологе играют, во-первых, знакомую нам по лисповским символам роль «объектов, которые не равны ничему, кроме самих себя»; кроме того, атомы используются в качестве своеобразных конструкторов сложных данных; ещё с атомами могут связываться ***предикаты***, называемые также ***пролог-процедурами***. В довершение, будто этого мало, в традиционных версиях Пролога атомы используются ещё и в роли строковых констант, как это было сделано в нашем примере «Hello, world».

Несмотря на это, атом — лишь частный случай атомарного терма; использование термина «атом» в Прологе порождает постоянную путаницу, и чтобы не стать очередной жертвой этой путаницы, следует уяснить: атом в Прологе — это именно идентификатор, роль которого — быть уникальным, тогда как атомарными термами являются, наряду с атомами, ещё и другие «неделимые» объекты, в том числе числа. Ещё один атомарный объект (но не атом) — это знак `[]`, обозначающий пустой список.

Помимо перечисленного, SWI-Пролог вводит *строки* (*strings*), которые мы подробно обсудим чуть позже, и так называемые *блобы* (*blobs*), предназначенные для хранения произвольных двоичных данных при взаимодействии с подсистемами, реализованными на других языках (обычно на Си); блобы мы рассматривать не будем. Больше атомарных типов данных в Прологе нет.

Для конструирования более сложных структур данных из более простых в Прологе применяются так называемые ***сложные термы*** (англ. *compound terms*); такой терм состоит из атома, называемого ***главным функтором*** терма, и списка ***аргументов***. По своему смыслу сложные термы соответствуют хорошо знакомым нам *записям*, *структурам* и прочим подобным сущностям, при этом главный функтор соответствует «типу» записи, а аргументы — значениям полей; например, можно было бы использовать для записи даты терм `date(2019, may, 31)`, для представления сведений о человеке — что-нибудь вроде `person('John Doe', m, 1978)`, а географические координаты упаковать в какое-нибудь `place(n, 54.83843, e, 37.59556)`. В роли главных функторов у нас здесь выступают атомы `date`, `person` и `place`, первые два терма имеют по три аргумента, последний — четыре. Кстати, количество аргументов сложного терма называется его ***арностью*** (англ. *arity*); если это слово кажется вам странным, припомните термины «унарный», «бинарный» и «тернарный» — можно надеяться, что они для вас выглядят уже вполне привычно.

В роли каждого из аргументов может выступать произвольный терм, как атомарный, так и сложный. Например, мы могли бы в какой-нибудь программе, связанной с геометрией, обозначить точку на плоскости через её координаты, применив в роли функтора слово `point`: `point(0, 0)`, `point(17.3, -213.27)` и т. п.; окружность теперь можно задать, имея точку и радиус — получится что-то вроде `circle(point(2, 2), 1)`. В этом терме собственно ***функторов*** два — `point` и `circle`, но только `circle` считается *главным* (хотя если рассмотреть отдельно первый аргумент, то есть терм `point(2, 2)`, то в нём `point` уже будет, естественно, главным функтором).

Возвращаясь к аналогии с записями и структурами, отметим, что один и тот же функтор можно использовать для построения сложных термов различной арности и — продолжая нашу аналогию — термы разной арности, пусть и сконструированные одним и тем же функтором, будут, конечно, соответствовать записям *разных* типов. Для обозначения всего множества возможных термов заданной арности, образованных конкретным атомом, обычно пишут этот атом, а через слэш от него — арность, что-то вроде `point/2`, `date/3`, `place/4` и т. п.

Отметим ещё один казус. Арность вполне может быть *нулевой*, то есть Пролог допускает термы, не содержащие ни одного аргумента. Круглые скобки для такого терма не пишутся, так что по внешнему виду «0-арный» терм ничем не отличается от того атома, который использован в роли его главного функтора. Не различаются они и по своему смыслу; можно сказать, что вообще любой атом может в Прологе рассматриваться как вырожденный (0-арный) случай «сложного» терма. Подчеркнём, что это верно для атомов, но отнюдь не для всех *атомарных термов*; например, числовые константы выступать в роли функторов не могут, так что в их отношении ни о каком «сложном 0-арном терме» речи не идёт.

Пожалуй, самое время предостеречь начинающих пролог-программистов от одной довольно распространённой среди новичков ошибки. **Между именем атома, выступающего в роли функтора, и открывающей скобкой, с которой начинается список его аргументов, *нельзя* ставить пробелы!** Если такой пробел вы всё же поставите, интерпретатор попытается проанализировать ваш атом-функтор и ваш список в круглых скобках как два отдельных терма; скорее всего, у него ничего не выйдет, но если и выйдет, то уж точно не то, чего вы хотели.

В качестве функторов могут выступать не только «правильные» идентификаторы, как в наших примерах; вполне можно написать что-нибудь вроде

```
'Yes, this is a functor, too'(1, 2, 3)
```

и Пролог это поймёт, но вот надо ли такое писать — вопрос отдельный; пожалуй, всё-таки не стоит. Несколько иначе обстоят дела с атомами, чьи имена состоят из спецсимволов: их как раз применяют в роли функторов довольно активно. Дело в том, что унарные и бинарные функторы Пролог позволяет объявить *операторами*[25], так что будет можно, например, вместо `a(b, c)` написать `b a c` (что вряд ли осмысленно), или вместо `loves(john, mary)` написать `john loves mary`, что уже хотя бы понятно. Такую запись сделает корректной с точки зрения Пролога вычисление встроенной цели `op`:

```
?- op(1000, xfx, loves).
```

Проверить новую специальную роль атома `loves` можно так:

```
?- X = john loves mary.
X = john loves mary.

?- X = loves(john, mary).
X = john loves mary.
```

Стоит обратить внимание, что интерпретатор теперь сам «предпочитает» именно такую («инфиксную») форму терма с атомом `loves` в роли главного функтора.

Подробный рассказ об операторах мы опустим; отметим лишь, что в нашем примере 1000 — это приоритет оператора, который тем выше, чем число меньше, а `xfx` — своеобразная «схема применения», означающая бинарную операцию, не допускающую в выражении (без скобок) операций того же приоритета; инфиксные операторы, для

[25] В оригинале здесь используется слово *operator*, с которым у нас начались проблемы ещё при изучении Си++; к счастью, в Прологе нет тех императивных «операторов», которые по-английски называются *statement*, а слово «операция» тут выглядело бы несколько неуместно.

которых указана схема `yfx`, применяются слева направо, схема `xfy` указывает на применение справа налево (как, например, операция присваивания в языке Си), `fx` и `fy` задают обычные (префиксные) унарные операции, `xf` и `yf` — постфиксные. Подробности читатель найдёт в любом учебнике по Прологу, например, в книге И. Братко [23].

Стоит заметить, что, например, арифметические выражения строятся в Прологе из тех же сложных термов, а функторами в них выступают знаки операций. Например, мы начали знакомство с интерпретатором `swipl` с вопроса «сколько будет дважды два», записав его в виде `X is 2 * 2`. С таким же успехом мы могли бы написать и `X is *(2, 2)`, и `is(X, *(2, 2))` — всё это корректные способы записи одного и того же терма, в котором главным функтором выступает слово `is`, а второй аргумент строится с помощью функтора `*`. Теперь мы, пожалуй, можем догадаться, что вообще любой запрос интерпретатору Пролога представляет собой не что иное, как терм, после которого стоит точка.

Забегая вперёд, заметим, что так выглядят не только *запросы*, но и *предложения* (*clauses*), из которых, собственно, и состоит программа на Прологе. Это осознают почему-то далеко не все, кто на Прологе программирует: многие продолжают верить в некую «особую» синтаксическую роль символа `:-`, а также запятой, которую ставят между целями в предложении, и точки с запятой, которая связывает между собой несколько возможных вариантов вычисления. В действительности всё это не более чем функторы, для которых «операторная» сущность определена самим интерпретатором (можно сказать, «встроенные операторы»).

У запятой, впрочем, особая синтаксическая роль всё же присутствует, ведь именно она разделяет, например, элементы в списке аргументов терма или, как мы сейчас увидим, элементы списка в краткой записи списка. Есть особая роль и у точки — она завершает предложения, что не мешает использовать её же в роли функтора. Этим — использованием точки в роли функтора — мы сейчас и займёмся.

Выше мы уже несколько раз упоминали наличие в Прологе *списков*. По-видимому, авторы Пролога хорошо знали Лисп и умели на нём программировать; об этом свидетельствует тот факт, что **списки в Прологе состоят из *точечных пар*** — точно так же, как в Лиспе, с поправкой на синтаксис. Точечные пары в Прологе — это сложные термы, образованные главным функтором «`.`» (точка) и имеющие два аргумента, то есть бинарные. Примерами таких термов могут служить `.(1, 2)`, `.(john, george)` и т. п. Надо сказать, что имеющийся в Прологе специальный синтаксис для списков работает и для точечных пар — приведённые примеры можно записать иначе: `[1|2]` и `[john|george]`. Сам интерпретатор выдаёт их именно в таком виде. Дальнейший фокус нам уже известен по Лиспу: для точечной пары, в правой части которой стоит точечная пара (и так сколько угодно раз) имеется специальная форма записи. Например, комбинация `[1|[2|[3|4]]]` записывается короче: `[1, 2, 3|4]`. Если припомнить, что в роли пустого списка выступает атомарная константа `[]`, можно догадаться, что, если поставить именно её в правую часть последней пары списка, мы получим «правильный список»: выражение `[1|[2|[3|[]]]]` эквивалентно выражению `[1, 2, 3]`. Списки, в которых последний элемент отделён знаком `|`, аналогичны *точечным спискам* Лиспа.

Отметим, что в SWI-Прологе, начиная с седьмой версии, точку волюнтаристски лишили роли атома — конструктора точечных пар; впрочем, так на Прологе всё равно почти никогда не писали. Вышеприведённый рассказ о точечных парах призван показать сходство между прологовской и лисповской моделями данных. Так или иначе, ещё в шестой версии SWI-Пролога точка в этой роли прекрасно работала; ну а начиная с седьмой версии в этой роли используется функтор `'[|]'/2`, про который создатели SWI-Пролога гордо заявляют, что он выглядит уродливо (*ugly*) и это так и задумано, тогда как «хорошо выглядящую» точку теперь можно использовать в другой роли. Атом `'[|]'` действительно смотрится так себе, особенно с учётом того, что записывать его приходится в апострофах. Например, список `[1, 2]` в свежих версиях SWI-Пролога можно записать так: `'[|]'(1, '[|]'(2, []))`.

Простим создателям SWI-Пролога этот небольшой идиотизм, тем более что от смены обозначений суть, как легко догадаться, не меняется: точечная пара не перестаёт быть таковой, несмотря на то, что имя её главного функтора теперь вовсе не точка, а странное `'[|]'`.

Как и в Лиспе, списки в Прологе гетерогенны и допускают произвольное количество уровней вложенности. При известной сноровке можно получить и «закольцованный» список.

С обыкновенными строками, то есть фрагментами текстовых данных, в Прологе ситуация сложилась довольно странная. В ранних диалектах Пролога, как мы уже отмечали, в этой роли использовались атомы; уместно будет вспомнить, что в Лиспе когда-то дела обстояли так же (см. §11.1.12), но если в Лиспе строковые константы как отдельная сущность появились довольно быстро, то в культуру Пролога им почему-то пришлось проникать намного дольше. Как мы знаем, одиночные кавычки (апострофы) здесь используются для формирования атомов. До сравнительно недавних пор большинство диалектов Пролога хотя и поддерживало ещё строковые литералы, записываемые в двойных кавычках, но обозначались этими литералами не атомарные выражения особого (строкового) типа, как можно было бы ожидать, а *списки*, притом даже не из букв, а из *чисел, соответствующих кодам букв*; так, выражения `"Hello"` и `[72, 101, 108, 108, 111]` в соответствии с этим правилом — просто синонимы.

Начиная с версий 3.*, SWI-Пролог ввёл отдельную сущность (атомарный тип) для работы со строками, но, как ни странно, вплоть до версий 6.* для этой сущности по умолчанию не было обозначения, то есть нельзя было в программе задать константу этого типа. Чтобы работать с «новыми» строками, нужно было изменить настройки интерпретатора, вычислив псевдоцель `set_prolog_flag(double_quotes, string)` и приказав тем самым интерпретатору воспринимать литералы в двойных кавычках как обозначение именно строки, а не списка кодов символов. Лишь в седьмой версии SWI-Пролога таким стало поведение по умолчанию, при этом была, естественно, частично утрачена совместимость со старыми текстами программ. Одновременно в SWI-Прологе ввели ещё литералы в *обратных* апострофах для обозначения «традиционных» списков кодов.

Если говорить совсем строго — и двойные кавычки, и обратные апострофы в новых версиях SWI-Пролога интерпретатор воспринимает в соответствии с настройками, которые задаются с помощью вышеупомянутой процедуры `set_prolog_flag`. «Флаг» для двойных кавычек называется `double_quotes`; ему можно приписать значение `string` (двойные кавычки порождают атомарную строку), `codes` (порождается «традиционный» список кодов символов), `chars` (порождается список символов) и `atom` (создаётся обычный атом, такой же, как с помощью апострофов). Аналогичный флаг для обратных апострофов называется `back_quotes`; набор возможных значений для него отличается: помимо `string`, `codes` и `chars`, допускается ещё вариант `symbol_char`, в этом случае обратный апостроф становится одним из символов, применимых в именах атомов, состоящих из знаков препинания, наряду со всякими плюсами и минусами (именно таким символом он был вплоть до шестой версии SWI-Пролога).

Начиная с седьмой версии, кавычки по умолчанию обозначают атомарную строку, обратные апострофы — список кодов символов; можно при запуске интерпретатора указать в командной строке ключ `--traditional`, тогда кавычки будут обозначать список кодов, а обратный апостроф превратится в знак препинания для соответствующих атомов.

В нашем разговоре о парадигмах уместно упомянуть текст, включённый создателями SWI-Пролога в документацию в качестве параграфа 5.4.2, озаглавленного «Why has the representation of double quoted text changed?» (см. `http://www.swi-prolog.org/pldoc/man?section=ext-dquotes-motivation`). Авторы текста отмечают, с одной стороны, три причины, почему использование атомов для представления текста «неадекватно». Во-первых, это «скрывает концептуальную разницу» между текстом и символами программы; содержимое текста, как правило, важно, поскольку используется во вводе-выводе, тогда как символы программы используются как идентификаторы, то есть их роль состоит в том, чтобы совпадать с такими же идентификаторами в других местах программы; выбор конкретных имён для них не должен, по идее, ни на что влиять. Во-вторых, атомы, используемые в роли глобально-уникальных идентификаторов, порождают дополнительные затраты времени — приходится каждый раз при создании атома проверять, нет ли в памяти интерпретатора атома с таким же именем. В-третьих, многие пролог-системы ограничивают длину имён атомов (хотя SWI-Пролог в их число и не входит).

С другой стороны, представление строки в виде списка кодов имеет свои проблемы. Во-первых, такие «строки» невозможно отличить от «обычных» списков чисел, под которыми, возможно, никто и не собирался подразумевать коды символов; во-вторых, списки оказываются слишком дорогостоящим способом хранения строк: на каждый символ приходится хранить в памяти целый сложный терм, так что памяти расходуется в десятки раз больше, чем при хранении строки способом, привычным по Паскалю или Си, и, кроме того, просмотр списков происходит намного медленнее. Вдобавок приходится тратить время на сборку мусора, остающегося после работы с ними.

Прочитав этот параграф из документации по SWI-Прологу и согласившись с ним, можно разве что удивиться, почему столь очевидные вещи дошли до апологетов Пролога столь поздно (версия 7.1 была выпущена в 2014 году).

Отдельного внимания заслуживают специфические прологовские ***переменные***, которые записываются как идентификаторы, обязательно начинающиеся с заглавной латинской буквы или с подчёркивания; `X`, `Person`, `_list`, `F15` и т. п. Можно написать и что-нибудь вроде `_75` (не делайте так).

То, насколько прологовские переменные не похожи на переменные из других языков, нам предстоит осознать позднее, после обсуждения операции ***унификации термов*** и алгоритма работы пролог-решателя. Пока отметим лишь несколько основных моментов: во-первых, переменные в Прологе всегда локальны в *одном предложении* программы; во-вторых, переменная в каждый момент времени может быть связана, а может и не быть связана с неким

значением, и значением переменной может быть произвольный терм, в том числе содержащий переменные (другие, а бывает, что и ту же самую); и, наконец, поскольку Пролог ищет подходящее решение методом проб и ошибок, переменные в ходе работы могут как связываться со значениями, так и терять значения (при возврате из тупика). Переменная, с которой в настоящий момент не связано никакое значение, называется ***свободной***; в противном случае, как можно догадаться, переменную называют ***связанной***.

Подчеркнём, что **переменная в Прологе именно связывается со своим значением, а не присваивается**; присваивание здесь вообще отсутствует как жанр. Если вы не уверены, что понимаете разницу, попробуйте взглянуть на происходящее под таким углом: *связать* можно только такую переменную, которая до этого была *свободна*, тогда как *изменить значение* переменной в Прологе нельзя; таким образом, прологовское связывание — в отличие от присваивания — не является деструктивной операцией, ведь при связывании создаётся новый объект (собственно связь переменной с её значением), но никаких старых объектов не уничтожается.

11.4.4. Операция унификации термов

Понятие ***унификации термов*** играет чрезвычайно важную роль в работе пролог-решателя. Сама по себе унификация не слишком сложна: *унифицировать* два терма значит подобрать (конечно, если это возможно) такие значения для входящих в эти термы переменных, чтобы при подстановке найденных значений вместо переменных у нас оба унифицируемых терма превращались в одно и то же выражение. Попытка унификации может закончиться неудачей, если термы унифицировать нельзя; если же унификация успешна, то её результатом становятся новые значения для переменных, или, иначе говоря, *новые связи* переменных со значениями.

Операция унификации в Прологе традиционно обозначается знаком равенства: `=(T1, T2)` или просто `T1 = T2`; например, унификация `[1, X] = [Y, 2]`, если переменные `X` и `Y` были свободны, приведёт к тому, что с переменной `X` будет связано значение `2`, а с переменной `Y` — значение `1`:

```
?- =([1, X], [Y, 2]).
X = 2,
Y = 1.

?- [1,X] = [Y,2].
X = 2,
Y = 1.
```

Более подробно описать унификацию можно так. Атомарные термы, такие как атомы и числа, унифицируются только с самими собой, не создавая при этом новых значений. Связанную переменную при унификации обрабатывают так, как будто вместо неё подставлено соответствующее значение. Свободная переменная успешно унифицируется с произвольным термом, и этот терм становится её значением. Что касается сложных термов (включая, естественно, и списки, ведь это тоже сложные термы), то они унифицируются следующим образом. Если у унифицируемых термов отличается главный функтор и/или различается арность, унификация заканчивается неудачей; в противном случае процедуре унификации последовательно подвергаются первые аргументы термов, затем вторые, третьи и так далее, при этом значения, получаемые переменными, *накапливаются*. Если очередная унификация аргументов (производимая с учётом уже полученных переменными значений) оканчивается неудачей, то неудачной считается и попытка унификации исходных термов; если же все аргументы успешно попарно унифицировались, унификация исходных термов объявляется успешной, а её результатом становится всё множество связей переменных с их значениями, полученных по мере унификации аргументов.

Забегая вперёд, проговорим ещё один момент, часто порождающий путаницу. Мы видели, что в ходе вычислений пролог-программы бывают ситуации с множественностью решений, то есть когда для одного и того же запроса Пролог может предложить несколько разных вариантов. Так вот, унификация двух термов либо проходит, либо нет, но если проходит — то ровно одним способом, то есть **нескольких разных решений для одной унификации не бывает**. Где в действительности находится источник множественности решений, мы увидим позднее.

Отдельного упоминания заслуживают часто возникающие связи переменных друг с другом. Простейший пример такой связи — результат унификации `X = Y` при условии, что обе переменные перед её выполнением были свободны. В этом случае, формально говоря, обе переменные остаются свободными в том смысле, что с любой из них можно связать значение, но при этом они *связаны друг с другом*: если одна из них оказывается связана, вторая получает то же самое значение.

В действительности свободной остаётся только одна из двух переменных, и именно она (как терм) становится значением второй переменной. Легко видеть, что ситуация здесь симметрична, то есть нам всё равно, какую из переменных Пролог оставил свободной, а какую связал.

Явное связывание двух переменных (конструкция «`X = Y`») обычно не применяется, поскольку в таком случае от одной из этих двух переменных можно попросту избавиться — например, заменить в предложении все вхождения `Y` на `X`. Тем не менее две переменные достаточно часто оказываются связаны друг с другом — например, когда они служат аргументами для унифицируемых сложных термов.

Ещё интереснее случай, когда переменная оказывается связана со сложным термом, внутри которого есть свободная переменная — что-то вроде `X = f(Y)`; здесь `X` получит значение `f(Y)`, а `Y` останется свободной. Надо сказать, что подобные случаи встречаются даже чаще, чем простая связь двух переменных друг с другом.

Пожалуй, самая любопытная ситуация создаётся, когда переменная связывается со сложным термом, *в который входит она сама*. Простейший случай такой связи — результат унификации `X = f(X)`. Старые версии Пролога в таких случаях в качестве значения `X` печатали что-то вроде

```
X=f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(...
```

Современные версии SWI-Пролога эту ситуацию отлавливают и печатают просто `X = f(X)`, что, к сожалению, снижает степень наглядности: можно ведь и не заметить, что структура связей получилась циклической. Но это именно так; например, результатом унификации `X = [1,2|X]` будет циклический список вида `[1,2,1,2,1,2,...]`. Важно понимать, что это именно ***циклический*** («бесконечный») список, а не что-то другое; так, полученный объект прекрасно унифицируется с термом `[1,2,1|Y]`:

```
?- X = [1,2|X], X = [1,2,1|Y].
X = [1, 2|X],
Y = [2, 1|Y].
```

11.4.5. Отношения, предикаты и факты

Центральным понятием прологовской (и вообще любой логической) вычислительной модели можно считать ***отношение*** (англ. *relation*), и здесь есть определённая проблема — нужно понять, что же это такое.

Из курса высшей алгебры читатель, скорее всего, может припомнить понятие *бинарного отношения на заданном множестве*. Строго говоря, множеств там может быть два, некие $\mathcal{A}$ и $\mathcal{B}$; говорят, что над ними задано бинарное отношение R, если для произвольной пары $(a, b), a \in \mathcal{A}, b \in \mathcal{B}$ можно ясно и недвусмысленно сказать, выполнено над ними сие отношение или нет; если отношение выполнено, это записывается как aRb.

Если задаться целью привести пример именно из математики и именно бинарного отношения над *различными* множествами, то можно взять множество эллипсов и множество треугольников, а в качестве примера отношений рассмотреть хорошо известные для окружностей (но ничуть не более сложное для произвольных эллипсов) «описан около» и «вписан в». Для каждого треугольника найдётся бесконечное множество описанных около него эллипсов, как и бесконечное множество эллипсов, вписанных в него, и в этом плане пример с эллипсами интереснее, чем с окружностями, ведь там «в одну сторону» получается обычная функция — каждому треугольнику соответствует ровно одна окружность, вписанная в него, и ровно одна описанная. «В другую сторону», конечно, всё интереснее, поскольку в каждую окружность можно вписать бесконечное множество различных треугольников, как и описать бесконечное их множество около неё; но если рассматривать именно эллипсы, то отношение получается «бесконечность к бесконечности» в обоих направлениях.

Намного чаще встречается бинарное отношение, для которого множества $\mathcal{A}$ и $\mathcal{B}$ совпадают: если, скажем, взять множество чисел, то над ним бинарными отношениями будут равенство, неравенство, всевозможные сравнения (в том числе $a < b$, называемое отношением порядка), если рассматривать множество *целых* чисел — делимость одного на другое и т. п. Важно тут отметить два момента. Во-первых, сам термин «отношение»: люди часто за-

бывают, что «$3 < 7$» в действительности означает «три и семь находятся в отношении “меньше”».

Во-вторых, **бинарное отношение R над множествами $\mathcal{A}$ и $\mathcal{B}$ есть подмножество декартова произведения множеств $\mathcal{A}$ и $\mathcal{B}$**, то есть мы можем записать что-то вроде $R \subseteq \mathcal{A} \times \mathcal{B}$, если же множества совпадают — то $R \subseteq \mathcal{A}^2$. Наблюдения показывают, что для кого-то это очевидно, а для кого-то, напротив, совершеннейший тёмный лес. Если это ваш случай — то разобраться следует прямо сейчас, тем более что ничего сложного тут нет. В самом деле, $\mathcal{A} \times \mathcal{B}$ — это множество *всевозможных* пар $(a, b), a \in \mathcal{A}, b \in \mathcal{B}$, тогда как бинарное отношение, каково бы оно ни было, выполняется лишь на *некоторых* из этих пар, и, что немаловажно, для каждой пары, для которой оно всё-таки выполняется, нельзя сказать ничего *другого*, кроме того, что вот да, для этой конкретной пары бинарное отношение выполняется (пара находится в отношении). Таким образом, чтобы задать бинарное отношение, и притом задать *исчерпывающе*, так, чтобы к сказанному не нужно было ничего прибавлять, *достаточно* будет просто тем или иным способом указать все пары значений, находящихся в описываемом отношении — то есть банально указать, что все вот эти пары в отношение входят, а все остальные — не входят. Это и есть, собственно говоря, *подмножество декартова произведения*.

Отношения бывают не только бинарными. В математике рассматриваются тернарные, кватернарные (четырёхместные) и вообще n-арные отношения для произвольных натуральных $n \geqslant 2$, но обычно всё-таки не рассматриваются вырожденные случаи $n = 1$ и тем более $n = 0$; как мы увидим чуть позже, нас такие ограничения не остановят. Важно понимать, что в любом случае **отношение есть подмножество множества кортежей**, сколько бы элементов ни входило в кортеж — два, пятнадцать, один или даже ноль; слово ***кортеж*** служит обобщением всевозможных терминов «пара», «тройка» (элементов), «пятёрка» (опять же элементов) и т. п.; можно считать, что это синоним неказистой словоформы «энка» (n-ка). Полезно знать, что в рассматриваемом контексте английский эквивалент термина «кортеж» — слово *tuple*, а вовсе не *cortege* и тем более не *train*, как изредка пишут русскоязычные авторы, заглянув в словарь.

Множество кортежей, рассматриваемое в роли отношения, может быть конечным, бесконечным, каким угодно — строго говоря, математическое понятие отношения не обязывает это отношение даже быть сколько-нибудь конструктивным. Никто, к примеру, не помешает нам рассмотреть множество всевозможных алгоритмов и ввести на нём бинарное отношение «A круче B», причём считать «круче» тот алгоритм, для которого окажется больше минимальная длина цепочки, к которой он неприменим. Мы даже сможем заявить, что введённое отношение есть отношение частичного порядка, и будем правы; нас при этом совершенно не смутит, что вообще-то задача определения «кто круче» окажется алгоритмически неразрешима.

Отметим ещё один терминологический момент. Читателю наверняка хорошо знакомо слово «предикат»; кто-то даже, возможно, с ходу припомнит, что предикат в учебниках математической логики часто определяют как «булеву функцию над произвольной областью определения», то есть такую функцию, которая для любой точки (точнее, любого элемента) своей области определения даёт либо значение 0, либо значение 1, подразумевая ложь или истину. Так вот, про ***предикаты*** мы скажем сразу две вещи. Во-первых, **не надо воспринимать предикат как булеву функцию** и в особенности как функцию в программистском смысле, *возвращающую* (в том или ином виде) истину или ложь; такое восприятие уводит в сторону от формирования парадигмы логического программирования. Во-вторых, **предикат и отношение — это одно и то же**, то есть эти термины — попросту синонимы. В самом деле, что такое элемент области определения предиката, как не кортеж? Но если это кортеж, то их множество, на котором предикат обращается в истину (а если говорить правильно — *множество кортежей, которые соответствуют предикату*) есть *буквально* то же самое, что и отношение.

В литературе можно встретить конструкции вроде «предикат определяет отношение» или, наоборот, «отношению соответствует предикат» (в последнем случае его, чтоб жизнь мёдом не казалась, обзывают «характеристическим» по аналогии с характеристической функцией). Нельзя сказать, чтобы такие словесные конструкции были совсем недопустимы; дело тут в том, что термины «отношение» и «предикат» относятся к разным предметным областям: первое пришло из алгебры, второе — из логики (в том числе математической). Само слово «предикат» исходно означает «суждение» или «утверждение». Но всё это никак не отменяет того факта, что речь в действительности идёт об одном и том же предмете — не о двух «соответствующих», а именно об одном.

Для полноты терминологической картины напомним ещё один момент. В теории баз данных ***таблицу*** часто называют *отношением*, отсюда происходит само название *реляционных* баз данных: поскольку *relation* по-английски «отношение», реляционная база данных — это база данных, построенная на отношениях. Чтобы исключить сомнения на тему «то ли это “отношение” или всё-таки какое-то другое», напомним, что строки таблиц — опять же *в теории* — называют кортежами. Чем «таблица» отличается от «предиката», так это тем, что в силу сугубо технических ограничений таблица всегда конечна, а предикаты (в том числе реализованные на Прологе) вполне могут давать бесконечную последовательность решений, т. е. в отношении, заданном в пролог-программе, может быть бесконечное множество кортежей. В остальном таблица, отношение и предикат — это одно и то же.

Отдельного упоминания заслуживают, по-видимому, вырожденные случаи. С унарным отношением всё более-менее понятно; говорить об «элементах, находящихся в отношении», кажется не вполне уместно, ведь элемент всего один; но всё остальное остаётся в силе: один элемент можно рассматривать как *унарный кортеж*, из этих кортежей можно образовать множество, так что и предикат, и таблица для унарного случая выглядят вполне естественно. Иное дело — случай 0-арный. Здесь приходится рассматривать некий «пустой кортеж» (кортеж, состоящий из элементов в количестве ноль штук); в принципе его рассмотреть никто не мешает, вот только он — увы — существует всего один, ведь если попытаться таких «пустышек» построить больше одной, то как мы их будем друг от друга отличать? Это, впрочем, не мешает существовать целым двум 0-арным предикатам, а равно и двум таблицам, имеющим ноль столбцов: первый предикат на «пустом кортеже» истинен, а соответствующая ему таблица состоит из одной строки — собственно говоря, пустой; второй предикат ложен, а таблица пуста, то есть у неё не только ноль столбцов, но ещё и ноль строк.

В Прологе оба очевидных 0-арных предиката встроены; тот, что истинен, называется `true`, а тот, что ложен, можно обозвать `false` или `fail`. Но это лишь начало истории; в Прологе есть ещё один 0-арный предикат, который не просто «истинен», но истинен *бесконечное число раз*, то есть он, если к нему обратиться, не просто выдаст истину, но и проинформирует решатель, что в этой точке вычислений возможно другое решение, и будет выдавать истину столько раз, сколько раз решатель в ходе вычислений в эту точку вернётся. Называется этот предикат `repeat`; почему именно так — станет ясно в одном из следующих параграфов.

Отметим, что любой из этих 0-арных предикатов можно было бы реализовать средствами Пролога, если бы они не были в него встроены. Так, аналог `true` (назовём его `success`) реализуется одним фактом:

```
success.
```

Немногим сложнее будет реализация аналога `false`:

```
failure :- 1 = 2.
```

Здесь важно понимать разницу между *всегда ложным* предикатом и предикатом *неопределённым*. Вычисление запроса `failure` даст *неуспех*, тогда как вычисление чего-нибудь вроде «`abrakadabra`» — при условии, конечно, что мы его не определили в программе — приведёт к *ошибке* и аварийному завершению программы.

Аналог `repeat` реализуется чуть сложнее — двумя предложениями:

```
forever.
forever :- forever.
```

Надо сказать, что множество встроенных 0-арных предикатов (или, лучше сказать, процедур) этим не исчерпывается: Пролог предусматривает довольно много таких процедур, имеющих побочный эффект, и в их число входит так называемое ***отсечение***, играющее чрезвычайно важную роль.

Для идентификации отношения в Прологе используется пара из имени и арности; в роли имени, что вполне естественно, может выступать произвольный ***атом*** (напоминаем, что атомы — в том значении слова, которое принято в Прологе — не следует путать с атомарными объектами; если здесь возникла заминка, вернитесь к стр. 145). С парой имя + арность мы уже встречались при обсуждении функторов (см. стр. 145), и такое сходство не случайно: именно сложные термы, образованные тем же атомом при той же арности, служат и для описания предиката, и для обращения к нему — но при этом, что важно, могут также работать и как простые данные, в общем случае никак с предикатом не связанные.

Пролог, как и Лисп, представляет собой язык типизированных значений и нетипизированных переменных; в языке отсутствуют средства для указания, какого типа

должен быть тот или иной аргумент. Как следствие, можно считать, что все отношения в Прологе определены над одним и тем же множеством — а именно, множеством возможных выражений. Если обозначить множество всех корректных термов Пролога, не содержащих переменных, через $\mathbb{T}$, получится, что для любого предиката P_k верно следующее: $P_k \subseteq \mathbb{T}^n, n = 0, 1, 2, ...$

Как ни странно, люди, причастные к реализации Пролога, обычно вообще не употребляют слово «предикат», а для обозначения той сущности, которая лежит в основе программы (собственно, программа из них состоит), к которой можно обратиться и которая при этом вычисляется, применяют слово ***процедура***. Попытаемся так поступить и мы, тем более что Пролог, как уже было сказано, нельзя рассматривать как язык чисто логический, так что, возможно, рассмотрение отдельно взятого «кирпичика» пролог-программы именно в виде обособленного фрагмента программы, а не в виде некоего «отношения» между объектами данных — более адекватно и лучше соответствует реальности. Тем не менее от слова «предикат» мы тоже не станем отказываться, а выбор между двумя терминами будем делать, основываясь на том, какой из них удачнее смотрится в том или ином контексте.

Самый простой способ задания предиката состоит, очевидно, в явном перечислении тех кортежей, которые ему удовлетворяют. Аналогично таблицу базы данных проще всего задать, перечислив все её строки (собственно говоря, в области баз данных только так всегда и поступают). В Прологе для такого явного перечисления используется набор ***фактов*** — простейших утверждений вида «данному предикату удовлетворяют данные значения аргументов».

Слегка видоизменив пример, приводившийся в обзоре основных парадигм, мы получим следующие факты:

```
parent("Mary", "Ann").
parent("Mary", "John").
parent("George", "John").
male("George").
male("John").
female("Mary").
female("Ann").
```

Итак, чтобы сформировать факт, мы выписываем в тексте программы сложный терм, соответствующий (по главному функтору и арности) нужному нам предикату, при этом, что важно, не содержащий переменных, и ставим после такой записи точку.

Процедуры, чьё описание состоит из одних фактов, играют в Прологе несколько особую роль. Неявно предполагается, что более сложные процедуры составляют «программу», чем бы она ни являлась, и не предполагают изменения во время исполнения, тогда как факты можно добавлять и убирать прямо во время работы, то есть это скорее не часть программы, а некие данные, над которыми программа работает.

Отметим, что SWI-Пролог позволяет во время работы изменять любые процедуры, а не только состоящие из фактов, то есть добавлять в программу и убирать из неё произвольные предложения. Мы бы не советовали читателю этим пользоваться; программа — это программа, данные — это данные. Когда программа начинает то и дело менять сама себя, удержать контроль за происходящим зачастую оказывается решительно невозможно. Так или иначе, по изначальной задумке модифицировать во время работы можно было только отношения, относящиеся к *базе данных*, то есть состоящие исключительно из фактов.

11.4.6. Правила и процедуры

Продолжая тему отношений между персонами, попытаемся сформировать предикат `loves(A, B)`, показывающий, кто кого любит в нашем небольшом универсуме. Разумеется, факты никто не отменял, так что можно, например, предположить, что `Mary` и `George` в нашем примере любят друг друга:

```
loves("Mary", "George").
loves("George", "Mary").
```

Вообще-то мы сделали такое предположение, зная, что у этих двух персонажей есть общие дети; Пролог позволяет при желании сделать такой вывод автоматически:

```
loves(X, Y) :- parent(X, Z), parent(Y, Z).
```

Здесь `loves` будет считать, что два объекта (`X` и `Y`) ему удовлетворяют, если найдётся некий `Z`, такой, что и `X`, и `Y` являются его родителями. Но мы так, пожалуй, всё-таки делать не будем — все прекрасно знают, что это допущение чрезмерно идеалистично. Тем не менее для предиката `loves` правила (в отличие от голых фактов) могут оказаться полезны. Для начала припомним известного персонажа по имени Дон Жуан, который (предположительно) любит поголовно всех женщин:

```
loves("Don Juan", X) :- female(X).
```

Дальше — больше: в старом мультфильме «Влюбчивая ворона» главная героиня — собственно Ворона — любит вообще всех. Это можно задать так:

```
loves("Vorona", X).
```


Обратите внимание, что это никакой не факт, хотя и выглядит похоже — но не следует забывать про условие об отсутствии в фактах переменных; здесь переменная присутствует, и, следовательно, это не факт, а правило, пусть и очень простое. А ещё эта запись содержит некую проблему, так что интерпретатор Пролога даже выдаст предупреждение. Дело в том, что переменная `X` входит в это предложение ровно один раз, то есть найденное для неё значение не может быть никак использовано (вспомним, ведь переменные Пролога локализованы в предложении). В таких случаях следует показать интерпретатору явным образом, что значение нас не интересует; делается это с помощью ***анонимной переменной***, которая обозначается одним знаком подчёркивания:

```
loves("Vorona", _).
```

Очень важно понимать, что это не простая переменная: вы можете в своём предложении использовать имя «`_`» сколько угодно раз, и каждое вхождение будет считаться свободным, то есть эти несколько экземпляров анонимной переменной никакого влияния друг на друга не окажут.

Теперь мы, можно надеяться, готовы к формальному описанию структуры программы на Прологе. Программа строится из отдельных ***предложений*** (англ. *clauses*). Предложение состоит из «головы» (*head*) и «хвоста» (*tail*), разделённых знаком `:-`; как мы видели, хвост (вместе с разделительным знаком) может отсутствовать; отсутствовать может и голова, в этом случае знак `:-` ставится в начале предложения — такое предложение задаёт *запрос*, то есть на самом деле нечто такое, что должно быть вычислено (исполнено) сразу после прочтения интерпретатором. Заканчивается предложение всегда точкой.

Голова предложения, если она есть, представляет собой ровно один *сложный терм*. Главный функтор и арность этого терма позволяют интерпретатору отнести предложение к одной из *процедур*; обычно программисты пишут предложения, составляющие процедуру, подряд, но Пролог этого не требует (хотя, конечно, это совершенно не повод рассеивать предложения одной процедуры по всему тексту — читать и анализировать такую программу будет совершенно невозможно). Как мы вскоре увидим, порядок предложений в процедуре важен; Пролог располагает их внутри процедуры в том же порядке, в котором они были встречены в тексте.

С «хвостом» предложения всё не столь просто. До сей поры в наших примерах он либо отсутствовал, либо состоял из последовательности термов, записанных через запятую. В действительности запятая — это *оператор*, объединяющий два терма и подразумевающий конъюнкцию, то есть считается, что между этими термами поставлена логическая связка «и». Это не единственная возможная связка; наряду с запятой используется ещё и точка с запятой «`;`», означающая «или». Её приоритет, что вполне естественно, ниже, чем приоритет запятой, ну а где появляются приоритеты — там обязательно возникает нужда в группировочных символах, в роли которых используются обычные круглые скобки.

Термы, из которых составлена хвостовая часть предложения, задают *обращения к процедурам* (написанным на Прологе или встроенным в него); **обращение к процедуре, записанное в виде терма, называется *целью*** (англ. *goal*). Вычисление целей в составе предложения происходит с учётом приоритета связок между ними, если же приоритеты одинаковы — то слева направо.

Как мы уже знаем, результатом вычисления отдельно взятой цели может стать неуспех, успех (при котором некоторые переменные, ранее свободные,

могут получить значения), а также успех с потенциальной возможностью других решений. В последнем случае пролог-решатель создаёт ***развилку***, то есть запоминает состояние вычисления данной цели, чтобы можно было к этой точке вычислений вернуться, получить следующее решение и снова (уже с новыми значениями переменных) попробовать остальные цели. Аналогичная развилка создаётся при вычислении дизъюнкции (точки с запятой), только здесь предлагается пробовать не другое решение, а другую (следующую) ветку дизъюнкции.

Получив при вычислении очередной цели неуспех, решатель возвращается к ближайшей развилке, при этом все переменные, получившие значение в точке развилки или позднее, снова становятся свободными. Такой возврат к развилке с освобождением переменных называется ***бектрекингом*** (англ. *backtracking*).

Если очередная цель, которую пришлось вычислять пролог-решателю, соответствует встроенной процедуре, то дальнейшее её вычисление полностью зависит от цели; многие встроенные процедуры вычисляются по общим правилам и вообще могли бы быть написаны на Прологе, если бы их не встроили, но некоторые делают что-то такое, чего написать на Прологе нельзя, напрямую влияют на базу данных и на ход вычислений. Если же обращение идёт к процедуре, которая написана на Прологе в виде последовательности предложений, то тут можно довольно чётко расписать, что́ будет происходить дальше.

Первое, что решатель делает при обращении к обычной процедуре, — последовательно пытается унифицировать цель (с учётом значений связанных переменных) с головной частью каждого из предложений процедуры. Если очередная унификация проходит успешно, решатель вычисляет хвостовую часть соответствующего предложения; при этом, если в процедуре ещё остались нерассмотренные предложения, создаётся развилка: сюда решатель может вернуться и попробовать оставшиеся предложения, если текущее упрётся в неуспех, либо если в неуспех упрётся вышестоящее вычисление (и потребует от текущей процедуры другого решения). Вообще **множественность предложений в процедуре — основной источник развилок**.

Если цель удалось унифицировать с *последним* (или единственным) предложением процедуры, то развилка не создаётся, а если она уже была создана — она удаляется, поскольку здесь бóльшему количеству решений взяться неоткуда.

Эффект от неуспеха унификации цели с головной частью предложения тоже зависит от того, последнее ли это предложение: если нет, решатель рассматривает следующее предложение, и так пока либо с каким-то из них не получится, либо пока они не кончатся; если же предложение последнее, вычисление всей цели объявляется неуспешным.

Надо сказать, что побочные эффекты в Прологе не в почёте, хотя во многих случаях без них не обойтись. Как мы уже обсуждали, без побочных эффектов циклы не имеют смысла, и их место занимает рекурсия; в Прологе это тоже так, предикаты часто приходится выражать через самих себя, или, если угодно, процедуры прямо и косвенно обращаются сами к себе. При этом переменные могут образовывать достаточно сложные взаимосвязи, и в особенности ситуацию запутывают связи между переменной и *ею же самой, но на другом уровне вложенности вычислений*.

11.4.7. Обращения к процедурам, прототипы и инверсия

Из предыдущего параграфа мы знаем, что *сложный терм*, расположенный в правой части пролог-предложения, представляет собой ***цель*** (англ. *goal*), то есть попросту обращение к процедуре. Обратим внимание, что *аргументами* терма-цели могут выступать как обычные (константные) выражения, так и переменные, и выражения, содержащие переменные. Если на момент обращения к процедуре (вычисления цели) переменная связана со значением, цель будет вычисляться так, как если бы вместо переменной подставили её значение. Гораздо интереснее ситуация, когда цель содержит свободные переменные: в этом случае в ходе вычисления цели переменные могут получить (а могут, впрочем, и не получить) значения.

Оставаясь в рамках логической семантики, мы можем сказать, что переменным будут *подобраны* такие значения, которые удовлетворяли бы *отношению*, заданному вычисляемой целью; если же вернуться к более привычной нам *процедурной* семантике, всё окажется (во всяком случае, на первый взгляд) изрядно проще: через свободные переменные процедура может ***вернуть значение***.

Мы уже встречались с такой ситуацией в примерах; попробуем привести ещё один пример, который позволит нам проиллюстрировать интересное свойство переменных в целях. Вспомним наш пример с отношением `parent` («является родителем») и опишем с его помощью небольшое генеалогическое древо[26]:

```
parent("Lucas", "Mary").
parent("Lucas", "Jason").
parent("Lucas", "Peter").
parent("Mary", "Fred").
parent("Mary", "Jane").
parent("Jason", "Sean").
parent("Jason", "Jessica").
parent("Jason", "Hannah").
parent("Jessica", "Joseph").
parent("Jessica", "John").
parent("Jessica", "Laura").
```

Опишем теперь более сложное отношение «является предком». Очевидно, что `X` является предком `Y`, если является его родителем, либо если является родителем кого-то другого, кто уже является предком `Y`; «кого-то другого» обозначим `Z`. Получим следующее:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Если вы используете версию SWI-Пролога старше, чем 7.*, вставьте в начало вашего файла запрос

```
:- set_prolog_flag(double_quotes, string).
```

Отношению SWI-Пролога к строкам мы посвятили рассуждение на стр. 146. В других реализациях Пролога, не поддерживающих понятие строковой константы, вам может потребоваться использовать апострофы вместо двойных кавычек, т. е. атомы вместо строк. Это не слишком изящно, но, увы, классические варианты Пролога нам не оставляют другого выхода: строки, заданные в двойных кавычках, интерпретатор потом печатает как списки кодов букв. В некоторых реализациях (включая SWI-Пролог) это поведение может быть изменено настройками, но не во всех.

Итак, теперь мы можем задать Прологу простейший вопрос — является ли кто-то предком кого-то ещё, например:

```
?- ancestor("Lucas", "Laura").
true .

?- ancestor("Jason", "Joseph").
true .

?- ancestor("Jason", "Fred").
false.
```

Стоит обратить внимание, что после первых двух запросов нам пришлось самим поставить точку, поскольку Пролог рвался найти «другие решения»; он бы их, впрочем, не нашёл — во всяком случае, для приведённого нами примера. Если бы у нас некий (возможно, дальний) предок некоего потомка был таковым по нескольким линиям (для этого где-то должен быть ребёнок от родственного брака), Пролог мог бы выдать своё `true` несколько раз.

Сформулируем более сложный запрос: кто является (известными Прологу) предками Джона?

```
?- ancestor(X, "John").
X = "Jessica" ;
X = "Lucas" ;
X = "Jason" ;
false.
```

Попробуем теперь «перевернуть» задачу и узнать, кто является *потомками* Джейсона; для этого спросим у Пролога, *для кого верно, что Джейсон является их предком*. Пролог прекрасно справится и с этим:

[26] Пример из статьи «Family tree» английской Википедии.

```
?- ancestor("Jason", X).
X = "Sean" ;
X = "Jessica" ;
X = "Hannah" ;
X = "Joseph" ;
X = "John" ;
X = "Laura" ;
false.
```

Мы можем и вовсе указать две переменные, спросив тем самым, «кто здесь чей предок». В ответ мы получим все 22 возможные пары значений, для которых (опять же, при заданных фактах) выполняется отношение `ancestor`:

```
?- ancestor(X, Y).
X = "Lucas",
Y = "Mary" ;
X = "Lucas",
Y = "Jason" ;
X = "Lucas",
Y = "Peter" ;
X = "Mary",
Y = "Fred" ;
   ...
```

(с позволения читателя мы обрежем выдачу интерпретатора и не будем показывать все 22 варианта).

Итак, написанный нами предикат `ancestor` способен отвечать на запрос с известными данными, подбирать для заданного первого аргумента подходящие значения второго аргумента и наоборот, и, наконец, находить все возможные (удовлетворяющие отношению) пары значений для обоих аргументов. Говорят, что *процедура* `ancestor` *способна работать в разных прототипах*: `ancestor(+, +)`, `ancestor(+, −)`, `ancestor(−, +)` и `ancestor(−, −)`; здесь «+» означает, что соответствующий аргумент задан, «−» — что используется свободная переменная. Такое свойство прологовских предикатов часто называют ***инверсией***[27].

Инверсия предикатов — это своего рода визитная карточка логического программирования, неотъемлемая и очень важная часть декларативной семантики. Возможность указывать свободные переменные *любым* аргументом цели позволяет думать о цели как просьбе *найти объекты, состоящие в заданном отношении* — между собой или с другими объектами, а то, как именно, в соответствии с какой процедурой решатель будет эти объекты искать — исключить из рассмотрения, чтобы не отвлекаться.

Чтобы привести более эффектный пример инверсии, обратимся к спискам и попробуем написать процедуру, удаляющую из заданного списка ноль или более элементов, равных данному. Иными словами, у нас есть элемент, есть список, и нужно построить такой список, который получен из исходного удалением *какого-то* (возможно, нулевого) количества заданных элементов. Ясно, что решений может быть больше одного; точнее говоря, если в исходный список входит n элементов, равных заданному, то решений получится 2^n, ведь каждый из подходящих элементов можно удалить, а можно и не удалять. Это выполняется и для случая, когда в список не входит вообще ни одного подходящего элемента: $2^0 = 1$, и решение действительно будет одно — равное исходному списку.

Мы назовём эту процедуру `del_any`; очевидно, что она будет тернарной. Первым аргументом будем задавать удаляемый элемент, вторым — исходный список, а третий аргумент используем для возврата результата; в терминах логической семантики мы пишем отношение между элементом и двумя списками, которое выполнено тогда и только тогда, когда второй список получен из первого путём удаления какого-то (возможно, нулевого) количества элементов, равных заданному.

Реализация, естественно, получится рекурсивной, ведь списки могут быть сколь угодно длинными, а в отсутствие циклов у нас для борьбы с такими ситуациями остаётся только рекурсия. Простейший возможный случай достигается на пустых списках: попытка «необязательного удаления» чего угодно из пустого списка даёт пустой список; этим случаем мы и воспользуемся как базисом для нашей рекурсии. При этом совершенно неважно, какой элемент удаляется, так что для него мы используем анонимную переменную. Получим следующее:

```
del_any(_, [], []).
```

Следующий важный случай — когда заданный элемент является первым элементом заданного списка. Его можно из списка выкинуть, тогда результат получится после того, как мы нашу операцию «необязательного удаления» произведём над остатком списка:

```
del_any(Elem, [Elem|Tail], Res) :- del_any(Elem, Tail, Res).
```

Наконец, последний случай — когда мы первый элемент из списка не удаляем, причём нам в общем всё равно, равен он заданному или нет, ведь мы и элемент, равный заданному, «имеем право» оставить, а тот случай, когда мы его не оставляем (удаляем), мы уже рассмотрели. Нам по-прежнему надо произвести «необязательное удаление» над хвостом списка, но полученный «новый хвост» послужит не результатом, а хвостом результата; головой будет голова исходного списка:

```
del_any(Elem, [X|Tail], [X|NewT]) :- del_any(Elem, Tail, NewT).
```

Попробуем полученную процедуру, что называется, в деле:

```
?- del_any(1, [1,2,1,3,1], L).
L = [2, 3] ;
L = [2, 3, 1] ;
L = [2, 1, 3] ;
L = [2, 1, 3, 1] ;
L = [1, 2, 3] ;
L = [1, 2, 3, 1] ;
L = [1, 2, 1, 3] ;
L = [1, 2, 1, 3, 1] ;
false.
```

Как можно заметить, Пролог выдал все решения, в которых первая единичка из списка удалена, и только потом — те решения, где она осталась; внутри каждой группы решений сначала показаны те, где удалена вторая единичка; ну и с третьей то же самое: сначала показывается решение, где её удалили, а потом то, в котором её оставили. Если поменять местами второе и третье предложение нашей процедуры, порядок выдачи сменится на обратный: сначала будут показываться решения, где удаления не произошло. Дело тут, очевидно, в том, что Пролог рассматривает предложения одной процедуры в том порядке, в котором они встретились в программе — и, как следствие, находит сначала те решения, которые получились из второго предложения, и лишь потом — те, что из третьего.

Разумеется, наша процедура будет работать и при всех трёх заданных аргументах:

```
?- del_any(1, [1,2,1,3,1], [1,2,3]).
true .

?- del_any(1, [1,2,1,3,1], [1,2,3,4]).
false.
```

Но гораздо интереснее другое: мы можем задать второй и третий аргумент, спросив, не сможет ли Пролог подобрать для нас подходящий первый:

```
?- del_any(X, [1,2,1,3,1], [1,2,3]).
X = 1 .

?- del_any(X, [1,2,1,3,1], [1,2,4]).
false.

?- del_any(X, [1,2,3], [1,2]).
X = 3 .
```

Можно и вовсе задать только второй аргумент. Соответствующий запрос в переводе на русский будет значить «какие списки можно получить из исходного удалением одного или более вхождения какого-то одного элемента»:

[27] В русскоязычных источниках, посвящённых Прологу, в том числе переводных, часто используются термины «обращение», «обратимость» и т. п. Соглашаясь с тем, что этот вариант для русского языка благозвучнее, чем иностранная (и чужеродная) «инверсия», мы, тем не менее, решили его не использовать. Причина тут в омонимичности слова «обращение»; в нашем тексте это слово часто используется как синоним термина «вызов» (подпрограммы, процедуры, функции, предиката и т. п.); соседство в тексте словосочетаний «обращение предиката» и «обращение к предикату», означающих при этом вещи совершенно разные и друг с другом не связанные, не добавит тексту понятности.

```
?- del_any(X, [1,2,1], L).
X = 1,
L = [2] ;
X = 1,
L = [2, 1] ;
X = 2,
L = [1, 1] ;
X = 1,
L = [1, 2] ;
L = [1, 2, 1].
```

Стоит обратить внимание, что в последнем решении не указано значение для `X`; дело в том, что эта переменная осталась свободной, ведь из списка ничего удалено не было, так что и связывать `X` оказалось не с чем.

Можно себе представить ещё один вариант инверсии этого предиката: задать элемент и результирующий список и посмотреть, что будет. Но здесь множество решений оказывается бесконечным, а полный перебор по бесконечному множеству вариантов — идея сомнительная. В частности, запрос `del_any(1, L, [2,3])` уйдёт в бесконечную рекурсию, так и не найдя ни одного решения; если же всё-таки поменять местами второе и третье предложение в нашем примере (назовём эту версию `del_any_inv`), получится вот что:

```
?- del_any_inv(1, X, [2,3]).
X = [2, 3] ;
X = [2, 3, 1] ;
X = [2, 3, 1, 1] ;
X = [2, 3, 1, 1, 1] ;
X = [2, 3, 1, 1, 1, 1] ;
       . . .
```

— и так пока нам не надоест. Формально тут всё верно, ведь каждый из этих списков удовлетворяет запрошенному отношению; ну а что мы, возможно, предпочли бы видеть другие решения — это наши проблемы. Дело в том, что с точки зрения Пролога множество решений является *упорядоченным*; например, список `[1,2,3]` входит в множество решений, но он входит в него *после* всех решений, образованных добавлением произвольного числа единиц в конец, а этих решений — бесконечное количество. Следовательно, до `[1,2,3]` мы просто никогда не доберёмся, хоть оно «там» (в гипотетическом множестве возможных решений) и «есть».

Перебор по бесконечности — не единственная причина, по которой инверсия предиката может сломаться. В следующих двух параграфах мы познакомимся с встроенными процедурами Пролога, которые начисто убивают декларативную (логическую) семантику, оставляя только семантику процедурную; их применение тоже мешает «бесплатной» инверсии предикатов. К сожалению, обойтись без этих инструментов на практике не получается, и мы это обязательно продемонстрируем; но прежде чем уходить из логической семантики, рассмотрим ещё два примера, эффектно демонстрирующих её несомненные достоинства.

Начнём с предиката `join/3`[28]. Пользуясь привычными терминами, мы можем сказать, что этот предикат (или, если уж на то пошло, *процедура*) объединяет два списка, создавая третий; но правильнее будет другое — логическое — описание того, чтó мы хотим получить. Итак, предикат (именно предикат, а не процедура) `join` должен задавать *отношение* между тремя списками (т. е. тернарное отношение), выполненное тогда и только тогда, когда третий список состоит из элементов, входящих в первые два списка, причём сначала в нём стоят элементы первого списка, а затем — второго, т. е. третий аргумент есть конкатенация двух первых.

К сожалению, для *эффективной* реализации этого предиката нам всё-таки придётся припомнить кое-что за пределами логической семантики — а именно то, как устроены списки, и что добавлять элементы следует в их начало, и изымать — тоже из начала. Из этого следует, что второй список вообще не надо анализировать; вместо этого нужно то ли расщепить первый список, добавить его элементы по одному в начало второго и посмотреть, получился ли третий, то ли, наоборот, сравнивать элементы первого и третьего, а то, что от третьего останется, сравнить со вторым.

Если всё это рассуждение непонятно, можно позволить себе отвлечься от логически-декларативного мышления и вспомнить, как мы соединяли два списка в других языках, выстраивая рекурсию по первому из соединяемых списков. После этого реализация оказывается совсем тривиальной: базисом рекурсии послужит вырожденный случай, когда первый список пуст — тогда результатом становится второй список; ну а рекурсивный переход тут в том, чтобы расщепить первый список на голову и хвост, соединить полученный хвост со вторым списком, а потом к тому, что получилось, добавить ещё и голову. Такое рассуждение, с одной стороны, плохо тем, что явным образом обозначает первые два списка как исходные данные, а третий список — как получаемый результат, тогда как в логическом программировании нет никаких «данных и результатов», есть лишь отношения между объектами; зато, с другой стороны, оно мгновенно переводится в код на Прологе:

```
join([], L, L).
join([H|T], L, Res) :- join(T, L, M), Res = [H|M].
```

Эта реализация верна и будет работать, но опытный пользователь Пролога напишет иначе:

```
join([], L, L).
join([H|T], L, [H|M]) :- join(T, L, M).
```

Нетрудно видеть, что семантически это совершенно то же самое и вычисляться будет абсолютно так же; разница между этими двумя фрагментами в том, что первый показывает недостаточное владение мышлением в терминах отношений (в противоположность «результатам, получаемым из данных»).

А теперь попробуем этот предикат инвертировать. Варианты `join(+, −, +)` и `join(−, +, +)` пропустим; они тоже работают, просто они не столь интересны; запустим сразу вариант `join(−, −, +)`:

```
?- join(X, Y, [a,b,c]).
X = [],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = [] ;
false.
```

Кстати, это будет работать для обеих приведённых реализаций — как без переменной `Res`, так и с ней; это и понятно, ведь семантически эти реализации эквивалентны. Если у читателя есть лишние 20-30 минут, рекомендуем попробовать проследить, как именно пролог-решатель вычисляет цель `join(X, Y, [a,b,c])` для реализации с переменной `Res`.

Попробуем теперь описать предикат `listrev/2`, который выполнен тогда и только тогда, когда оба его аргумента являются списками, имеют одинаковую длину и первый список получен из элементов второго (или второй — из элементов первого, что то же самое), расположенных в обратном порядке[29]. При реализации нам здесь придётся волей-неволей выйти за пределы не только логической парадигмы, но даже и рекурсивного мышления: как мы знаем (см. §§9.2.4 и 9.2.5), «переворот» списка эффективнее всего реализуется с помощью накапливающего параметра, но техника накапливающего параметра предполагает императивное, а не рекурсивное мышление. Поскольку прологовские процедуры могут иметь одинаковые имена, если при этом различаются арностью, мы не будем вводить дополнительных идентификаторов для вспомогательной тернарной процедуры, назвав её тоже `listrev`. Накопителем будет выступать её третий аргумент:

```
listrev(L, R) :- listrev(L, R, []).

listrev([], A, A).
listrev([H|T], R, A) :- listrev(T, R, [H|A]).
```

В том прототипе, для которого мы эту процедуру написали, она, естественно, работает сразу же и без проблем:

```
?- listrev([1, 2, 3, 4], X).
X = [4, 3, 2, 1].
```

[28] В Прологе уже есть встроенный предикат, который делает именно это; он называется `append`.

[29] Было бы естественнее назвать этот предикат просто `reverse`; создатели Пролога так и сделали, встроенный предикат с этим именем работает именно так.

Попробуем устроить инверсию:

```
?- listrev(X, [1,2,3,4]).
X = [4, 3, 2, 1]
```

Нас уже не удивляет ситуация, когда Пролог, выдав единственное решение, спрашивает нас, искать ещё решения или нет: единственность решения может быть очевидна для нас, но не очевидна для Пролога. Если здесь поставить точку, всё будет в порядке — инверсия прошла, вычисление цели привело к нахождению решения (единственного), можно как будто бы радоваться. Но этот пример как раз и интересен тем, что здесь не всё так просто. Попытаемся нажать точку с запятой, разрешив Прологу поиск других решений — и решатель *уйдёт в бесконечную рекурсию*.

С сугубо логической точки зрения тут всё корректно: правильное решение найдено, никаких «неправильных» не возникло, ну а что в бесплодных поисках *несуществующих* решений решатель рухнул — это уже относится не к логике, а к процедурной реализации. Да и вообще, разве так сложно нажать точку?

Вспомним теперь, что вообще-то предикаты предназначены не для того, чтобы их вызывать из командной строки интерпретатора, а для использования в программах в качестве подпрограмм. Если наш `listrev` будет вызван из какой-то другой процедуры, причём именно в прототипе `listrev(`$-$`, +)`, «нажимать точку» там будет некому, а итогом станет аварийное завершение программы после исчерпания стека. Между тем вызов именно в таком «инверсном» прототипе может получиться непреднамеренно — из-за того, что кто-то решил попробовать инвертировать какой-то из «вышестоящих» предикатов.

Обнаруженная нами проблема оказывается неожиданно интересной в контексте обсуждения логической и процедурной семантики Пролога; её особая прелесть состоит в том, что способность предиката `listrev` к корректной работе в любых прототипах восстанавливается одним движением — но это «движение» не имеет никакого отношения к логической семантике и в её терминах не описывается. Мы вернёмся к нашей реализации `listrev` в конце следующего параграфа.

11.4.8. Отрицание и отсечение: убийцы логики

Отсечением в Прологе называется довольно специфическое действие: строго говоря, при его выполнении *уничтожаются все «развилки»* (точки, куда можно вернуться в надежде найти новое решение), созданные с момента входа в текущую процедуру. Обозначается эта страшная штука восклицательным знаком (формально говоря, это очередной встроенный «ноль-арный сложный терм»). С точки зрения логической (декларативной) семантики отсечение оказывается «не из этой сказки», это средство никакого отношения к предикатам первого порядка не имеет; если применить процедурную семантику, то есть рассматривать прологовскую процедуру в терминах последовательно выполняемых шагов, отсечение можно прочитать ещё и как «мы на верном пути, отбросить все сомнения!» — причём отбросить до такой степени, что забыть о них начисто, так что, если «верный путь» всё-таки не приведёт к решению, процедура окончится неуспехом (к счастью, это касается только текущей процедуры).

Откровенно говоря, отсечение не добавляет программам понятности и совершенно не способствует мышлению в парадигме логического программирования; кроме того, оно весьма эффективно убивает ***инвертируемость***, которую мы рассматривали в предыдущем параграфе. В некоторых случаях отсечение позволяет сделать процедуры эффективнее за счёт отбрасывания ненужных ветвей перебора, но это вряд ли могло бы оправдать его существование. Проблема в том, что в некоторых случаях без отсечения в Прологе *невозможно обойтись*. Оставшиеся в стеке «лишние» развилки могут приводить не только к снижению эффективности, но и к бесконечным бесплодным попыткам найти решение (такой случай мы наблюдали в конце предыдущего параграфа на примере предиката `listrev`), т. е. делать логически правильную программу некорректной с процедурной точки зрения. Кроме того, попытки обойтись без отсечения часто приводят к использованию ничуть не лучшего решения в виде *отрицания*.

Отрицание в Прологе записывается как `not(`T`)`, где T — произвольная цель. Отрицание имеет свои собственные проблемы; кратко говоря, оно тоже разрушает логическую семантику программы. Дело тут вот в чём. Цель `not(`T`)` будет истинной (что вполне естественно) тогда и только тогда, когда попытка вычислить цель T потерпела неудачу, то есть не дала ни одного решения. Коль скоро это так, в ходе вычисления отрицания *не могут возникнуть новые значения переменных*. В самом деле, отрицание не содержит никаких переменных, кроме входящих в цель T; но если T вычислится успешно (тем самым, возможно, связав некоторые переменные), то `not(`T`)` потерпит неудачу и вычисления дальше не пойдут (Пролог, как всегда при неудачах, попытается вернуться к ближайшей развилке); если же вычислить T не удалось, то `not(`T`)`, конечно, пройдёт успешно, но значениям переменных при этом взяться неоткуда.

Если вы ещё не увидели здесь проблему, попробуйте подумать о происходящем вот с какой точки зрения. Неявно любой запрос к пролог-решателю выглядит так: «Вот тебе терм; нельзя ли подобрать такие значения входящих в него переменных, чтобы он оказался истинным?». К целям, образованным с помощью `not`, это не относится.

Пусть, к примеру, переменные `X` и `Y` обе свободны и Пролог вычисляет такой фрагмент предложения:

```
... X = abrakadabra, not(X = Y), ...
```

С *логической* точки зрения тут всё просто, для `Y` подходит любое значение, кроме `abrakadabra`; но Пролог, за неимением лучшего, попытается вычислить цель `X = Y`, это у него, естественно, прекрасно получится, ведь `Y` (по условию) свободна, так что её можно связать с тем же значением, что и `X` (а точнее, просто с самой переменной `X`). Как следствие, такое вычисление в Прологе *всегда* кончится неудачей. В частности, попытка вычислить запрос

```
:- X = abra, Y = kadabra, not(X = Y).
```

«обречена на успех», решение тут очевидно — значения для обеих переменных явным образом заданы, цель `not` прекрасно пройдёт, поскольку к моменту её вычисления `X` и `Y` связаны, а их значения между собой не унифицируются. Если же мы поменяем цели местами:

```
:- X = abra, not(X = Y), Y = kadabra.
```

— то этот запрос, напротив, заведомо не пройдёт: до последней его цели вообще не дойдёт управление.

Отметим, что практически в любой версии Пролога имеется встроенный предикат, по смыслу противоположный предикату `=`, т. е. эквивалентный цели `not(`A`=`B`)`, где A и B — произвольные термы. Записывается это символом `\=`: например, `X \= Y`.

В SWI-Прологе имеется «альтернативное» обозначение для `not`: `\+`. Например, можно написать `\+ X = Y` или `\+ ancestor("Vasya", "Petya")`. Создатели SWI-Пролога почему-то предпочитают это обозначение, так что в документации по SWI-Прологу вы встретите именно его; ответ на вопрос «почему» автору книги неизвестен. Так или иначе, ничего привлекательного автор в обозначении `\+` не видит и использовать его в дальнейшем тексте не будет, отдав предпочтение традиционному `not`.

Обсуждение отсечения и отрицания мы начнём с простого примера. Попытаемся переделать недетерминированный предикат `del_any` из предыдущего параграфа в `del_all`, который будет однозначно и без колебаний удалять из заданного списка *все* вхождения заданного элемента. Первая строчка нашей процедуры будет такая же, как и в `del_any`, поскольку удаление чего угодно из пустого списка даёт, естественно, пустой список:

```
del_all(_, [], []).
```

А вот дальше у нас возникнут проблемы. Первое, что приходит в голову — это написать

```
del_all(Elem, [Elem|Tail], Res) :- del_all(Elem, Tail, Res).
del_all(Elem, [X|Tail], [X|NewT]) :- del_all(Elem, Tail, NewT).
```


— но в результате у нас получится уже знакомая нам процедура `del_any`: в самом деле, если очередной элемент равен заданному, то сработают *обе* эти строки, давая в качестве решения как список без удаляемого элемента, так и список, в котором элемент остался на месте.

Чтобы последнее предложение не срабатывало для случаев, когда голова списка (`X`) совпадает с удаляемым элементом (`Elem`), придётся ввести дополнительное условие, и нам для этого потребуется в той или иной форме записать отношение «не равно» (точнее, «одно не унифицируется со вторым»). Это можно сделать через отрицание, то есть написать `not(X = Elem)` или `X \= Elem`. Итоговая реализация получится такой:

```
del_all(_, [], []).
del_all(Elem, [Elem|Tail], Res) :-
    del_all(Elem, Tail, Res).
del_all(Elem, [X|Tail], [X|NewT]) :-
    X \= Elem,
    del_all(Elem, Tail, NewT).
```

В прототипе, который мы исходно имели в виду — `del_all(+, +, −)` — эта процедура будет работать правильно, как и в прототипе `del_all(+, +, +)`. Проблемы начнутся, если мы попытаемся этот предикат инвертировать. В прототипе `del_all(−, +, +)` наша процедура поведёт себя примерно как полицейская мигалка из известного анекдота — «то работает, то не работает». Для начала попробуем так:

```
?- del_all(X, [1,2], [2]).
X = 1 ;
false.
```

Всё вроде бы прекрасно, но радоваться нам явно рановато:

```
?- del_all(X, [1,2], [1]).
false.
```

Как говорится, оппаньки. И это несмотря на совершеннейшую очевидность решения:

```
?- del_all(2, [1,2], [1]).
true ;
false.
```

Легко видеть, что причиной столь странной работы нашей процедуры оказывается именно `not`. Если первый аргумент не задан, то третье предложение процедуры никогда не даст успешного решения; получить своё значение первый аргумент может во *втором* предложении, и тогда все рекурсивные вызовы (для списков меньшей длины) уже отработают успешно. Но для этого нужно, чтобы искомое значение оказалось в голове списка, что мы, собственно говоря, и наблюдаем: если третий аргумент получается из второго путём удаления всех вхождений его (второго аргумента) головного элемента, то процедура находит правильный ответ, в противном случае — не находит.

Заставить `del_all` корректно инвертироваться *возможно* — например, задействовав встроенные процедуры `var` и `nonvar`, позволяющие узнать, связана та или иная переменная или свободна; фактически при этом нам придётся написать для прототипа `del_all(−, +, +)` отдельное решение, к которому основная процедура будет обращаться только в случае, если значение первого аргумента не задано. Но это уже, как говорится, «не то». Инверсия предикатов столь привлекательна лишь до тех пор, пока она «бесплатна», то есть получается сама собой; с `del_all` такого не получится, как ни крутись.

По этому поводу можно разве что вздохнуть; но если логическая семантика у нас всё равно уже убита и процедура не может задавать логическое отношение, став именно что процедурой (а не предикатом), то хотя бы в этом качестве мы можем сделать её несколько лучше. Коль скоро при вычислении процедуры успешно прошла унификация запроса с головным термом второго предложения (т. е. если первый элемент списка подлежит удалению), то нам остаётся лишь выполнить рекурсивный вызов, и всё; сам этот рекурсивный вызов теоретически может выдать больше одного решения (хотя в нашем случае — не может, но это не столь важно), но *на текущем уровне вызовов* никаких дополнительных вариантов уж точно не предвидится, третье предложение заведомо не сработает из-за `not`. Итак, позади у нас развилка, образованная множественностью предложений процедуры (мы вычисляем второе предложение, а есть ещё третье), но при этом мы точно знаем, что третье уже вычислять не нужно, поскольку оно заведомо не даст дополнительных решений. Развилка есть, но она бесполезна и было бы, видимо, *правильно* её убрать; вспомнив начало параграфа, мы догадаемся, что здесь как раз самое время применить отсечение. Заметим кстати, что если унификация прошла с головой *первого* предложения, то других решений тоже уже не будет, поскольку головные термы остальных предложений заведомо не могут унифицироваться с запросом, у которого во втором аргументе пустой список. Итак, напишем новую версию `del_all`, воспользовавшись в двух местах отсечением; `not` в третьем предложении нам теперь не нужен, уберём его. Получим следующее:

```
del_all_cut(_, [], []) :- !.
del_all_cut(Elem, [Elem|Tail], Res) :-
    !, del_all_cut(Elem, Tail, Res).
del_all_cut(Elem, [X|Tail], [X|NewT]) :-
    del_all_cut(Elem, Tail, NewT).
```

Обнаружить, чем этот вариант лучше предыдущего, можно сразу же. Предыдущая версия `del_all`, в каком бы прототипе она ни работала, всегда пыталась найти больше решений там, где их нет и быть не может — нам всё время приходилось нажимать точку с запятой (или простую точку), чтобы завершить вычисления. Новая версия — `del_all_cut` — напротив, не требует этого никогда, она выдаёт единственное решение, если оно есть, и завершает вычисления без нашего вмешательства, поскольку теперь пролог-решатель точно знает, что здесь никаких дополнительных решений не будет. Ну а знает он это, поскольку на момент выдачи решения «позади» нет ни одной развилки — мы их все изничтожили своими отсечениями.

Вернёмся теперь к предикату `listrev`, который мы пытались написать в конце предыдущего параграфа:

```
listrev(L, R) :- listrev(L, R, []).
listrev([], A, A).
listrev([H|T], R, A) :- listrev(T, R, [H|A]).
```

и попробуем понять, почему инвертированные запросы приводят к бесконечной рекурсии. Для этого попытаемся проследить вычисления на примере самого простого запроса, приводящего к аварии: `listrev(X, [])`.

Здесь нам придётся вспомнить, что переменные в Прологе локальны в данном предложении на данном уровне вложенности вызовов. Чтобы не запутаться при одновременном использовании переменных, имеющих одинаковые имена, но находящихся на разных уровнях вычислений, введём для этих уровней специальное обозначение — номер уровня в нижнем индексе. Например, $\mathtt{X}_0$ будет соответствовать переменной `X` из исходного запроса, а $\mathtt{R}_7$ — переменной `R` на седьмом уровне вложенности.

Для рассмотрения нашего простенького примера этого будет достаточно. На самом деле в общем случае переменная в Прологе может продолжать существовать после того, как вычисления на уровне, породившем эту переменную, уже закончились; между тем другая ветвь вычислений может снова дойти до того же уровня и дальше. По мере выхода из вложенных вычислений Пролог вынужден переносить задействованные переменные в *кучу* (в некую область динамической памяти), а если говорить совсем точно, это происходит, когда значением одной переменной становится сложный терм, содержащий другую переменную. Для случая связи двух переменных между собой, даже если они относятся к разным уровням вычислений, перенос в кучу не требуется.

Каждая реализация Пролога тем или иным способом решает вопрос, как показать на печати переменную из кучи, если она так и осталась свободной. Для примера можно рассмотреть процедуру, состоящую из одного предложения «`pred(X, Y) :- X = f(Y).`». Если при вызове этой процедуры второй параметр оставить свободным (например, поставить там анонимную переменную), мы сможем увидеть, как SWI-Пролог печатает свободные переменные из своей кучи:

```
?- pred(X, _).
X = f(_G894).
```

Вот это вот `_G894` — как раз оно и есть. Не удивляйтесь, когда увидите что-то подобное при отладке своих программ на Прологе.

Итак, посмотрим, что происходит при вычислении цели `listrev(X, [])`. На «нулевом» уровне — т. е. на уровне запроса — у нас всего одна переменная, $\mathtt{X}_0$. Решатель начинает вычисление цели, то есть производит обращение к процедуре `listrev/2`; эта процедура состоит из единственного предложения

```
listrev(L, R) :- listrev(L, R, []).
```

Следовательно, у нас на первом уровне вложенности вычислений появятся переменные $\mathtt{L}_1$ и $\mathtt{R}_1$. Решатель производит унификацию вычисляемой цели с головой предложения; унификация, естественно, проходит успешно, а её результатом становятся связи $\mathtt{L}_1$ = $\mathtt{X}_0$ и $\mathtt{R}_1$ = `[]`. Имея этот контекст, решатель переходит к рассмотрению правой части предложения, где стоит единственная цель `listrev(L, R, [])`, причём обе входящие в неё переменные у нас связаны, так что эта цель вычисляется так же, как вычислялась бы цель `listrev(`$\mathtt{X}_0$`, [], [])`.

Здесь мы имеем уже обращение к процедуре `listrev/3`, вычисляться она будет на втором уровне вложенности. Предложений в этой процедуре, напомним, два:

```
listrev([], A, A).
listrev([H|T], R, A) :- listrev(T, R, [H|A]).
```

Унификация активной цели с головой первого предложения проходит успешно, добавляя в наш контекст ещё две связи: X_0 = [] и A_2 = []. Правая часть предложения пуста, так что процедура сейчас «вернёт управление» на вышестоящий уровень; слова «вернёт управление» мы вынуждены взять в кавычки, поскольку вычисление цели listrev(X_0, [], []) на этом не закончится, ведь в нашей процедуре есть ещё и второе предложение, так что здесь возникнет (на самом деле уже возникла — перед началом вычисления первого предложения) развилка, к которой позже можно будет вернуться. Пока запомним, что на втором уровне вложенности вычисляется цель listrev(X_0, [], []), вызванная из единственного предложения процедуры listrev/2 с контекстом L_1 = X_0 и R_1 = [], и первое предложение процедуры listrev/3 уже рассмотрено, ожидает рассмотрения второе предложение.

Следующие несколько шагов решателя довольно очевидны: на первом уровне (в процедуре listrev/2) предложение закончилось, фиксируется успех вычисления цели нулевого уровня listrev(X, []), при этом в контексте имеется связь X_0 = [], так что решатель печатает первое решение — ту самую строку

```
X = []
```

Пока что всё идёт хорошо, но мы помним, что авария происходила как раз после этого, и чтобы её воспроизвести, нажимаем точку с запятой, требуя от Пролога продолжить поиск решений. Развилка в ходе вычислений возникла всего одна, к ней решатель и вернётся; мы снова окажемся на втором уровне вычислений в процедуре listrev/3, а связи, попавшие в контекст позже создания этой развилки, исчезнут, так что у нас контекст снова состоит всего из двух связей — L_1 = X_0 и R_1 = []; цель listrev(X_0, [], []) в этот раз будет унифицироваться с термом listrev([H|T], R, A), и эта унификация, как ни странно, пройдёт; новые связи будут выглядеть так: X_0 = [H_2|T_2], R_2 = [], A_2 = []. Поскольку унификация с головой предложения прошла успешно, решатель вынужден рассмотреть правую часть этого предложения, где стоит единственная цель listrev(T, R, [H|A]). С учётом связей переменных эта цель будет вычисляться как listrev(T_2, [], [H_2|[]])[30]; у нас, таким образом, имеется рекурсивное обращение к процедуре listrev/3, в этот раз она будет вычисляться на третьем уровне вложенности. Унификация цели с головой первого предложения не пройдёт из-за третьего аргумента, в цели этот аргумент представляет собой список, хоть и недостроенный, но уж точно не пустой. Так что остаётся всего одно предложение — второе; развилки никакой при этом не возникнет (точнее говоря, она, скорее всего, будет создана перед попыткой унифицировать цель с первой головой, а после неудачи этой унификации — ликвидирована).

Унификация активной цели с головой второго предложения, как ни странно, пройдёт, установив связи T_2 = [H_3|T_3], R_3 = [], A_3 = [H_2|[]]. Цель в правой части предложения на этом уровне вычислений будет рассмотрена как listrev(T_3, [], [H_3, H_2|[]]). Вычисление этой цели приведёт к обращению listrev(T_4, [], [H_4, H_3, H_2|[]]) и так далее до бесконечности — точнее говоря, до аварии, ведь в стеке очень скоро кончится место, да и память, содержащая переменные и их связи, тоже не безразмерна.

Как видим, проблема тут в том, что, выдав решение, полученное из первого предложения listrev/3, решатель пытается найти какие-то ещё решения, вычисляя *второе* предложение процедуры, которое, конечно же, никаких новых решений дать не может, но при этом уходит в бесконечную рекурсию, приводя к аварии всего вычисления. Но ведь если у нас в процедуре listrev/3 прошла унификация цели с головой первого предложения, то (мы, в отличие от Пролога, это прекрасно понимаем) никаких других решений найти уже нельзя. Попробуем отразить это понимание в тексте процедуры, поставив знак отсечения:

```
listrev([], A, A) :- !.
listrev([H|T], R, A) :- listrev(T, R, [H|A]).
```

С помощью интерпретатора легко убедиться, что в этом варианте процедура прекрасно работает в обоих прототипах, ни в какую бесконечную рекурсию не уходя!

Результат нашего эксперимента может показаться противоречащим тому, что мы уже успели наговорить про отсечение — ведь здесь отсечение не только не «убивает» инверсию предиката, но, напротив, заставляет её работать. Так в самом деле бывает, и даже довольно часто, listrev здесь совершенно не уникален; мы, впрочем, и не утверждали, что отсечение *всегда* убивает инвертируемость, речь шла лишь о возможности такого убиения.

Отметим, что ситуации, в которых применяется отсечение в del_all_cut и listrev, принципиально различны. В процедуре del_all_cut развилка, уничтожаемая отсечением, привела бы (если бы её не убрали) к бóльшему числу найденных решений, хоть и неправильных по условию решаемой задачи. Иначе говоря, отсечение в del_all_cut меняет происходящее с *логической* точки зрения.

Другое дело — процедура listrev; здесь мы отсечением уничтожаем развилку, которая *всё равно не могла бы дать никаких решений*, и неважно, сможет ли пролог-решатель в какой-то момент обнаружить тупиковость дальнейших вычислений или, как в нашем примере, уйдёт в бесконечность. С *логической точки зрения* мы, поставив отсечение, вообще здесь ничего не изменили, решений не прибавилось и не убавилось, мы просто отсекли заведомо бесплодную ветвь вычислений — но это уже не логическая семантика, а процедурная. Отсечение здесь — это чистая оптимизация, и в такой своей ипостаси отсечение обычно ни к каким плохим последствиям не приводит.

11.4.9. Арифметика

Сложно представить себе язык программирования без возможности вычисления арифметических выражений, и в Прологе, разумеется, эта возможность есть. В арифметических вычислениях участвуют целые числа, числа с плавающей точкой и *рациональные числа*, записываемые с помощью функтора rdiv, который можно использовать как оператор; например, rdiv(2, 3) или 2 rdiv 3 — это запись числа $\frac{2}{3}$.

Выражения образуются с помощью функторов +/2, -/2, */2, //2, -/1, +/1 и целого ряда других; кроме чисел трёх перечисленных типов, в выражениях можно использовать переменные. К примеру, -1 * (3 * X - 25) — это с точки зрения Пролога то же самое, что и *(-(1), -(*(3, X), 25)), то есть самый обыкновенный сложный терм. Он, собственно говоря, так и останется просто термом, то есть обычной прологовской структурой данных, которую можно унифицировать с другими термами, вставлять в другие термы, в том числе в списки, связывать с переменными и т. п., и никаких вычислений при этом происходить не будет.

Вычисление терма как арифметического выражения — это особое свойство некоторых встроенных процедур, простейшая из которых называется is, предполагает два аргумента и используется обычно как инфиксный оператор. К примеру, если терм X is 4*5 вычислить в качестве цели, с переменной X будет связано значение 20; если после этого вычислить цель A is -1 * (3 * X - 25), A получит значение -35. В общем случае **процедура is/2 вычисляет свой второй (правый) аргумент как арифметическое выражение и полученный результат унифицирует со своим первым (левым) аргументом**. Ясно, что на момент вычисления арифметического выражения все переменные, входящие в него, должны иметь значения; свободная переменная вызовет ошибку, причём это будет именно ошибка, а не неуспех. В качестве левого аргумента is обычно используют переменную, и она на момент вызова is в большинстве случаев свободна. Иначе говоря, данная процедура предназначена для использования в прототипе is(−, +).

Для примера рассмотрим процедуру, вычисляющую длину списка. В Прологе уже есть встроенная процедура для этого, она называется length, так что нашу процедуру мы назовём иначе, чтобы они не путались:

```
listlen([], 0).
listlen([_|T], L) :- listlen(T, TL), L is TL + 1.
```

Кроме процедуры is, вычисление выражений предполагают предикаты сравнения чисел, а именно <, >, >=, =< (меньше либо равно; обратите внимание на непривычный порядок символов — дело тут в том, что атом <= в Прологе используется для других целей, не связанных с арифметикой), =:= и =\= («равно» и «не равно» в арифметическом смысле, т. е. арифметическое сравнение двух чисел на равенство и неравенство). Все эти предикаты сначала *вычисляют* оба своих аргумента, считая их арифметическими выражениями, и только после этого выполняют сравнение двух полученных чисел. Каждое из

[30] Для наглядности здесь и далее мы оставляем запись [H_n|[]] как есть, хотя это то же самое, что и [H_n].

двух чисел может быть любого из трёх типов, проблем со сравнением чисел разных типов у Пролога не возникает.

Требование связанности переменных, входящих в вычисляемое арифметическое выражение, можно сформулировать иначе: **Пролог не умеет решать уравнения**. Причина этого очевидна, если вспомнить, что для полиномиальных уравнений первой и второй степени формулы решения проходят в школе, для уравнений третьей и четвёртой степени формулы существуют, но мало кто их помнит, про уравнения пятой степени и выше доказано, что в общем виде их невозможно разрешить в радикалах, при этом для уравнений пятой степени общий аналитический метод решения всё-таки есть (не в радикалах), если же рассматривать более высокие степени, то для них общие аналитические решения неизвестны. Заметим, речь пока что идёт только о полиномиальных уравнениях, а Пролог в арифметических выражениях позволяет использовать и экспоненту, и тригонометрические функции, и много чего ещё. Очевидно, что встроить в Пролог «решатель любых уравнений» заведомо невозможно, как и вообще построить такой решатель. Естественно, решать *неравенства* Пролог и подавно не умеет.

«Необратимость» (невозможность инверсии) процедуры `is` и других процедур, вычисляющих арифметические выражения, ставит эти процедуры в один ряд с отрицанием и отсечением в том смысле, что их использование рушит логическую семантику прологовских программ. Рассмотрим для примера процедуру `prefix`, которая из заданного списка выделяет первые N элементов и строит из них список. Написать такую процедуру несложно. Базисом рекурсии послужит «нулевой» случай, когда результат должен быть пустым, а исходный список нас вообще не волнует; если же нужно выделить больше нуля элементов, то результат строится из первого элемента исходного списка и хвоста, который получается как префикс длины $N - 1$ хвоста исходного списка:

```
prefix([], _, 0).
prefix([H|T2], [H|T], N) :- N2 is N - 1, prefix(T2, T, N2).
```

В том прототипе, который мы изначально предполагали по условию — `prefix`$(-, +, +)$ — наша процедура будет прекрасно работать:

```
?- prefix(L, [a,b,c,d,e,f], 3).
L = [a, b, c]
```

— но на этом, собственно говоря, всё, инвертировать её не получится, даже если длина нас вообще не волнует:

```
?- prefix([a,b,c], [a,b,c,d,e,f], _).
ERROR: prefix/3: Arguments are not sufficiently instantiated
```

Нетрудно видеть, что при таком обращении переменная `N` на момент вычисления цели `N2 is N - 1` окажется свободной, отсюда ошибка; а жаль, ведь вопрос, является ли один список префиксом другого, вполне может возникнуть, и придётся для него писать другую процедуру.

Средств, введённых к настоящему моменту, нам для решения этой проблемы не хватит. Позже, в §11.4.11, мы вернёмся к предикату `prefix` и заставим его работать в разных прототипах, воспользовавшись инструментами, не имеющими отношения к логике.

Рассмотрим ещё один своеобразный и довольно показательный пример. Пусть нам потребовался предикат `ten/1`, истинный для целых чисел от 1 до 10 и ложный для всех остальных чисел. «Лобовой» способ

```
ten(1).
ten(2).
    ...
ten(10).
```

мы применять откажемся, как и другие подходы, основанные на явном перечислении всех возможных решений, такие как

```
ten(X) :- member(X, [1,2,3,4,5,6,7,8,9,10]).
```

В самом деле, такие решения подходят, пока множество решений сравнительно невелико, но если решений будет, скажем, несколько тысяч, то вряд ли нам захочется перечислять их все в явном виде; поэтому попробуем написать решение, основанное на вычислениях. При этом мы столкнёмся с несколько неожиданными проблемами. Настоятельно рекомендуем читателю разобраться, какова причина проблем в каждом из рассмотренных ниже случаев — это позволит лучше прочувствовать работу пролог-решателя.

Начнём с простого варианта: *единица входит в множество* (базис рекурсии), *а произвольное число входит в множество, если в множество входит число, на единицу меньшее, чем это, и при этом само число не превосходит 10*. На Прологе это можно записать так:

```
ten(1).
ten(X) :- ten(Y), X is Y+1, X =< 10.
```

Попробовав это решение в деле, мы обнаружим, что при заданном аргументе от 1 до 10 обращение к предикату завершается успехом, и это вроде бы правильно, но вот если задать число, не входящее в множество, то вычисление уйдёт в бесконечную рекурсию. Больше того, и «успех» тоже выглядит не очень убедительно: интерпретатор выдаёт ответ `true` и ждёт, что́ мы ему скажем дальше — и если нажать точку с запятой, мы получим, опять же, бесконечное вычисление. Ещё интереснее выглядят результаты запроса `ten(X)`: интерпретатор честно выдаёт все десять возможных решений, после чего уходит в бесконечные вычисления.

Отметим, что перенести сравнение из конца предложения в его начало мы не можем: в этом случае предикат перестанет зацикливаться на аргументах, бо́льших десяти, но для незаданного аргумента он работать не будет вообще, поскольку сравнению нужно, чтобы все переменные были связанными. Если же сравнение разместить в середине предложения, т. е. сразу после рекурсивного вызова (естественно, сравнивать придётся `Y`, и уже не с десяткой, а с девяткой), то в поведении нашей процедуры не изменится ровным счётом ничего.

Отсечение здесь тоже, на первый взгляд, не помогает. Если поставить его в начало предложения:

```
ten(X) :- !, ten(Y), X is Y+1, X =< 10.
```

— то мы получим уже порядком поднадоевшее исходное поведение: зацикливание после выдачи всех решений при запросе `ten(X)`, а также при запросах на числах, не входящих в множество решений; а в какое бы место *после* рекурсивного вызова `ten` мы ни вставили отсечение — любым из следующих способов:

```
ten(X) :- ten(Y), !, X is Y+1, X =< 10.
ten(X) :- ten(Y), X is Y+1, !, X =< 10.
ten(X) :- ten(Y), X is Y+1, X =< 10, !.
```

— предикат внезапно начнёт делать вид, что решений существует всего два (`1` и `2`); во всяком случае, при запросе со свободной переменной предикат будет выдавать именно их. Из запросов с заданным значением успешными окажутся `ten(1)`, `ten(2)`, а если отсечение стоит правее цели `is` — то ещё `ten(3)`, но на этом всё, как будто никаких других решений здесь нет и никогда не было.

Попробуем уцепиться за тот факт, что теперь процедура, выдавая не все нужные решения, при этом хотя бы не зацикливается, так что мы вполне можем оказаться на верном пути. Несложно понять, что, поставив отсечение после рекурсивного вызова, мы *отбрасываем верные решения*. Например, при запросе `ten(3)` (самом простом из работающих неправильно) пролог-решатель попробует первое предложение, унификация не пройдёт, он попробует второе предложение, получит из рекурсивного вызова значение `Y = 1`, после чего следующая цель (`X is Y+1`) при подстановке переменных превратится в `3 is 1+1` и выдаст неуспех. Если отсечение стоит перед ней, то на этом всё кончится; если оно стоит *после* неё (перед сравнением или в конце предложения), то, как ни странно, получится успех, что верно. Но радоваться рано: запрос со свободной переменной по-прежнему не будет выдавать решение `3`. В самом деле, в ответ на этот запрос предикат создаст развилку, выдаст решение `1`, после возврата к развилке попробует второе предложение, получит `Y = 1` из рекурсивного вызова, путём вычислений получит `X = 2` и выдаст его, но перед тем, как оно будет выдано, отсечение ликвидирует развилку, а с ней и все остальные решения. Как следствие, запросы с заданным значением по-прежнему не будут считать верным решение `4` и все последующие.

Хитрость здесь в том, что «рубить» развилку нужно лишь тогда, когда все правильные решения уже выданы, то есть отсечение должно произойти не раньше, чем очередное число, полученное из рекурсивного вызова, окажется равным 10 или больше. Пока число, фигурирующее в роли значения `Y`, меньше, отсечение происходить не должно.

Здесь самое время вспомнить, что наряду с запятой мы для связывания целей в правых частях предложений ввели (см. стр. 149) ещё и точку с запятой («;»), которая означает операцию «или» (в отличие от запятой, которая символизирует «и»). До сих пор мы точкой с запятой ни разу не воспользовались и вполне могли уже забыть о её существовании, однако без неё исправить нашу процедуру `ten` будет довольно затруднительно. Напомним, что приоритет у точки с запятой ниже, чем у запятой, так что нам могут потребоваться (и потребуются) скобки. Правильная версия процедуры `ten` выглядит так:

```
ten(1).
ten(X) :- ten(Y), ((Y >= 10, !, fail) ; X is Y + 1).
```

Здесь мы, получив из рекурсивного вызова некое число, проверяем, можно ли из него сделать решение, и если нельзя — ликвидируем развилку и констатируем неуспех; в противном случае вычисляем новое решение, а развилку при этом не трогаем. Надо сказать, что в более сложных случаях точка с запятой бывает нужна довольно часто; процедура `ten` интересна разве что своей примитивностью, при которой, тем не менее, точка с запятой нам потребовалась.

Можно, впрочем, применить совсем другое решение и всё-таки обойтись без неё. Например, напишем предикат `mbetween`, выдающий все целые числа из заданного отрезка; первый аргумент задаёт начало отрезка, второй — конец отрезка, третий — собственно число, которое должно этому отрезку принадлежать. Такой предикат в Прологе уже есть и называется `between`, так что нам пришлось воспользоваться другим именем:

```
mbetween(A, B, _) :- A > B, !, fail.
mbetween(A, _, A).
mbetween(A, B, X) :- A1 is A + 1, mbetween(A1, B, X).
```

Предикат `ten` теперь реализуется тривиально — как частный случай `mbetween`:

```
ten(N) :- mbetween(1, 10, N).
```

11.4.10. Ввод-вывод

Рассказывая о средствах ввода-вывода в Common Lisp и Scheme (см. §§11.1.14, 11.2.3), мы многократно отмечали, что эти средства несут на себе отпечаток той эпохи, когда в природе не существовало конечных пользователей, а общее понимание ввода-вывода, основанное на потоках байтов, не успело сформироваться; кроме того, нам постоянно приходилось констатировать непродуманность предложенных нам инструментов и их оторванность от реальности.

В Прологе со всем этим дела обстоят довольно забавно. Имеется чёткое разделение средств ввода-вывода, представленных в Прологе, на «классические», которые в целом не подходят для серьёзных программ, и «новые», добавленные в реализациях, чтобы программы на Прологе более-менее вписывались в имеющееся устройство реального мира. В документации по SWI-Прологу «старую» модель ввода-вывода называют «Эдинбургским стилем», а «новую» — «стилем ISO». Впрочем, это касается только работы с множеством потоков ввода-вывода (например, файлов), а пока мы работаем со стандартным вводом и стандартным выводом, разницы между этими двумя моделями не видно. С работы со стандартными потоками мы и начнём.

Как и в Лиспе, в Прологе предусмотрен вывод произвольных выражений в виде, допускающем их обратное прочтение, и собственно чтение сколь угодно сложных прологовских выражений из текстового потока, в котором они должны быть представлены в соответствии с прологовским синтаксисом. Как обычно, мы настоятельно рекомендуем читателю сразу же забыть о самой возможности применения таких средств (см. стр. 129). Единственный полезный встроенный предикат из этой группы — `write`, он пригодится для печати чисел, а также строк, представленных в виде атомов или атомарных строковых объектов (см. стр. 146). Этот предикат мы уже встречали в §11.4.2; там же рассказано, как победить навязчивый сервис интерпретатора — приглашение к вводу и перехват `Ctrl-C`. Напомним, что выдать перевод строки можно предикатом `nl`, но можно вставить символ перевода строки в печатаемую строку — так же, как в Си, с помощью комбинации «`\n`».

Отбросив средства для чтения выражений, мы, как водится, остаёмся с одним посимвольным вводом. Изначально для этого в Прологе предназначался предикат `get/1`, единственным аргументом которого обычно выступает свободная переменная; по своей сути `get` напоминает функцию `getchar` из Си — читает символ из стандартного потока ввода и возвращает (в данном случае — связывает с переменной) его код от `0` до `255`, `-1` в случае наступления конца файла. Как водится, тут тоже не обошлось без нюанса: `get` пропускает все пробельные символы; когда-то кому-то показалось, что «так удобнее». Позднее стало ясно, что «так», разумеется, не просто неудобно, а вообще невозможно работать, и появился ещё один встроенный предикат — `get0/1`, который как раз обычно и используется. Делает он всё то же самое за исключением пропуска пробелов.

Как можно догадаться, для вывода символа по его коду используется предикат с названием `put`; например, `put(65)` напечатает заглавную латинскую букву `A`.

Картину способны изрядно запутать символы, не входящие в ASCII. Даже если в вашей системе используется однобайтовая кодировка, например koi8-r, SWI-Пролог считает своим долгом перевести прочитанный символ, не входящий в ASCII, во внутреннее представление, в котором для кодирования символов используются их номера из реестра Unicode. Например, если вы введёте строчную букву «я», `get0` «прочитает» код 1103, какая бы локаль ни была настроена в вашей системе.

По идее решить проблему должен бы был предикат `get_byte` — согласно описанию он работает точно так же, как `get0`, но читает из потока именно байт, а не что-то иное. Но не тут-то было: попытавшись его применить к стандартному вводу, вы, скорее всего, получите сообщение, что, мол, текстовый поток побайтово читать нельзя. Ситуацию можно исправить, если явным образом потребовать, чтобы стандартный ввод (или другой поток) рассматривался как простой бинарный; для этого нужно изменить атрибут `encoding` для данного потока, установив значение `octet`:

```
:- set_stream(current_input, encoding(octet)).
```

После этого `get_byte` благополучно заработает; впрочем, мы тут же обнаружим, что он нам уже не нужен — `get0` теперь тоже читает простые байты. Например, для буквы «я» при использовании koi8-r прочитанное значение будет 209.

SWI-Пролог предусматривает ещё `get_code` и `get_char`; зачем нужен `get_code` — остаётся загадкой, поскольку работает он, насколько можно судить, точно так же, как `get0`. Что касается `get_char`, то он превращает прочитанную букву в *атом* с именем из одной этой буквы, а при наступлении ситуации «конец файла» возвращает атом `end_of_file`.

Аналогичным образом для вывода представлены, кроме `put`, также `put_char` (печатает символ, представленный атомом с именем из одного символа), `put_code` (делает ровно то же, что и `put`) и `put_byte`; чтобы этот последний заставить работать, придётся сменить (точнее говоря, *отключить*) кодировку уже на стандартном выводе:

```
:- set_stream(current_output, encoding(octet)).
```

Эдинбургский стиль работы с потоками предполагает, что в каждый момент времени программа работает с *одним* потоком ввода и одним потоком вывода. Изначально это стандартный ввод и стандартный вывод. Чтобы начать чтение из файла, применяется предикат `see`, аргументом которому служит имя файла (в виде, как ни странно, атома, то есть тут даже строковые объекты не работают); файлов можно таким способом понаоткрывать сколько угодно, переключаясь между ними тем же предикатом `see` — при этом файлы не закрываются и повторно не открываются, имя используется для идентификации потока. Выполнение цели `seen` (без аргументов) закрывает текущий файл и возвращает активную роль стандартному вводу. Временно переключиться на стандартный поток ввода, не закрывая текущий активный читаемый файл, можно вычислением цели `see(user)`. Между прочим, это означает, что *дисковый файл с именем* `user` *так не прочитать*.

Чтобы привести пример работы с вводом в этом стиле, припомним, что встроенный предикат `repeat` выдаёт успех неограниченное количество раз, что позволяет использовать его для организации циклов. С учётом этого можно, например, распечатать содержимое файла `file.txt` следующим запросом:

```
?- see('file.txt'),
   repeat, get0(X), (X = -1; put(X), false), !,
   seen.
```

Читатель с непривычки может не сразу понять, как здесь работает цикл, но на самом деле всё довольно просто: после чтения очередного символа с помощью `get0` мы попадаем в скобки, где прочитанное значение сравнивается с `-1`; если сравнение успешно, то решатель рассматривает всё выражение в скобках как успешное и идёт дальше вправо, где отсечение отрезает путь

назад. Если же сравнение не прошло, то внутри скобок выполняется вторая ветка (та, что после точки с запятой), которая печатает прочитанный символ и с помощью `false` устраивает неуспех, в результате которого решатель возвращается назад к ближайшей развилке; в роли этой развилки выступает `repeat`, который немедленно выдаёт очередной успех и всё повторяется сначала.

Для переключения вывода аналогичным образом применяются предикаты `tell` (открыть заданный файл на запись и сделать активным потоком вывода) и `told` (закрыть текущий поток вывода, вернуть активную роль стандартному выводу). Кроме того, поток вывода можно открыть на добавление предикатом `append`. Узнать, какой поток активен прямо сейчас, можно с помощью предикатов `seeing` (для ввода) и `telling` (для вывода); это, в частности, позволяет запомнить текущий поток, изменить его, поработать с другим файлом, а затем снова активировать запомненный поток. Например, следующий фрагмент:

```
telling(Save),
append('file.txt'), write('Hello, world\n'), told,
tell(Save),
```

запишет строку `Hello, world` в конец файла `file.txt`, закроет этот файл и восстановит текущий поток.

Основной недостаток всей этой «эдинбургской» механики — использование имён файлов для идентификации потоков; например, открыть один файл дважды не получится, и это может иметь достаточно странные последствия, к примеру, если головная программа откроет некий файл, начнёт с ним работать, после чего обратится к какому-нибудь модулю или библиотеке, а они, в свою очередь, попытаются работать с тем же файлом, не зная, что с ним уже работает головная программа.

Кроме того, предикаты эдинбургского стиля не обошло хорошо известное нам на примере Лиспа родовое проклятие всех «классических» подходов к вводу-выводу — полное отсутствие возможности обработать ошибки самостоятельно, а не отдавать происходящее на откуп интерпретатору. Точнее говоря, конкретно в SWI-Прологе обработать такие ошибки возможно; как водится, для этого потребуется изучить обработку исключений. К исключениям мы вернёмся позже.

Рассмотрим теперь более новый набор примитивов для работы с файлами, основанный на традиционных для POSIX потоках ввода-вывода. Отметим, что SWI-Пролог позволяет для идентификации потоков использовать сами *объекты потоков*, имеющие странный вид при попытке их напечатать (что-нибудь вроде `<stream>(0x128a4d0)`) и не допускающие, что вполне естественно, обратного прочтения с помощью `read`. Кроме того, можно использовать обычные атомы («идентификаторы»), например, `myfile`, `data_source` и т. п. Здесь важно понимать две вещи: во-первых, такие идентификаторы потоков не имеют никакого отношения к именам файлов (в отличие от эдинбургской модели), и, во-вторых, к самим потокам они тоже отношения не имеют, Пролог просто помнит, какой атом связан с каким потоком.

SWI-Пролог сам определяет пять таких атомов: `user_input`, `user_output`, `user_error`, `current_input` и `current_output`. Атомы с префиксом `user_` обозначают хорошо знакомые нам стандартные потоки — во всяком случае, это так на момент старта программы. Атомы `current_input` и `current_output` исходно ассоциируются с теми же потоками, что и атомы `user_input` и `user_output`, но используются иначе: процедуры ввода-вывода, при вызове которых не указан поток в явном виде, работают именно с этими потоками, а «эдинбургские» процедуры `see`, `tell`, `seen`, `told` и примкнувшие к ним «новые» процедуры `set_input`, `set_output` и `with_output_to` *меняют* потоки, ассоциированные с этими атомами. Подчеркнём, что при этом не меняются ни атомы, ни «их значения» (в Прологе у атомов вообще нет значений), ни сами потоки как объекты. Меняется только информация о связи атома с потоком.

Все процедуры ввода-вывода, в том числе рассмотренные нами `get0`, `put`, `write` и `nl`, в Прологе представлены в двух вариантах — без указания потока (именно в таком виде мы их и использовали) и с указанием потока, подобно тому, как в Си есть `printf` и `fprintf`. Имена у «поточных» процедур точно такие же, интерпретатор отличает их по лишнему аргументу, задающему поток; этот аргумент всегда ставится первым. Например,

```
write(user_output, 'Hello, world\n')
```

всегда будет выдавать надпись в поток стандартного вывода, что́ бы мы ни делали с текущим потоком с помощью `tell`/`told` и других средств; ну а

```
write(user_error, 'I\'m in trouble\n')
```

выдаст заданную строку в поток диагностики.

Создать новый поток можно с помощью процедуры `open`, у которой может быть три или четыре параметра. Первый параметр указывает имя файла, причём в этот раз можно использовать любой из трёх вариантов представления строки — атом, список кодов и строковый объект. Кроме того, существуют специальные виды значений, допустимые в роли этого параметра; например, терм `pipe("ls")` означает, что требуется запустить внешнюю программу (в данном случае `ls`), а один из её стандартных потоков перехватить. Какой из двух — зависит от того, на чтение или на запись мы будем открывать поток.

Второй параметр `open` может принимать одно из четырёх значений: `read`, `write`, `append` и `update`. С `read` и `append` всё очевидно, тогда как `update` отличается от `write` тем, что не уничтожает старое содержимое файла, новое содержимое будет записываться поверх старого; `write` всегда начинает запись в файл «с чистого листа».

Третий параметр предназначен для получения результата, то есть собственно открытого потока. Здесь возможно два варианта: указать свободную переменную, тогда с ней будет связан объект потока; либо указать атом, тогда новый поток окажется с этим атомом связан в том же смысле, в каком стандартные потоки связаны с рассмотренными выше атомами `user_*`. Если с данным атомом уже был связан поток, эта старая связь будет разрушена.

Для атомов `user_*` это тоже работает, так что, как видим, даже если некая процедура, подсистема и т. п., например, производит вывод, каждый раз явно указывая, что выводить нужно в `user_output`, ей всё равно можно подменить поток. Для этого, впрочем, открывать новый файл не обязательно, можно вычислить цель

```
set_stream(Stream, alias(user_output))
```

где `Stream` — произвольный поток вывода, и дело сделано.

Четвёртый параметр представляет собой *список опций*; теоретически его можно оставить пустым и вообще не указывать, но мы возьмём на себя смелость порекомендовать всегда указывать хотя бы одну опцию — `encoding(octet)`. Это снизит количество странных сюрпризов. Помните только, что четвёртый параметр `open` — это именно список, так что даже если опция ровно одна, её следует заключить в квадратные скобки, превратив в список из одного элемента.

Закрытие потока производится процедурой `close`; тут всё просто, у неё один параметр — закрываемый поток. Впрочем, без сюрпризов не обошлось и тут: стандартные потоки закрытию не поддаются. Попытка закрыть стандартный ввод попросту тихо игнорируется (то есть не делает вообще ничего), попытка закрыть стандартный вывод или диагностический поток вытесняет информацию из его буфера. Если же вы в какой-то момент изменили потоки, ассоциированные с атомами `user_input`, `user_output` или `user_error`, то такой поток можно закрыть, и интерпретатор вернёт на место поток, бывший «стандартным» изначально (на момент запуска программы). Иначе говоря, реальные (низкоуровневые) потоки с дескрипторами `0`, `1` и `2` Пролог не позволяет ни закрыть, ни подменить — вообще никаким способом, хотя подменить высокоуровневые «стандартные потоки» в том виде, в котором о них знают процедуры Пролога, можно (эта подмена всегда остаётся обратимой).

В целом «новые» средства файлового ввода-вывода пока вроде бы производят приятное впечатление: те, кто это придумывали, явно знали модель POSIX и довольно близко к ней подошли — ну, если не считать бардака с кодировками, который легко победить, но неприятна сама необходимость его «побеждать». Увы, если мы припомним наш опыт с интерпретаторами Лиспа и Scheme, можно будет догадаться, что неприятности у нас впереди — мы пока ещё не пытались обрабатывать ошибки.

Копнув в эту сторону, мы немедленно обнаружим, что Пролог (в том числе SWI-Пролог) содержит подсистему для уже изрядно поднадоевшей нам обработки исключений — и ошибки ввода-вывода можно перехватить только через неё. В принципе это даже до определённой степени логично, ведь если заставить предикаты, связанные с вводом-выводом, генерировать неуспех как показатель ошибки, то информацию о самой ошибке придётся передавать каким-то нетрадиционным способом, поскольку переменные в случае неуспеха не могут получить значения; если «нетрадиционные» (читай — непрологовские) средства всё равно неизбежны, то почему бы не воспользоваться об-

работкой исключений, которая хоть и чужеродна для Пролога, но по крайней мере хорошо известна программистам.

Обработка исключений в SWI-Прологе устроена довольно просто. Для выбрасывания исключения используется встроенная процедура `throw`, имеющая один аргумент — произвольный терм; этот терм выступает в роли объекта исключения. Чтобы выловить и обработать исключение, применяется процедура `catch` с тремя параметрами: первый — цель, во время вычисления которой мы ожидаем возможного возникновения исключительной ситуации, второй — шаблон, с которым будет унифицироваться объект исключения, и третий — цель, вызываемая в качестве обработчика исключения. Если исключение, возбуждённое при попытке вычислить первый аргумент, не может быть унифицировано со вторым аргументом, оно остаётся необработанным — выкидывается выше по стеку вызовов; если же унификация прошла успешно, то переменные, получившие значение в ходе этой унификации, могут быть использованы в третьем аргументе. Простейший способ поймать ошибку открытия файла, не дав интерпретатору обработать её по-своему, выглядит примерно так:

```
catch(open(Fname, read, Stream, [type(binary)]), E, err(E))
```

Здесь подразумевается, что с переменной `Fname` связано имя файла, переменная `E` свободна (так что с ней может быть унифицирован любой объект исключения), а `err` — некая написанная нами процедура обработки.

Объекты исключений, генерируемые встроенными процедурами Пролога, имеют вид терма с главным функтором `error` и арностью `/2`; например, если попытаться открыть на чтение несуществующий файл, получим что-то вроде следующего:

```
error(existence_error(source_sink, 'abrakadabra.txt'),
      context(system:open/3, 'No such file or directory'))
```

Здесь для нас наиболее любопытен главный функтор первого аргумента терма, обозначающий тип ошибки (в данном случае это атом `existence_error`, то есть «ошибка существования»), а также второй аргумент терма, выступающего вторым аргументом объекта исключения — в данном случае строка-атом `'No such file or directory'`. Кроме `existence_error`, нас может также заинтересовать `permission_error` (при открытии файла означает недостаток полномочий) и `io_error` (автору удалось проявить этот вид ошибки, открыв на чтение директорию, причём `open` при этом прошёл успешно, а ошибка вывалилась из `get0`; это не столь удивительно, системный вызов `open` в таких случаях действительно отрабатывает успешно). К сожалению, никакого «официального» списка ошибок на сайте SWI-Пролога не нашлось — возможно, он там где-то и есть, но тщательно закопан, так что ручаться за полноту представленного списка невозможно.

Хорошая новость состоит в том, что сообщение об ошибке, извлечённое из второго аргумента терма-исключения, вроде бы не подвержено влиянию установленной локали — во всяком случае, в версии, имевшейся у автора этих строк. Плохая — в том, что это тоже нигде не документировано. Если в какой-нибудь из будущих версий SWI-Пролога, в том числе той, которую будете использовать вы, читая эту книгу, всё окажется совсем иначе — это никого, увы, уже не удивит.

Для примера попробуем решить ту же задачу, которую мы решали на Лиспе и Scheme — читать текст из стандартного ввода или из заданного в командной строке файла и печатать (всегда в стандартный вывод) длины прочитанных строк. Для простоты картины оформим программу в виде скрипта, начав со строк

```
#!/usr/bin/swipl -q
:- initialization(main).
```

Чтение текста по одному символу реализуем двумя взаимнорекурсивными процедурами: `reading`, состоящая из одного предложения, будет читать символ, на всякий случай убирать развилки (хотя их тут и так быть не должно) и вызывать процедуру `handle`, которая будет разбираться, что там было прочитано (или ничего не прочитано), если вместо кода символа обнаружилась `-1` — прекращать работу, если прочитан конец строки — печатать накопленное значение счётчика, обнулять счётчик и снова вызывать `reading`, если же прочитан любой другой символ — вызывать `reading`, передавая ему значение счётчика, на единицу большее текущего. У обеих процедур будут параметры `Stream` (для передачи потока, из которого читать) и `N` (текущее значение счётчика); процедура `handle` будет дополнительно получать ещё результат работы `get0`. Всё это будет выглядеть примерно так:

```
handle(_, -1, _) :- !.
handle(Stream, 10, N) :- !, write(N), nl, reading(Stream, 0).
handle(Stream, _, N) :- N1 is N+1, reading(Stream, N1).

reading(Stream, N) :- get0(Stream, C), !, handle(Stream, C, N).
```

Обработчик исключений напишем самый простой и тупой: он будет распечатывать атом, обозначающий тип исключения, добавлять к нему извлечённое из объекта сообщение и после этого завершать выполнение программы (здесь мы могли бы сэкономить объём кода, если бы рассказали про форматный вывод, но сам рассказ бы, увы, затянулся). Нам потребуется встроенный предикат `=..`, который позволяет «разложить» терм в список, состоящий из его главного функтора и аргументов (подробнее об этом предикате будет рассказано в следующем параграфе). Текст обработчика будет таким:

```
err(error(E, C)) :- !,
    E =.. [ErrorReason|_], C =.. [_, _, Message],
    write(user_error, ErrorReason),
    write(user_error, ' ('),
    write(user_error, Message),
    write(user_error, ')\n'),
    halt.
err(Ex) :-
    write(user_error, 'Exception unknown: '),
    write(user_error, Ex),
    nl(user_error),
    halt.
```

Имея этот обработчик, напишем две вспомогательные процедуры: одна из них будет открывать файл с заданным именем на чтение, после чего вызывать `reading` с начальным значением счётчика, равным нулю:

```
process_file_on(Fname) :-
    open(Fname, read, Stream, [encoding(octet)]),
    reading(Stream, 0).
```

Вторая будет вызывать первую, «обвесив» её вызов обработчиком исключений:

```
process_file(Fname) :- catch(process_file_on(Fname), Ex, err(Ex)).
```

Осталось только написать функцию `main/0`; если что-то здесь окажется непонятно, перечитайте §11.4.2:

```
main :-
    on_signal(int, _, default),
    prompt(_, ''),
    current_prolog_flag(argv, Argv),
    (Argv = [Fname|_],
        process_file(Fname) ;
        reading(user_input, 0)
    ),
    halt.
```

Полностью текст нашего примера вы найдёте в файле `strlens_file.pl`.

11.4.11. Анализ атомов и термов

Рассмотрим несколько встроенных предикатов, позволяющих добраться до кое-какой информации о термах, которой располагает интерпретатор.

Для начала упомянем предикаты `var/1` и `nonvar/1`. Цель вида `var(X)` будет истинной (её вычисление завершится успехом) тогда и только тогда, когда её единственный аргумент — переменная, и эта переменная в настоящий момент не имеет значения, то есть не является связанной. Здесь стоит сделать замечание, что переменная `X` может быть связана с другой переменной (например, `Y`), а та уже окажется свободной; в этом случае вычисление `var(X)` тоже пройдёт успешно, как, впрочем, и вычисление `var(Y)`. Определяющим тут является то, что унификация вида `X =` *term* (как и `Y =` *term*) в настоящий момент «обречена на успех», каков бы ни был *term*. С другой

стороны, что-нибудь вроде `var(f(X))` никогда истинным не будет, даже если переменная `X` свободна.

Предикат `nonvar/1` по смыслу прямо противоположен: обращение к нему оказывается успешно тогда и только тогда, когда его аргумент представляет собой что угодно, кроме свободной переменной.

Эти два предиката, естественно, не имеют никакого отношения к логике, но при грамотном их использовании позволяют повысить «видимую логичность» вашей программы. Дело в том, что с их помощью можно обеспечить обратимость (инвертируемость) таких процедур, которые никаким иным способом невозможно заставить работать в разных прототипах.

Вернёмся, скажем, к предикату `prefix` (см. стр. 156), на примере которого мы показали, как `is` «убивает» обратимость предикатов. Имея в арсенале `var` и `nonvar`, мы можем легко поправить ситуацию:

```
prefix([], _, 0).
prefix([H|T2], [H|T], N) :-
    nonvar(N), N2 is N - 1, prefix(T2, T, N2).
prefix([H|T2], [H|T], N) :-
    var(N), prefix(T2, T, N2), N is N2 + 1.
```

В таком варианте предикат будет работать во всех осмысленных прототипах — например, его можно попросить построить все возможные префиксы заданного списка:

```
?- prefix(P, [1,2,3], N).
P = [],
N = 0 ;
P = [1],
N = 1 ;
P = [1, 2],
N = 2 ;
P = [1, 2, 3],
N = 3 ;
```

Сложный терм можно построить во время работы программы из отдельных элементов — главного функтора и набора аргументов; можно и декомпозировать терм на отдельные элементы. Обе операции выполняются встроенным предикатом `=..`; этот предикат задаёт *отношение*, в котором находятся сложный терм и *список*, первый элемент которого — главный функтор терма, а остальные элементы — аргументы терма. Так, будет истинной цель

```
loves("George", "Mary") =.. [loves, "George", "Mary"]
```

или, что то же самое,

```
=..(loves("George", "Mary"), [loves, "George", "Mary"])
```

Этот предикат обратим, то есть способен работать «в обе стороны». Так, при вычислении цели `X =.. [a,b,c]` переменная `X` получит значение `a(b,c)`, а вычисление цели `p(1,2,3) =.. Y` свяжет с переменной `Y` значение `[p,1,2,3]`. Иначе говоря, с помощью `=..` можно как синтезировать, так и анализировать сложные термы.

Ещё один полезный встроенный предикат — `name/2` — позволяет конвертировать атомы и числа в их строковые представления и наоборот; под «строковым представлением» здесь подразумевается список кодов. Первым аргументом предиката служит атом или число, вторым — список кодов; этот предикат тоже обратим и может служить как для создания, так и для анализа атомов, а также для перевода чисел в строковое представление и обратно. Например:

```
?- name(vasya, X).
X = [118, 97, 115, 121, 97].

?- name('Anna', X).
X = [65, 110, 110, 97].

?- name(X,  [76, 105, 122, 97]).
X = 'Liza'.

?- name(12321, X).
X = [49, 50, 51, 50, 49].

?- name(X, [55,55,55]).
X = 777.
```

Пролог содержит ряд других встроенных предикатов для преобразования различных типов данных друг в друга. В частности, стоит вспомнить, что современные версии Пролога всё-таки поддерживают строки как отдельную атомарную сущность (см. стр. 146). Для перевода строки из такого представления в список *кодов* и обратно используется предикат `string_codes/2`, для перевода из атома в строку и обратно — `atom_string/2` и т. д. Полный список таких (а также и всех других) встроенных предикатов можно найти в технической документации.

11.4.12. Списки решений

Пусть у нас имеется *недетерминированная* цель, то есть такая цель, которая может дать больше одного решения — но при этом обязательно конечное их количество. Пролог содержит несколько встроенных процедур, позволяющих получить сразу все решения для заданной цели и оформить их в виде списка. Самая простая из этих процедур называется `findall` и предполагает указание трёх параметров — *шаблона*, *цели* и *списка*; первые два считаются входными, третий — выходным, с ним `findall` унифицирует полученный список решений. Со вторым параметром вроде бы тоже всё понятно — это та самая цель, к которой нужно обратиться столько раз, сколько она будет выдавать новых решений.

Сложнее всего здесь понять, зачем нужен *первый* параметр. Обычно в литературе его роль пытаются объяснить примерно такими заклинаниями: «он задаёт внешний вид результатов, включаемых в список результатов», «после получения каждого решения на основе полученных связей переменных из шаблона строится терм» и т. п. Если вы ничего не поняли, ничего страшного: не вы первые, не вы последние. Попытаемся объяснить происходящее на примерах.

Простейший вариант первого аргумента — это попросту та переменная (единственная), возможные значения которой нас интересуют. Для иллюстрации напишем простенький одноместный предикат, состоящий только из фактов:

```
color(red).
color(green).
color(blue).
```

Запрос `color(X)` выдаст, естественно, три решения:

```
?- color(X).
X = red ;
X = green ;
X = blue.
```

Чтобы построить *список* из них, можно дать такой запрос:

```
?- findall(X, color(X), Result).
Result = [red, green, blue].
```

В роли шаблона тут выступает переменная `X` — единственная свободная переменная в искомой цели. Можно из найденных решений сделать что-нибудь посложнее, применив шаблон в виде сложного терма, например:

```
?- findall([X], color(X), Res).
Res = [[red], [green], [blue]].

?- findall(f(X), color(X), Res).
Res = [f(red), f(green), f(blue)].

?- findall(X+X, color(X), Res).
Res = [red+red, green+green, blue+blue].

?- T=light, findall(T/X, color(X), Res).
T = light,
Res = [light/red, light/green, light/blue].
```

Можно сделать и так (непонятно зачем):

```
?- findall(abrakadabra, color(X), Res).
Res = [abrakadabra, abrakadabra, abrakadabra].
```

Для чего в действительности используется шаблон в `findall`, можно легко понять, если вспомнить, что *решение* для запроса может состоять из *более чем одной связи переменной со значением*. Для иллюстрации вернёмся к предикату `ancestor`, описанному на стр. 150. Пусть нам зачем-то понадобилось найти все пары «предок-потомок», присутствующие в системе и при этом не находящиеся в отношении `parent` — иначе говоря, нас интересуют все пары «дед-внук», «дед-правнук» и пр. Построить *список* таких пар можно с помощью `findall`, осталось только придумать, в каком виде мы хотим получить каждую из найденных пар. Например, можно построить список *списков из двух элементов*:

```
?- findall([A, D], (ancestor(A, D), not(parent(A, D))), L).
L = [["Lucas", "Fred"], ["Lucas", "Jane"], ["Lucas", "Sean"], [
"Lucas", "Jessica"], ["Lucas", "Hannah"], ["Lucas", "Joseph"],
["Lucas", "John"], ["Lucas"|...], [...|...]|...].
```

А можно использовать какой-нибудь инфиксный *оператор*:

```
?- findall(A/D, (ancestor(A, D), not(parent(A, D))), L).
L = ["Lucas"/"Fred", "Lucas"/"Jane", "Lucas"/"Sean", "Lucas"/
"Jessica", "Lucas"/"Hannah", "Lucas"/"Joseph", "Lucas"/"John"
, "Lucas"/"Laura", ... / ...|...].
```

К сожалению, в обоих случаях Пролог счёл значение L слишком длинным, чтобы печатать его целиком, и благополучно обрезал списки. Впрочем, всё в наших руках:

```
?- findall(A/D, (ancestor(A,D),not(parent(A,D))), L), write(L), nl, abort.
[Lucas/Fred,Lucas/Jane,Lucas/Sean,Lucas/Jessica,Lucas/Hannah,Lucas/Joseph,
Lucas/John,Lucas/Laura,Jason/Joseph,Jason/John,Jason/Laura]
```

Здесь мы потребовали от Пролога выполнить запрос к `findall`, напечатать полученное значение L, перевести строку (физически печатаемое значение попало на экран только в этот момент) и после этого прекратить вычисление запроса — чтобы после списка, который уже напечатан, Пролог не печатал ещё и полученное решение для всего запроса (как оно выглядит, мы уже видели).

Следующей мы рассмотрим встроенную процедуру `bagof`. Так же как и `findall`, `bagof` принимает три аргумента — шаблон, цель и результат. Более того, если все свободные переменные, присутствующие в цели, также присутствуют и в шаблоне, `bagof` работает точно так же, как `findall`; различие между ними проявляется, лишь когда какая-то из переменных цели в шаблон не включена. Вернёмся к нашим внукам-правнукам и заменим `findall` на `bagof`. Если это будет единственным отличием нового запроса, то результат получится ровно тот же самый, что и раньше, но вот если мы из шаблона уберём переменную, соответствующую предку, и оставим только потомка, получим вот что:

```
?- bagof(D, (ancestor(A, D), not(parent(A, D))), L).
A = "Jason",
L = ["Joseph", "John", "Laura"] ;
A = "Lucas",
L = ["Fred", "Jane", "Sean", "Jessica", "Hannah", "Joseph",
"John", "Laura"].
```

Как видим, решений теперь два — по одному на каждого предка, у которого в принципе есть хотя бы один искомый потомок (внук, правнук и т. д.). Общее правило такое: `bagof` формирует отдельное решение для каждой возможной комбинации значений свободных переменных, входящих в цель, но не входящих в шаблон.

Ещё одна полезная встроенная процедура называется `setof` и отличается от `bagof` тем, что список результатов в каждом полученном решении она сортирует и убирает повторяющиеся элементы.

11.4.13. Работа с базой данных

Исходно под ***базой данных*** в Прологе понималось множество всех предикатов, определение которых состояло исключительно из ***фактов***, то есть предложений, состоящих из одной «головы» и не содержащих переменных. В ныне существующих версиях Пролога, включая SWI, базой данных считаются предикаты, явно или неявно объявленные как ***динамические***. Для явного объявления используется *директива* `dynamic/1`; например, если мы хотим во время исполнения программы менять в ней информацию из наших «генеалогических» примеров, стоит (хотя и не совсем обязательно, но об этом позже) предусмотреть в тексте примерно такую строчку:

```
:- dynamic parent/2, male/1, female/1.
```

Во время работы программы можно добавлять в базу данных новые факты и изымать из неё имеющиеся. Добавление производится встроенными процедурами `assert`, `asserta` и `assertz`. Чтобы понять, в чём состоит отличие между ними, нужно вспомнить, что факты (и вообще предложения), входящие в предикат, с точки зрения Пролога *упорядочены*, и от этого порядка может зависеть получаемое решение, причём в особо тяжёлых случаях вопрос состоит не в том, каков будет порядок выдаваемых решений для какого-нибудь запроса, а в том, например, будет ли решение вообще найдено или решатель уйдёт в бесконечную рекурсию. Так вот, `asserta` добавляет новый факт в начало их цепочки, то есть *перед* всеми другими фактами того же предиката; `assertz`, наоборот, добавляет в конец; ну а для обычного `assert` позиция, в которую будет добавлен факт, не специфицирована (хотя на самом деле в SWI-Прологе `assert` работает точно так же, как `assertz`).

Удаление фактов из базы данных производится процедурами `retract` и `retractall`. С `retractall` всё просто: параметром ей служит *сложный терм*, возможно, содержащий свободные переменные (но в этом случае обычно используется *анонимная переменная*); из базы данных при этом удаляются *все факты, которые можно унифицировать с данным термом*. Например, `retractall(parent("Jessica", _))` уберёт из отношения `parent` сведения обо всех детях Джессики, а `retractall(male("John"))` уберёт запись о том, что Джон является мужчиной. Если указать `_` в качестве всех аргументов терма, соответствующее *отношение* (если угодно, таблица) базы данных опустеет. Стоит отметить, что обращение к `retractall` всегда успешно, даже если не удалено ни одного факта.

Понять, как работает простой `retract`, несколько сложнее. Смысл его параметра точно такой же, как и у `retractall`, то есть ему передаётся терм и он удаляет из базы данных факты, которые с этим термом унифицируются; но делает он это *по одному* факту, после каждого успешного удаления `retract` выдаёт успех, а свободные переменные из его параметра оказываются связаны со значениями, полученными при унификации. Когда фактов больше не остаётся, `retract` завершается неуспешно. Например:

```
?- assert(color(red)), assert(color(green)), assert(color(blue)).
true.

?- retract(color(X)).
X = red ;
X = green ;
X = blue.
```

Отметим один интересный момент. Если в качестве аргумента любому из вариантов `assert` или процедуре `retractall` передать терм с таким главным функтором и арностью, которым в базе данных (и вообще в среде выполнения) не соответствует ни один факт, то есть такого предиката попросту пока что нет, то он будет создан и объявлен динамическим. Таким образом, если ваша программа сама по себе (в своём тексте) не вносит в отношение базы данных ни одного факта, предполагая, что это произойдёт уже во время исполнения, то директиву `dynamic` указывать не обязательно. Впрочем, лучше её всё-таки написать, чтобы показать будущим читателям программы, что́ мы имели в виду.

Нынешние реализации Пролога позволяют делать динамическими (и модифицировать во время исполнения) любые процедуры, а не только предикаты, состоящие только из фактов. Возьмём на себя смелость настоятельно рекомендовать не пользоваться этой возможностью. Не стоит путать обрабатываемые данные с правилами, по которым производится эта обработка; такие вещи обычно плохо кончаются. Например, если не очень опытный программист напишет программу так, что она будет создавать динамические процедуры (состоящие из сложных предложений) на основании информации, прочитанной из внешних источников (скажем, из файла, который кто-то прислал через Интернет), то эту программу можно будеть заставить *делать что угодно*, т. е. она будет заведомо уязвима к атаке, которую обычно называют *code injection*.

По этой же причине не следует использовать для чтения файлов с данными ни `consult`, ни `read`, да и представлять данные (в противоположность программам) в виде текста на Прологе тоже не стоит.

Базу данных в принципе можно использовать для имитации глобальных переменных. Например, можно предусмотреть динамический предикат `global_vars` с двумя аргументами:

```
:- dynamic global_vars/2.
```

Для присваивания теперь можно будет написать такую процедуру:

```
assign_var(Var, Value) :-
    retractall(global_vars(Var, _)),
    asserta(global_vars(Var, Value)).
```

Ну а обращаться к переменной можно будет напрямую через предикат `global_vars` примерно так:

```
..., global_vars(myvariable, X), ...
```

Прежде чем делать что-то подобное, хорошо подумайте. Программирование на Прологе полезно лишь тогда, когда программист умеет и хочет применять логическую парадигму, тогда как переменные, к тому же ещё и глобальные — это императивщина в одном из худших её проявлений.

11.4.14. Оформление кода на Прологе

Пролог можно считать характерным представителем довольно многочисленного семейства языков программирования, в которых программа состоит из *предложений*, каждое из которых имеет заголовок[31] и некое тело, возможно, пустое; тело отделяется от заголовка каким-то специальным знаком (в Прологе это «:-»). Предложения, заголовки которых в каком-то смысле «подобны» (для Пролога это означает совпадающие главный функтор и арность), формируют подпрограммы, будь то «процедуры» (предикаты) в Прологе или функции в том же Хаскеле и Эрланге; они как раз и становятся главными «боевыми единицами» программы в целом. Позже мы рассмотрим язык Хоуп, в котором функции тоже строятся по этому принципу, хотя (в отличие от Пролога) всё-таки предполагают какое-то подобие заголовка; но на практике читатель вполне может столкнуться с другими языками, в которых текст состоит из предложений. Подход к оформлению кода на языках, использующих синтаксис этого типа, имеет определённые особенности.

Пока предложение умещается в одной строке, его часто так в одну строку и записывают:

```
no_attack(_, []).
no_attack(Q, [X|L]) :- not(attacks(Q, X)), no_attack(Q, L).
```

— хотя в некоторых руководствах и примерах применяется другой стиль, предполагающий, что на одной строке с заголовком предложения не пишется больше ничего, даже если правая часть предложения состоит всего из одной «цели»; правая часть всегда записывается, начиная со следующей строки, со сдвигом вправо, причём обычно такой сдвиг выбирают равным табуляции:

```
queens(N, Result) :-
        do_queens(N, 0, [], Result).
```

В этом случае остаётся вопрос, как расположить цели правой части, если их больше одной, но они, в принципе, умещаются в одну строку. Допустима как запись «в столбик»:

```
no_attack(Q, [X|L]) :-
        not(attacks(Q, X)),
        no_attack(Q, L).
```

— так и запись «в линейку»:

```
no_attack(Q, [X|L]) :-
        not(attacks(Q, X)), no_attack(Q, L).
```

Какую из них предпочесть, решайте сами; как обычно, единственной рекомендацией здесь будет придерживаться какого-то одного стиля во всей программе. Если же предложение в одну строку не умещается, то остаётся фактически только один вариант, а именно — записать его «столбиком»:

```
sameset([], []).
sameset([X|L], M) :-
        member(X, M),
        delete_all(X, M, M1),
        delete_all(X, L, L1),
        sameset(L1, M1).
```

Многие авторы программ на Прологе делают одну и ту же характерную ошибку при оформлении таких многострочных предложений, а именно — записывают первую цель правой части в одной строке с заголовком, а последующие цели размещают точно под первой. Получается примерно так:

```
sameset([], []).
sameset([X|L], M) :- member(X, M),
                     delete_all(X, M, M1),
                     delete_all(X, L, L1),
                     sameset(L1, M1).
```

Недостаток такого решения мы уже неоднократно обсуждали: размер отступа оказывается зависящим от неких обстоятельств, в данном случае — от длины заголовка предложения.

Ситуацию может сильно осложнить появление в предложениях точки с запятой (напомним, она изображает в Прологе дизъюнкцию). Если предложение состоит из нескольких альтернатив, разделённых точкой с запятой, и каждая альтернатива умещается в строку, проще всего так и писать по одной альтернативе в строке:

```
lexic_process_char(Ch, Line, Newstate, Sofar, R) :-
        isspace(Ch), !, R = Sofar, Newstate = home;
        [Ch] = ",",  !, R = [comma|Sofar], Newstate = home;
        [Ch] = ":",  !, R = [colon|Sofar], Newstate = home;
        [Ch] = ";",  !, R = [semicolon|Sofar], Newstate = home;
        [Ch] = ".",  !, R = Sofar, Newstate = stop;
        isalnum(Ch), !, R = [[Ch] | Sofar], Newstate = id;
        !, write('lexical error at line '), write(Line), nl, fail.
```

Если в аналогичной ситуации некоторые альтернативы в строку не помещаются, можно посоветовать что-то вроде следующего:

```
lexic_process_char(Ch, Line, Newstate, Sofar, R) :-
        isspace(Ch), !,
                R = Sofar,
                Newstate = home
        ;
        [Ch] = ",", !,
                R = [comma|Sofar],
                Newstate = home
        ;
        [Ch] = ":", !,
                R = [colon|Sofar],
                Newstate = home
        ;
        % . . .
```

Здесь мы воспользовались тем, что в примере сопоставление вместе с отсечением представляет собой что-то вроде «заголовка» для альтернативы, поскольку задаёт условие, при котором данная альтернатива будет исполнена, а остальные исполнены не будут. Ну а для более сложных случаев, в особенности при использовании скобок для группировки целей, можно предложить следовать общим принципам оформления вложенных конструкций: начало и конец конструкции размещать строго друг над другом, а всё вложенное — сдвигать вправо на постоянный размер сдвига.

[31] В английском оригинале — просто *head*, то есть «голова».

11.5. Ленивые вычисления

11.5.1. Две возможные стратегии вычисления выражений

Все языки программирования, которые мы рассматривали до сих пор, объединяет ***энергичная стратегия вычислений*** (англ. *eager evaluation*)[32]: если результат вычисления выражения должен быть связан с переменной, в том числе с формальным параметром при вызове функции или подпрограммы, то выражение немедленно вычисляется. «Энергичную» стратегию часто называют также «строгой» (англ. *strict*); в англоязычной литературе можно встретить ещё термин *greedy* (буквальный перевод — «прожорливый»). Мы настолько привыкли к такому подходу к вычислениям, что обычно даже не задумываемся о возможных альтернативах.

Конечно, для классических фоннеймановских языков программирования, таких как Си или Паскаль, где переменная есть область памяти фиксированного размера, способная хранить значение заданного типа и только, никакой альтернативы немедленному вычислению, собственно говоря, нет. Иное дело — языки более высокого уровня. Строгая типизация переменных здесь может отсутствовать, как мы это видели в лиспах и Прологе, но даже если типизация есть (а во многих языках «очень высокого» уровня переменные строго типизированы), то она имеет отношение скорее к семантике программы, *видимой программисту*, нежели к тому, как переменные и их значения будут представлены на уровне машинного кода. Иначе говоря, если не ограничивать уровень языка явной работой с фоннеймановской памятью, то одно и то же «видимое» значение, пусть и фиксированного типа, можно хранить разными способами, например, в разных представлениях, ведь размером памяти, физически отведённым под переменную, мы здесь не ограничены. Ну а в число возможных представлений одного и того же значения входит, как можно внезапно догадаться, *невычисленное (пока) выражение, результатом которого станет искомое значение.*

Именно это соображение открывает путь к ***ленивым вычислениям***. Суть их в том, что любое значение, которое должно получиться как результат вычисления выражения, так и хранится *в виде выражения*, пока в какой-то момент в вычислениях это значение не потребуется в явном виде. Лишь тогда хранящееся в памяти выражение наконец вычисляется и заменяется собственным результатом.

Самое очевидное достоинство ленивой стратегии вычислений состоит в том, что некоторые выражения вообще не будут вычислены *никогда*. Например, некая функция (назовём её `f`) может принимать два параметра, но первый использовать всегда, а второй — только когда первый параметр равен нулю:

```
double f(double x, double y)
{
    return x == 0 ? 2*y : x/2;
}
```

Представьте себе теперь выражение вроде `f(1, g(a, b))`, причём, чтобы усилить эффект от примера, предположите, что функция `g` — какая-нибудь сложная и вычисляется очень долго. Конечно, если мы имеем дело с языком Си, функция `g` будет вычислена, от этого никуда не деться, но вот если аналогичная программа будет написана на каком-нибудь из языков с ленивой стратегией вычислений, то до реального вызова функции `g` дело так и не дойдёт: с формальным параметром `y` функции `f` будет связано значение в виде невычисленного выражения `g(a, b)`, в теле функции `f` этим выражением никто не воспользуется, так что оно благополучно исчезнет вместе с самой переменной `y` в момент окончания работы функции `f`.

Можно сделать наш пример ещё более убедительным, если представить, что функция `g` время от времени (например, при определённых значениях параметров) вообще не завершает свою работу — например, зацикливается или уходит в бесконечную рекурсию. При энергичных вычислениях та же участь постигнет и всю нашу программу, тогда как при вычислениях ленивых ничего страшного не произойдёт.

Часто можно встретить утверждение, что якобы ленивая стратегия вычислений *ускоряет* работу программ в целом за счёт отказа от вычисления ненужных значений. В действительности это, мягко говоря, преувеличение. С одной стороны, лишних вычислений всегда можно избежать путём грамотного написания программы; так, в нашем примере достаточно было бы функцию `f` заменить макросом с тем же именем, а при условии поддержки компилятором определённых видов оптимизации — даже не заменять её, а объявить как подставляемую (`inline`). С другой стороны, и это намного важнее, *хранение невычисленных выражений само по себе обходится достаточно дорого*, чтобы в большинстве случаев свести на нет любую экономию и даже, скорее всего, *перевесить* её. Но ленивые вычисления интересны, конечно же, совсем не скоростью работы.

Пожалуй, одно из самых удивительных свойств ленивых вычислений — это их способность работать с *бесконечными* структурами данных. Физически при этом часть обрабатываемой структуры данных — например, бесконечного списка — хранится в виде невычисленного выражения. По мере обращения к элементам такого списка вычисления продвигаются всё дальше и дальше, так что «упереться» в конец списка оказывается невозможно: сколько бы элементов мы ни запросили, они все окажутся существующими.

Одна из фундаментальных особенностей ленивой стратегии — её принципиальная несовместимость с ***побочными эффектами*** и, как следствие, с модифицирующими действиями как таковыми, ведь в функциональном программировании модифицирующее действие может быть только побочным эффектом. В самом деле, в той точке программы, где записано некое выражение, в общем случае неизвестно (и не может быть известно!), *когда* это выражение будет вычислено и будет ли оно вычислено вообще. Если допустить в выражениях побочные эффекты, то последовательность, в которой соответствующие эффекты проявятся, будет невозможно предсказать, причём некоторые из ожидаемых побочных эффектов вообще не произойдут. Представьте себе, к примеру, программу, которая вроде бы должна вывести на экран несколько фраз, но после запуска часть фраз не появляется на экране совсем, а остальные выводятся в самом неожиданном порядке, так что итоговый текст оказывается совершенно хаотичен. Вопрос о возможности использования такой программы кажется риторическим.

11.5.2. Хоуп и другие «ленивые» языки

Запрет на любые модифицирующие действия — *настоящий* запрет, а не то, что мы наблюдали в других языках («если очень надо, то можно») — порождает вполне очевидные проблемы. Обходиться без глобальных переменных, присваиваний, циклов и прочей императивщины мы, можно надеяться, уже научились и даже, возможно, вошли во вкус. В программах, чья задача — из исходных данных, полностью доступных на момент старта программы, получить некие результирующие данные — тоже можно всё сделать «функционально», например, одним движением выдать все результаты, построенные вызванной функцией, как мы сделали в §9.1.3 (см. пример на стр. 14). Чего точно не выйдет — так это написать без традиционного ввода-вывода программу *интерактивную*, то есть такую, работа которой зависит от входных данных, возникающих в ответ на её собственные действия; эту проблему мы уже упоминали — см. замечание на стр. 14. В «ленивых» языках с этим приходится как-то справляться, изобретая весьма изощрённые концепции; возможно, отчасти поэтому «ленивое» программирование не столь популярно, как можно было бы ожидать. Из реально используемых в наше время языков программирования фактически только один — Хаскель — обладает «ленивой» семантикой.

К сожалению, с этим языком есть одна проблема: он настолько сложен, что большинство программистов, пишущих на нём (в том числе за деньги), начинают это делать, не понимая толком, что́ на самом деле делают, и лишь по прошествии заметного времени достигают понимания происходящего; зачастую на это уходит полтора-два года. Если начать описывать Хаскель так, как это обычно делается, поначалу этот язык будет казаться чуть ли не императивным, поскольку то, что там происходит на самом деле, окажется скрыто за синтаксическим сахаром, который, похоже, специально сделан так, чтобы функциональной сути не было видно. Рассказать о Хаскеле кратко, но при этом так, чтобы продемонстрировать его природу с точки зрения парадигм программирования — судя по всему, задача невыполнимая[33].

Для иллюстрации возможностей ленивых вычислений мы воспользуемся другим языком программирования — далёким предком Хаскеля, который

[32] Английское слово *eager* может быть переведено как «энергичный» только в довольно специфических контекстах, а в применении к вычислению выражений было бы, наверное, правильно использовать другой перевод — «нетерпеливый», это лучше передаёт смысл термина.

[33] Автор не утверждает, что сие физически невозможно, факт пока лишь в том, что сделать такое введение в Хаскель не может ни сам автор, ни кто-либо из тех, с чьими текстами по этому языку автору доводилось сталкиваться. Возможно, кому-нибудь такой рассказ о Хаскеле удастся в будущем.

называется Хоуп (*Hope*[34]). Сейчас этот язык полностью вышел из употребления, было достаточно сложно найти его работающую реализацию. С другой стороны, именно этот язык использовался в качестве иллюстративного в известной монографии Филда и Харрисона, посвящённой функциональному программированию [24]; сам язык достаточно миниатюрен, так что его описание много места не займёт, и при этом он позволит продемонстрировать два основных свойства всего семейства языков, от него произошедших — во-первых, ленивую стратегию вычислений, и, во-вторых, строгую типизацию вкупе с автоматическим выводом типов.

Отметим, что теоретически существуют ещё два ленивых языка — Миранда (Miranda) и Клин (Clean). К сожалению, единственная реализация Миранды до второй половины 2019 года оставалась проприетарной, что не лучшим образом сказалось на её популярности; вкладывать своё время в изучение инструмента, находящегося в чьей-то собственности, несколько странно, так что до недавнего времени об этом языке многие слышали, но никто толком не видел, когда же реализацию всё-таки решили открыть, было уже поздно. Что касается Клина, то из него, возможно, что-то и получится; в освоении он всё же проще, чем Хаскель. Тем не менее, для наших учебных целей он слишком сложен, так что мы остановимся на Хоупе.

11.5.3. Интерпретатор Hopeless

Исходная версия реализации, которой мы воспользуемся, была написана Россом Патерсоном (*Ross Paterson*); судя по тому, что все материалы по языку Хоуп были с его личной страницы удалены (если верить архивам сайта `web.archive.org`, в 2013 году), автор реализации потерял к ней интерес. Некоторое время реализацию пытался поддерживать Александр Шабаршин, он как раз и придумал название *Hopeless* — но к настоящему моменту его сайт, посвящённый Хоупу, исчез, доменное имя досталось киберсквоттерам, страница на GitHub, содержавшая более свежие версии исходных текстов, тоже удалена, хотя можно найти некоторое количество клонов на том же GitHub.

При написании этой книги автор предпочёл создать свою страничку, посвящённую всё той же реализации — есть шанс, что эта страничка хотя бы не исчезнет, как остальные. В исходники внесены незначительные правки, в частности, добавлена возможность производить чтение из стандартного потока ввода, указав вместо имени файла пустую строку (а не `/dev/stdin`, как предлагалось в исходной реализации). Страница доступна по адресу `http://www.stolyarov.info/misc/hopeless`. Скачав по ссылке с этой страницы архив исходных текстов, распакуйте его командой

```
tar -xzf hopeless06avst002.tgz
```

— после чего зайдите в появившуюся директорию `hopeless` и дайте команду `make`. **Делать это настоятельно рекомендуется в сеансе работы обычного пользователя, не администратора!** Если всё собралось, зайдите в систему с правами суперпользователя, зайдите в директорию, где собирали интерпретатор, и дайте команду `make install`. Интерпретатор будет установлен в директорию `/usr/local/bin`, man-страничка для него — в `/usr/local/man/man1`, а библиотечные файлы — в `/usr/local/share/hopeless/` (эта информация пригодится, если вы захотите убрать интерпретатор из своей системы). Если что-то пойдёт не так, свяжитесь с автором.

На момент написания книги последняя версия называлась `hopeless06avst002.tgz`, но к тому времени, когда вы эту книгу будете читать, номер актуальной версии может возрасти.

Программы на Хоупе проще всего оформлять хорошо знакомым нам способом — в виде скриптов; для этого (точнее, вообще для чтения программы из файла) интерпретатор поддерживает флаг `-f`. Например, программа, печатающая свои аргументы командной строки, будет выглядеть так:

```
#!/usr/local/bin/hopeless -f
write argv;
```

Флаг `-f` в действительности означает «читать текст программы из файла», а имя вашего скрипта, как мы знаем, операционная система подставит автоматически при запуске. Интерпретатор можно запустить и без этого флага — в режиме интерактивного диалога с пользователем, который полезен при отладке, а также при изучении языка Хоуп, чтобы можно было сразу же попробовать небольшие функции. Средствами редактирования строк и хранения истории интерпретатор не оснащён, так что запускать его рекомендуется через утилиту `rlwrap`.

Приглашение интерпретатора к вводу состоит из двух символов `>:`. Стоит сразу же запомнить, что любое предложение Хоупа завершается точкой с запятой, так что, пока вы её не введёте, интерпретатор будет считать, что вы ещё не закончили ввод. Если вы нажали Enter, но ничего не произошло и, самое главное, не появилось очередное приглашение к вводу — скорее всего, вы просто забыли про точку с запятой; это не страшно, просто введите её и нажмите Enter ещё раз.

Единственным исключением из этого правила является почему-то команда `edit`, о которой речь пойдёт ниже. Помнить это не обязательно, точку с запятой можно ввести и тут.

Интерпретатор ожидает от вас ввода команд, в число которых входит любое выражение — в этом случае выражение будет вычислено, а для полученного результата интерпретатор напечатает значение и тип, предварённые зачем-то символом `>>`:

```
>: 2 + 3;
>> 5 : num
```

Находясь в интерпретаторе, можно описывать глобальные имена (константы и функции) с помощью команд `dec` и `---` (всё правильно, три тире — это тоже команда), например:

```
>: dec cube : num -> num;
>: --- cube x <= x * x * x;
>: cube 5;
>> 125 : num
>: cube 10;
>> 1000 : num
```

Подробности о том, как в Хоупе описываются функции, вы найдёте в §11.5.6.

Если у вас уже есть файл с функциями на Хоупе, вы можете загрузить его из интерпретатора командой `use`, параметром которой служит *имя модуля*. К сожалению, здесь наш интерпретатор проявляет довольно неприятную особенность: к имени модуля он добавляет суффикс `.hop` и пытается найти файл с таким именем в текущей директории. Заставить его прочитать файл с другим суффиксом и/или из другой директории невозможно, или, во всяком случае, ваш покорный слуга не понял как.

Следующая возможность интерпретатора на первый взгляд выглядит довольно странно, но, надо признать, оказывается удобна. Команда `edit` позволяет отредактировать исходный файл, написанный на Хоупе, не выходя при этом из интерпретатора; правила и ограничения тут те же, что и для команды `use` из предыдущего параграфа. Интерпретатор запускает текстовый редактор, указанный в переменной окружения `EDITOR`, а если этой переменной в окружении нет — редактор `vi`. Если «модуль», который вы открыли на редактирование, был подключён командой `use`, то после выхода из редактора интерпретатор обновит версии входящих в него функций.

Есть тут, впрочем, и определённое неудобство. Если вы при редактировании сделали ошибку, то интерпретатор, обнаружив это, вернёт вас обратно в редактор, где в тексте вы найдёте строки, начинающиеся с символа `@`, содержащие соответствующую диагностику. Идея, быть может, исходно неплохая, вот только теперь вам не удастся так просто уйти — интерпретатор будет раз за разом возвращать вас в редактор, пока вы не исправите все ошибки. Альтернативой этому остаётся разве что прибить интерпретатор командой `kill` из другого окошка. Если этого делать не хочется, но и исправлять ошибку прямо сейчас вы по каким-то причинам не хотите, можно попробовать «закомментарить» ошибочный фрагмент программы, поставив перед каждой строчкой символ «`!`».

Между прочим, то же самое произойдёт, если ошибки будут обнаружены в файле, который вы потребовали загрузить командой `use` — вы окажетесь в редакторе и интерпретатор вас оттуда не выпустит, пока вы не исправите ошибки.

Команда `display` позволяет увидеть все имена (функции и константы), которые вы ввели за время текущего сеанса, вместе с их описаниями; на объ-

[34] Слово *hope* в переводе с английского значит «надежда», но, если верить Википедии, в данном случае язык получил своё имя в честь площади Hope Park Square в Эдинбурге, а она в свою очередь названа по имени шотландского аристократа Томаса Хоупа, жившего в XVIII веке.

екты, загруженные из файлов с помощью `use`, это не распространяется — интерпретатор только укажет, что соответствующий модуль был подключён (собственно говоря, напечатает директиву `use` в таком виде, в котором её надо дать, чтобы подключить модуль). Команда `save` позволяет результаты своей работы (попросту говоря, всё, что показывает `display`) сохранить в заданном файле (суффикс `.hop` указывать не надо, интерпретатор его добавит автоматически). Есть и другие команды, но рассказ о них мы опустим.

К сожалению, интерпретатор Hopeless — откровенно экспериментальный, и из параграфа, посвящённого вводу-выводу, это будет хорошо видно. Увы, другого интерпретатора Хоупа у нас нет, а проиллюстрировать ленивые вычисления чем-то всё-таки надо.

11.5.4. Лексика и синтаксис языка Хоуп

Программа на Хоупе состоит из *предложений*, каждое из которых начинается с *имени директивы* (если угодно, команды) и заканчивается точкой с запятой. Предложение может занимать произвольное количество строк, хотя чаще всего его стараются уместить в одну строку, если, конечно, это не приводит к выходу за границу допустимой ширины текста (напомним, что это 80 символов; как водится, интерпретатору совершенно всё равно, какой ширины будет наш текст, но для нас важно, чтобы его, помимо прочего, было легко читать).

Лексический анализатор работает здесь довольно просто. Комментарии начинаются с восклицательного знака (при условии, что он не был внутри апострофов и кавычек) и продолжаются до конца строки. Символов-разделителей тут всего шесть: запятая, точка с запятой, круглые и квадратные скобки. Кроме них интерпретатор выделяет в качестве лексем числовые, символьные и строковые литералы (практически по правилам Си — символы в одиночных апострофах, строки в двойных кавычках, всевозможные `\n`, `\t` и прочие тоже работают). Наконец, *слова* (как ключевые, так и идентификаторы) могут относиться к одному из двух видов. Слова первого вида «почти сишные» — начинаются с латинской буквы или знака подчёркивания, дальше может быть любая последовательность букв, цифр, тех же подчёркиваний, плюс в конце можно добавить один или больше апострофов — «штрихов», получив что-то вроде `x'` или `light12'''`. Слова второго вида выглядят несколько неожиданно: в них могут входить любые символы, *кроме* пробелов, букв, цифр, подчёркивания, восклицательного знака, кавычек и апострофов. Примерами таких слов могут служить знаки арифметических операций и сравнений, но также и всякие экзотические комбинации вроде `$$$`, `^|^`, `-=#=-`, `%&@#$` и прочее в таком духе. Поскольку слова первого и второго типа не пересекаются по допустимым символам, их можно не разделять пробелами — интерпретатор сам знает, как их отделить друг от друга, а слова второго типа — ещё и от числовых литералов. Поэтому, например, `a+b-25` можно написать без пробелов.

Следующие слова являются ключевыми и используются имеющейся версией интерпретатора:

```
++ --- : <= == => |
abstype data dec display else edit exit if in infix infixr
lambda let letrec private save then type typevar uses
where whererec write
```

Кроме того, слова

```
end module nonop pubconst pubfun pubtype
```

зарезервированы, но не используются; их особый статус оправдывается совместимостью с какими-то другими реализациями Хоупа, которые к нашему времени давно и прочно ушли в прошлое. Наконец, слова `\`, `use` и `infixrl` тоже зарезервированы и считаются синонимами соответственно для `lambda`, `uses` и `infixr`.

С помощью директив `infix` и `infixr` можно любые слова, не являющиеся ключевыми, объявить «инфиксными операциями» и формировать с их помощью выражения, подобные арифметическим; подробно мы этот момент рассматривать не будем.

Арифметические операции и операции сравнения

```
+  -  *  /  mod  div  =  /=  <  =<  >  >=
```

(и другие) вводятся как имена библиотечных функций, то есть ключевыми словами не являются — например, можно их использовать как имена локальных переменных. Отметим один момент: **унарного минуса в Хоупе нет** (как, разумеется, и унарного плюса), приходится писать что-то вроде `0 - z`.

Как ни странно, не являются ключевыми также и имена встроенных типов, и многие другие слова, вводимые интерпретатором в глобальной области видимости.

11.5.5. Модель данных и система типов

Хоуп — язык строго типизированный; это означает, что функции принимают на вход параметры строго определённых типов, списки состоят из значений одного и того же типа и т. д. Как обычно в таких случаях, система типов основывается на нескольких (в данном случае трёх) ***встроенных типах***, из которых можно конструировать типы более сложные — ***пользовательские***.

Роль встроенных типов в Хоупе играют `num` — число (в имеющейся у нас реализации для чисел используется представление, известное в Си как тип `double`), `truval` — логическое значение `true` или `false`, и `char` — простой однобайтовый символ (как и в большинстве других языков, в Хоупе константы этого типа записываются в апострофах).

Версия Хоупа, описываемая в упоминавшейся выше монографии [24], содержала ещё тип `real`, а тип `num` ограничивала целыми числами.

Одним из основных видов пользовательских типов выступает ***кортеж***, по своей сути похожий на паскалевскую *запись* или *структуру* из Си: это упорядоченный набор заданного количества значений, каждое из которых имеет свой (опять же заданный) тип. Сами кортежи записываются в виде последовательности значений, заключённых в круглые скобки и разделённых запятыми, а тип кортежа задаётся перечислением типов его элементов через символ `#`. Например, `(25, 36, 49)` — это кортеж из трёх целых чисел, а `num#num#num` — его тип; `(25, 'a', true)` — это кортеж из трёх элементов *разных* типов, а его тип, как несложно догадаться, обозначается выражением `num#char#truval`. **Функции в Хоупе принимают на вход и возвращают произвольные кортежи**; в определённом смысле можно сказать, что у функций в этом языке всегда ровно один параметр, но этот параметр — кортеж, так что передать можно сколько угодно значений; то же касается и возвращаемого значения.

Ещё один «сложный» тип — список; в отличие от кортежа список может быть произвольной длины, т. е. длина списка никак на его тип не влияет; но элементы списка обязаны иметь один и тот же тип. К спискам мы ещё вернёмся.

Более сложный для понимания способ создания новых типов несколько напоминает то, как мы в Прологе строили *термы* с помощью *функторов*, только здесь вместо слова «функтор» используется термин «***конструктор***». В роли конструктора может выступать любой идентификатор. Конструктор применяется к произвольному кортежу, например, можно написать `point(num#num)`; если кортеж состоит из одного элемента или вообще пустой, скобки можно опустить. Конструктор, «применённый» к пустому кортежу — то есть попросту записанный сам по себе — даёт хорошо нам знакомое «значение, которое отличается от других», то есть значение, имеющее имя и представляющее собой «вещь в себе» — ничего, кроме имени и того факта, что оно не равно ничему, кроме самого себя, про такое значение сказать нельзя. С помощью таких значений и символа `++`, означающего *объединение множеств*, можно построить ***перечислимый тип*** — что-то вроде `red ++ green ++ blue`.

Сразу же оговоримся, что **символ `++` и новые имена-конструкторы могут встречаться в программе только в директивах описания новых типов, причём только на их верхнем уровне**. Этим они резко отличаются от символа `#`, задающего типы-кортежи — он может встречаться где угодно, где по смыслу предполагается тип. Директива, описывающая новый тип, начинается с ключевого слова `data`, после которого записывается идентификатор — имя для нового типа, затем ставится знак `==` и пишется выражение, задающее собственно вводимый тип. Как и любое предложение программы на Хоупе, директива `data` заканчивается точкой с запятой. Например, перечислимый тип для цветов радуги можно создать так:

```
data Rainbow == red ++ orange ++ yellow ++ green ++
                blue ++ indigo ++ violet;
```

Тип для точек на плоскости — если мы хотим, чтобы он отличался от простого кортежа из двух чисел — можно ввести так:

```
data PlanePoint == Point(num # num);
```

Обратите внимание, что мы ввели здесь десять новых идентификаторов: `Rainbow` и `PlanePoint` в роли *имён типов*, `Point` и имена цветов от `red` до `violet` — в качестве *конструкторов*.

Приведём более практичный пример использования типов с конструкторами. Довольно часто бывает так, что результатом некоторого вычисления должно вроде бы стать число, но может произойти какая-то ошибка или иная особая ситуация, так что числа не получится и нужно будет вернуть из функции специальное значение, показывающее, что вычисление завершилось неудачно. В Хоупе для этого можно воспользоваться, например, таким типом:

```
data MaybeNum == Good(num) ++ Bad;
```

Проиллюстрируем применение этого типа на примере деления, которое, как известно, иногда (конкретно — при нулевом делителе) бывает неуспешным:

```
dec SafeDiv : num # num -> MaybeNum;
--- SafeDiv(_, 0) <= Bad;
--- SafeDiv(x, y) <= Good (x / y);
```

Интересно, что одно и то же имя конструктора можно использовать для разных случаев (например, в разных типах), если будет различаться количество элементов в кортеже, к которому этот конструктор применяется; напомним, в Прологе функторы тоже могли различаться арностью при одинаковых именах. Так, мы могли бы описать тип для точки пространства:

```
data SpacePoint == Point(num # num # num);
```

и конфликта бы не возникло: интерпретатор считал бы выражение `Point(2, 3)` значением типа `PlanePoint`, а выражение `Point(2, 3, 4)` — имеющим тип `SpacePoint`. В то же время использовать данное имя конструктора с данной арностью можно только в одном типе: например, мы не могли бы воспользоваться идентификаторами `red`, `yellow` и `green` для обозначения сигналов светофора, поскольку уже задействовали их как имена цветов радуги (но могли бы при этом ввести какой-нибудь тип с использованием, к примеру, `red(num # num)`).

Отметим ещё один довольно неочевидный момент. Директива `data` предназначена для создания новых типов с использованием конструкторов, так что все идентификаторы, встреченные на верхнем уровне, рассматривает именно как вводимые конструкторы; это обстоятельство не позволяет использовать в теле директивы `data` (на верхнем уровне) символ `#` — точнее говоря, *один* раз это сделать можно, но всё равно получится совсем не то, чего мы ожидали. Следовательно, с помощью этой директивы нельзя описать тип, представляющий собой простой кортеж других типов, не вводящий новых конструкторов. Если очень надо это сделать, можно воспользоваться другой директивой — `type`; эта директива не создаёт новых типов, а только даёт имя некоторому уже существующему типу. Например, можно поступить так:

```
type Vec3num == num # num # num;
```

К происходящему можно подойти с другой стороны: считать, что всё бесконечное многообразие типов-кортежей *уже существует*, тогда как типы, использующие конструкторы, начинают существовать лишь в тот момент, когда мы создаём их с помощью директивы `data`.

Вернёмся к директиве `data`, сделаем замечание, что она допускает рекурсивное использование вводимого имени типа, и попытаемся с использованием этого свойства создать список чисел. Это можно сделать, например, так:

```
data NumList == NumNil ++ NumCons(num # NumList);
```

Значениями нового типа будут, в частности, `NumCons(1, NumNil)`, `NumCons(1, NumCons(2, NumCons(3, NumNil)))` и, конечно, просто `NumNil`. Если нам теперь придёт в голову создать список элементов какого-нибудь другого типа, хоть тех же `char`'ов, придётся использовать другие имена конструкторов:

```
data CharList == CharNil ++ CharCons(char # CharList);
```

Конечно, это не слишком удобно. К счастью, Хоуп позволяет вводить ***параметрические типы***, немного похожие на шаблоны классов в Си++. Например, для всё того же списка можно сделать так:

```
data List(any) == Nil ++ Cons(any # List(any));
```

Само слово `List` — это ещё не тип, это обозначение своеобразного *семейства* типов, таких как `List(char)`, `List(num)`, `List(List(char))` и т. п. В частности, `Cons('a', Nil)` будет значением типа `List(char)`, а `Cons(1, Cons(2, Nil))` — значением типа `List(num)`. Помимо того, что такие типы не нужно описывать каждый в отдельности, у них есть ещё одно несомненное достоинство: конструкторы (в данном случае `Cons` и `Nil`) используются одни на всё семейство, не надо каждый раз придумывать новые идентификаторы.

Между прочим, когда в роли аргумента (конструктора, параметрического типа или, если забежать вперёд, ещё и функции) выступает одно простое значение, можно не заключать его в круглые скобки; **кортеж из одного элемента — это то же, что и сам этот элемент**. Можно сделать более общее утверждение: **запятая, разделяющая элементы кортежа — это тоже конструктор**, она-то и объединяет отдельные элементы в кортеж, а скобки при записи кортежа нужны только потому, что у запятой низкий приоритет.

Аналогично обстоят дела и с символом `#` при записи типов. В частности, определение нашего параметрического типа мы могли бы записать и так:

```
data List any == Nil ++ Cons(any # List any);
```

Ещё один довольно нетривиальный вопрос — а какого типа значением будет выражение, состоящее из одного `Nil`; интерпретатор на этот вопрос ответит, что `Nil` имеет тип `List alpha`, а почему он ответит именно так — станет ясно из дальнейшего изложения. Пока просто скажем, что тип этого выражения *недоопределён*, ведь это с равным успехом мог бы быть пустой список чисел, символов, лысых чертей, волосатых мамонтов и вообще чего угодно; но наш интерпретатор такая «недоопределённость» не смущает.

Наши упражнения по построению списка имели целью показать в действии систему типов Хоупа, в том числе возможность задания параметрических типов; но в действительности эту конкретную задачу — введение списков произвольных элементов — решать особого смысла нет, поскольку, естественно, **в Хоупе есть встроенные списки**, или, если говорить совсем точно, имеется встроенный параметрический тип `list`, использующий в качестве обозначения пустого списка идентификатор `nil` (он же может быть обозначен пустыми квадратными скобками), а в роли конструктора списка из головы и хвоста — инфиксный символ «`::`». Конструктор «`::`» *правоассоциативен*, то есть разбирается справа налево, так что, например, `1 :: 2 :: 3 :: nil` — это то же самое, что `(1 :: (2 :: (3 :: nil)))` (а не наоборот, как можно было ожидать). Мы уже знаем по опыту Лиспа и Пролога, что эта конструкция — и есть список; в Хоупе для списков предусмотрена сокращённая запись в квадратных скобках, так что то же самое можно записать и так: `[1, 2, 3]` — почти как в Прологе, но не совсем. Как мы помним, любой список можно записать в виде «голова-хвост», но если в Прологе список `[1, 2, 3]` можно представить в виде `[1 | [2, 3]]`, то в Хоупе то же самое будет выглядеть так: `1 :: [2, 3]`.

Аналогии с Прологом на этом не заканчиваются: **строка в Хоупе — это список символов** (спасибо, хоть не их кодов), то есть, например, `"Hope"` — это то же самое, что `['H', 'o', 'p', 'e']`.

11.5.6. Функции в Хоупе

Программа на Хоупе, как и на любом функциональном языке, состоит из функций; функцию нужно сначала ***объявить*** с помощью директивы `dec`, а затем ***описать*** одним или более предложениями, начинающимися с тройного тире «`---`».

В действительности директива `dec` предназначена, чтобы объявить произвольный идентификатор *с некоторым значением*, и задаёт *тип* значения, тогда как *директива* `---` задаёт само значение. Например, мы можем с их помощью вводить константы:

```
dec TheUltimateAnswer : num;
--- TheUltimateAnswer <= 42;

dec TheUltimateQuestion : list(char);
--- TheUltimateQuestion <=
    "What do you get if you multiply six by nine";
```

Обратите внимание на двоеточие в директиве `dec` и знак `<=` в предложениях, начинающихся с `---`; эти знаки являются частью синтаксиса рассматриваемых директив.

Как можно догадаться, функция тоже имеет *тип*, зависящий от того, значение какого типа (в общем случае — кортеж) функция принимает и значение (кортеж) какого типа она возвращает. Между типами аргумента и значения ставится знак `->`. Например, функция, возводящая свой числовой аргумент в куб, то есть *принимающая число и возвращающая число*, будет объектом типа `num -> num`. Полностью её описание будет таким:

```
dec Cube : num -> num;
--- Cube(x) <= x * x * x;
```

Функция, принимающая на вход два числа и возвращающая их сумму и произведение в виде кортежа из двух чисел, будет выглядеть так:

```
dec SumProd : num # num -> num # num;
--- SumProd(x, y) <= (x+y, x*y);
```

Результатом вычисления выражения `SumProd(5,6)` станет кортеж `(11, 30)`, имеющий тип `num # num`.

Предложений, описывающих функцию, то есть начинающихся с `---`, может быть больше одного. В этом случае интерпретатор при вычислении значения выбирает подходящее предложение путём ***сопоставления с образцом***[35]. Пусть, к примеру, нам нужно из списка чисел выбросить заданное количество первых элементов, а если это количество превышает длину списка — выдать пустой список. Функцию, которая это делает, можно описать так:

```
dec DropElems : num # list(num) -> list(num);
--- DropElems(x, []) <= [];
--- DropElems(0, ls) <= ls;
--- DropElems(k, head :: tail) <= DropElems(k-1, tail);
```

В первом предложении из двух аргументов используется только один, а в последнем никак не используется первый элемент списка, который мы назвали `head`; этот факт можно подчеркнуть, задействовав знакомую нам по Прологу ***анонимную переменную***, которая в Хоупе обозначается точно так же — одним знаком подчёркивания «`_`». С учётом этого функцию можно переписать так:

```
dec DropElems : num # list(num) -> list(num);
--- DropElems(_, []) <= [];
--- DropElems(0, ls) <= ls;
--- DropElems(k, _ :: tail) <= DropElems(k-1, tail);
```

Когда список пуст, срабатывает первое предложение, возвращающее пустой список независимо от того, сколько элементов из него предлагается выкинуть; если первым аргументом передан ноль, выкидывать ничего не надо — второе предложение возвращает исходный список; наконец, если ни один из тривиальных случаев не подошёл, работает третье предложение — путём выкидывания первого элемента оно в полном соответствии с правилами рекурсивного программирования сводит задачу к ней же самой, но в «чуть-чуть более простом» случае. Интересно, что **порядок предложений в определении функции неважен** — интерпретатор всегда использует «наиболее специфичный» вариант образца (головы предложения) из всех подходящих; в данном случае предложения, у которых только один из аргументов функции представляет собой переменную, а для второго задано конкретное значение, очевидным образом «более специфичны», нежели последнее предложение («более общее»), где оба аргумента заданы переменными.

С этим «правилом наибольшей специфичности» (англ. *most specific rule*) в Хоупе есть определённые проблемы. Даже при взгляде на наш простой пример может возникнуть совершенно законный вопрос, а какое из предложений будет работать для вызова `DropElems(0, [])` — первое или второе. В нашем примере результат от этого не зависит, поскольку оба предложения при таких значениях аргументов вернут пустой список, но остаётся неясным, что случилось бы, если бы они отличались — например, если бы мы захотели возвращать что-то специфическое, когда вызывающий потребовал удалить больше элементов, чем есть в списке.

Это далеко не самый сложный вариант, ведь в образцах могут фигурировать конструкторы, а переменные могут быть «закопаны» сколь угодно глубоко; в некоторых особо тяжёлых ситуациях сопоставление может вообще никогда не завершиться. Для всех случаев, когда возможных значений результата больше одного, Росс Патерсон — автор исходной версии той реализации, которую мы используем — заявил, что неким «детерминированным» способом будет выбрано одно из возможных значений, но сам этот способ не указывается[36].

11.5.7. Операции `if` и `let`

До сих пор в телах функций мы использовали арифметические выражения, которые строятся вполне привычным образом, а также выражения с конструкторами, порождающие более сложные структуры данных из более простых, и вызовы функций. Кроме этого, Хоуп поддерживает ***условные выражения*** (`if-then-else`), а также `let`-выражения и `where`-выражения, позволяющие вводить локальные переменные и в конечном счёте строить более сложные тела функций. Мы оставим за рамками книги операцию `where`, которая до какой-то степени избыточна, и рассмотрим только `if` и `let`.

Начнём с условного выражния как более простого. Слова `if`, `then` и `else` нам хорошо знакомы по императивным языкам, так что подсознательно мы можем ожидать здесь чего-то вроде условного оператора — но окажемся неправы, поскольку в Хоупе вообще нет операторов в традиционном императивном смысле; тем не менее, хоуповское выражение, начинающееся со слова `if` — это инструмент, который нам хорошо знаком со времён изучения Си, просто это не оператор, а ***условная операция***. Читатель может припомнить, что та же операция присутствует и в классическом Лиспе, и в Scheme — в виде спецформы `if`. Как и в Си, условная операция в Хоупе тернарна, т. е. предполагает три аргумента, ни один из которых не может быть опущен; в этом плане Лисп и Scheme оказываются даже более «либеральны», ведь в них можно опустить ветку `else` — но условная операция в неполной форме имеет какой-то смысл только при наличии побочных эффектов, а их, как мы знаем, в Хоупе нет. Условное выражение в Хоупе выглядит так:

$$\texttt{if } A \texttt{ then } B \texttt{ else } C$$

где A — логическое выражение, B и C — произвольные выражения, имеющие *одинаковый тип*. При вычислении `if` интерпретатор сначала вычисляет A, и если получилась истина, вычисляет B, в противном случае вычисляет C; результат вычисления B или C становится результатом всего `if`.

Слово `let`, как и слово `if`, обозначает *операцию*, но работает эта операция сложнее. В общем виде выражение, основанное на операции `let`, выглядит так:

$$\texttt{let } V \texttt{ == } E \texttt{ in } B$$

Здесь V — выражение, содержащее переменные (возможно, просто одна переменная), E — выражение, задающее локальные значения для переменных из V, а B — произвольное выражение, в котором переменные из V могут встречаться и, если будут встречены, вместо них интерпретатор подставит полученные ими (при сопоставлении с E) значения. Например,

```
let x == 15 in
```

вычислит выражение, стоящее после `in`, подставив `15` вместо `x`. В таком варианте это вряд ли осмысленно, но вот следующий пример, можно надеяться, позволит вспомнить предназначение локальных переменных:

```
let x == f(a,b,c) in (p(x), q(x), r(x))
```

(если бы не `let`, `f(a,b,c)` пришлось бы вычислять трижды или вводить вспомогательную функцию). А по-настоящему мощь `let` раскрывается, когда в роли V выступает что-то более сложное, чем одна переменная — например, кортеж переменных. Это позволяет задать локальные значения сразу нескольким переменным, например:

[35] Возможно, это не самый удачный термин для описания происходящего, но в английском оригинале действительно используется термин *pattern matching*.

[36] В оригинале *one of the possible values is chosen, in a deterministic (but otherwise unspecified) manner*.

```
let (x, y, z) == (f(t), g(t), h(t)) in
```

Здесь вроде бы ничего особенного не происходит, просто вычисляются три значения (точнее, *кортеж* из трёх значений), и всем этим значениям даются имена, которые можно использовать в выражении после `in`. Вспомним теперь, что *функция может возвращать кортеж*, и мы догадаемся, что `let` будет очень удобен для анализа сложных значений, возвращаемых из функций. Например, если некая функция `v` возвращает в виде кортежа трёхмерный вектор, с помощью `let` можно получить доступ к каждой из координат вектора в отдельности и, скажем, вычислить длину (модуль) вектора:

```
let (x, y, z) == v(t) in sqrt(x*x + y*y + z*z)
```

В роли выражения V можно использовать не только кортежи, но и другие сложные данные, в том числе списки. Например, если бы функция `v` возвращала не кортеж, а *список* из трёх координат, можно было бы сделать так:

```
let [x, y, z] == v(t) in sqrt(x*x + y*y + z*z)
```

Разбить произвольный непустой список на голову и хвост можно так:

```
let head :: tail == SomeList(a, b) in
```

Теперь, кстати, можно догадаться, почему в стандартной библиотеке Хоупа нет функций, возвращающих голову и хвост списка (аналогов лисповских `car` и `cdr`): когда в нашем распоряжении столь мощный `let`, такие функции просто не нужны. Впрочем, написать их очень легко, причём `let` для этого не нужен; оставим это читателю в качестве упражнения.

В роли V и E могут выступать структуры данных, построенные с помощью любых конструкторов — лишь бы эти конструкторы были раньше введены с помощью `data`. Есть тут, впрочем, и ограничения. Во-первых, каждая переменная в V может встречаться не больше одного раза; во-вторых, в E переменные из V недоступны, так что хоуповский `let` существенно проигрывает по своей изобразительной мощности, скажем, операции унификации в Прологе (см. §11.4.4).

В выражении E нельзя использовать переменные, входящие в V — точнее говоря, интерпретатор попытается найти переменные с такими именами в объемлющем контексте, и, не найдя их там, выдаст ошибку. В этом рассматриваемый нами интерпретатор отличается от диалекта Хоупа, описанного в монографии [24]; операция `let`, описанная там, эквивалентна операции `letrec`, которой мы позже посвятим отдельный параграф (§11.5.10).

Чтобы продемонстрировать `if` и `let` в деле, рассмотрим чуть более сложный пример — функцию, разбивающую заданную строку на отдельные слова. Она будет принимать в качестве аргумента строку (то есть список символов), а возвращать список строк, то есть её объявление будет выглядеть так:

```
dec BreakString : list(char) -> list(list(char));
```

Для решения задачи мы, естественно, применим рекурсию. Функция будет возвращать список слов (т. е. *список списков* символов), первое из которых (возможно, *пустое* — пустой список) считается «недостроенным», то есть к нему спереди, возможно, ещё нужно добавить какие-то символы; все слова в возвращаемом списке, кроме самого первого, считаются уже готовыми. Будучи вызванной от некоторой строки, функция для начала расщепит эту строку на первый символ и остаток и применит сама себя к этому остатку. Дальнейшее зависит от того, что собой представляет отщепленный первый («текущий») символ. Если это не пробел, то тут всё просто: от списка, полученного из рекурсивного вызова, отделяем первый элемент (недостроенное слово), добавляем к нему спереди текущий символ, формируем список из нового слова и остальных («готовых») слов и в таком виде возвращаем. Если же текущий символ — пробел, то тут есть два варианта. Когда первый элемент списка, возвращённого рекурсивным вызовом, т. е. недостроенное слово — не пустое, его пора переводить в статус готового: в самом деле, раз перед ним в исходной строке стоит пробел, то спереди к нему добавлять больше нечего. Наша функция отработает эту ситуацию, добавив к списку в начало пустой элемент — этот элемент станет новым (пока что пустым) недостроенным словом. Если же недостроенное слово пустое — это означает, что наш текущий пробел незначащий, то есть или после него стоял пробел, или сразу после него строка кончилась. В этом случае мы вернём ровно то, что нам вернул рекурсивный вызов, тем самым мы просто проигнорируем незначащий пробел.

Базисом рекурсии, естественно, выберем пустую исходную строку; в этом случае нужно вернуть список из одного лишь пустого недостроенного слова, то есть список `[[]]`.

Остаётся ещё одна небольшая проблема. Если строка исходно была пустая, состояла только из незначащих пробелов, либо незначащие пробелы присутствовали в её начале (перед первым словом), в результирующем списке так и останется пустое недостроенное слово, которому в нашем конечном результате делать совершенно нечего. Поэтому весь вышеописанный алгоритм мы реализуем во вспомогательной функции, которую назовём `BreakStringDo`, имеющей такой же профиль, как `BreakString`, тогда как сама функция `BreakString` будет только вызывать `BreakStringDo`, проверять, не является ли первый элемент результата пустым списком, и если да — возвращать результирующий список уже без него.

В решении мы воспользуемся `let` для расщепления списка, возвращённого вызванной функцией, на первый элемент и остаток, а `if` задействуем, чтобы разделить случаи, когда первый элемент возвращённого списка (то самое «недостроенное» слово) пустой и непустой. Полностью решение будет таким:

```
! strwords.hop

dec BreakStringDo : list(char) -> list(list(char));
--- BreakStringDo([]) <= [[]];
--- BreakStringDo(' ' :: Rest) <=
        let Word :: List == BreakStringDo(Rest) in
                if Word = []
                        then [] :: List
                        else [] :: Word :: List;
--- BreakStringDo(c :: Rest) <=
        let Word :: List == BreakStringDo(Rest) in
                (c :: Word) :: List;

dec BreakString : list(char) -> list(list(char));
--- BreakString(str) <=
        let Word :: Rest == BreakStringDo(str) in
                if Word = []
                        then Rest
                        else Word :: Rest;
```

Если задаться целью сделать то же самое без `if` и `let`, то это окажется не очень сложно, просто потребуются ещё две вспомогательные функции:

```
dec DropFirstEmpty : list(list(char)) -> list(list(char));
--- DropFirstEmpty([] :: T) <= T;
--- DropFirstEmpty(H :: T) <= H :: T;

dec CharToFirstWord : char # list(list(char)) -> list(list(char));
--- CharToFirstWord(c, W :: L) <= (c :: W) :: L;

dec BreakStringDo : list(char) -> list(list(char));
--- BreakStringDo([]) <= [[]];
--- BreakStringDo(' ' :: Rest) <=
        [] :: DropFirstEmpty(BreakStringDo(Rest));
--- BreakStringDo(c :: Rest) <=
        CharToFirstWord(c, BreakStringDo(Rest));

dec BreakString : list(char) -> list(list(char));
--- BreakString(str) <=
        DropFirstEmpty(BreakStringDo(str));
```

11.5.8. Полиморфные функции

Вернёмся к функции `DropElems` (см. стр. 167). Эта функция убирает из списка *чисел* заданное количество первых элементов и со своей задачей вроде бы справляется. Но что делать, если нам потребуется убрать N первых элементов из списка *символов*, или из списка списков символов, или из списка элементов ещё какого-нибудь сложного типа? В теле `DropElems` нигде не используется тот факт, что список состоит именно из чисел, так что функция, делающая то же самое со списком любых других элементов, будет выглядеть точно так же — отличаться будет лишь директива `dec`.

Подобную ситуацию мы рассматривали в §10.7.1, чтобы объяснить, зачем в Си++ нужны шаблоны функций. В Лиспе, Scheme и Прологе такой проблемы не возникает, поскольку переменные там не привязаны к типу, а списки и прочие структуры данных гетерогенны, то есть могут состоять из значений разных типов. Но вот мы снова рассматриваем строго типизированный язык (а Хоуп именно такой) и сталкиваемся с перспективой написать две функции, которые будут совершенно одинаковы, если не считать некоторого *типа*.

Хоуп для таких ситуаций предусматривает так называемые ***полиморфные функции***. В объявлении (директиве `dec`) для такой функции вместо одного или нескольких типов указываются ***ти́повые переменные*** (*type variables*) — специальные идентификаторы, «значениями» которых выступают произвольные типы. Хоуп вводит три такие переменные — `alpha`, `beta` и `gamma`; если их не хватит, можно дополнительные ти́повые переменные ввести с помощью директивы `typevar`, например:

```
typevar delta, epsilon, anytype;
```

С использованием такой переменной мы можем переписать функцию `DropElems` так, что она будет работать со списками элементов произвольного типа:

```
dec DropElems : num # list(alpha) -> list(alpha);
--- DropElems(_, []) <= [];
--- DropElems(0, ls) <= ls;
--- DropElems(k, _ :: tail) <= DropElems(k-1, tail);
```

Стоит отметить, что изменили мы только профиль функции (текст директивы `dec`), тело менять не пришлось. В отличие от Си++, где тип, задаваемый параметром шаблона, может встретиться в теле функции-шаблона, например, при описании переменных, в Хоупе такого произойти не может — здесь переменные тоже типизированы, но их тип *всегда выводится автоматически*.

Читатель мог обратить внимание, что с «переменной, которая обозначает какой-нибудь тип», мы уже сталкивались при обсуждении *параметрических типов* на стр. 166, причём использованный там в этой роли идентификатор `any` мы никак специально не описывали. Дело в том, что в директиве `data` из контекста понятно, что́ имеется в виду, тогда как в директиве `dec` на месте ти́повой переменной может стоять имя какого-то конкретного типа. Именно поэтому, чтобы убедить интерпретатор, что мы действительно имеем в виду полиморфную функцию и нигде не опечатались, идентификаторы ти́повых переменных мы должны явным образом задекларировать.

Может возникнуть отдельный вопрос, следует ли «полиморфную функцию» считать, собственно говоря, *функцией*, или лучше, как в Си++, предполагать, что это только некая «заготовка» для функции, а в полноценную функцию она превращается лишь тогда, когда все неизвестные типы оказываются конкретизированы. В действительности для Хоупа этот вопрос лишён смысла. В Си++ мы вынуждены были делать различие между шаблоном и полученной из него функцией, поскольку Си++ — язык компилируемый, при этом перевести текст функции в машинный код, не зная типов используемых переменных и выражений, невозможно. Обычная функция Си и Си++ может быть, в частности, вызвана из другой единицы трансляции, и для этого не нужно видеть её тела, достаточно знать, какой у неё профиль — с остальным справится редактор связей; в противоположность этому шаблонная функция не может быть задействована, если нет доступа к её телу. В Хоупе ситуация совершенно иная, поскольку он интерпретируется, а не компилируется, и никакого редактора связей здесь нет, поэтому нет и никаких *видимых* различий между полиморфной функцией и её гипотетическим экземпляром, полученным при конкретизации ти́повых переменных.

Теоретически можно было бы, проанализировав исходные тексты интерпретатора, понять, создаёт ли он какое-то особое внутреннее представление для отдельных экземпляров полиморфной функции, и на этом основании всё-таки дать ответ, является ли полиморфная функция функцией сама по себе; но тогда этот ответ будет зависеть от внутренних особенностей реализации интерпретатора, которые, вообще говоря, пользователю (программисту, пишущему на Хоупе) не видны — их можно поменять, и для нас, программистов, при этом ничего не изменится.

11.5.9. Безымянные функции и функционалы

Как мы уже имели возможность убедиться, функции высоких порядков — то есть такие, которые получают через свои аргументы другие функции — это своего рода визитная карточка функционального программирования. Функционалы есть, разумеется, и в Хоупе, а как только появляются функционалы, для их полноценного использования тут же требуются «лямбды» — безымянные функции, которые в своём окончательном виде (в виде ***замыкания***, фиксирующего значения «свободных» переменных) создаются во время исполнения программы. Замыкания мы подробно обсуждали при изучении Лиспа (см. стр. 122), отметим только, что в Хоупе всё несколько проще, ведь *переменная не может изменить значение, полученное при её создании* (точнее, при входе в определённый контекст — предложение функции или `let`-выражение).

Начнём с функционалов и для примера попробуем описать классическую функцию `map`, которая получает на вход некую функцию одного аргумента и список, применяет полученную функцию к каждому элементу списка, а из полученных результатов формирует новый список, который и возвращает.

Подобная функция в Лиспе называется `mapcar` (см. стр. 119), а в Scheme — просто `map`; к сожалению, аналогия тут будет неполной, поскольку в Хоупе нет вариадических функций, и вдобавок строгая типизация не позволяет одному и тому же функционалу передавать в разных случаях функции от различного количества аргументов. Поэтому нам придётся в примере ограничиться одним списком и функцией одного аргумента — в отличие от лиспоподобных языков, где аналогичные функции принимают на вход функцию произвольного количества аргументов и соответствующее количество списков.

Итак, у нас имеется список из элементов *какого-то* типа и некая функция (назовём её `f`), принимающая единственный аргумент того же типа. Возвращать она может значение какого-то другого типа, хотя, конечно, никто не мешает этим двум типам совпадать. Наш функционал будет формировать список полученных результатов, то есть как раз элементов этого «другого» типа. Как можно догадаться, реализация `map` никак не зависит от того, какие конкретно это будут два типа, так что мы сделаем наш `map` полиморфным. Вспомнив, что в Хоупе есть три предопределённые ти́повые переменные — `alpha`, `beta` и `gamma`, мы обозначим через `alpha` тип аргумента функции `f`, через `beta` — тип значения, которое она возвращает. Иначе говоря, предположим, что `f` в терминах Хоупа *имеет тип* `alpha -> beta`.

Функция `map` должна принять на вход функцию `f`, то есть значение типа `(alpha -> beta)`, и список элементов типа `alpha` (то есть значение типа `list(alpha)`), а возвращать она будет список элементов типа `beta` (значение типа `list(beta)`). Следовательно, профиль её получается таким:

```
dec map : (alpha -> beta) # list(alpha) -> list(beta);
```

Определение функции будет довольно простым. Если исходный список пуст, вне зависимости от функции (которую попросту не к чему применять) нужно вернуть пустой список, в противном случае следует применить переданную функцию к первому элементу списка, с помощью рекурсивного вызова применить ту же функцию к элементам остатка списка, из полученного значения и списка создать новый список и вернуть его. Выглядит это так:

```
--- map(_, []) <= [];
--- map(f, x::ls) <= f(x) :: map(f, ls);
```

Если, например, у нас будет функция, умножающая свой аргумент на десять:

```
dec Times10 : num -> num;
--- Times10(x) <= 10 * x;
```

и мы применим `map` к этой функции и списку чисел `[1, 2, 3, 4]` вот таким образом:

```
map(Times10, [1, 2, 3, 4])
```

— то результатом станет список `[10, 20, 30, 40]`. Но мы уже знаем на примере Лиспа и Scheme, что в таких случаях намного удобнее использовать безымянную функцию, и в Хоупе такая возможность, разумеется, есть. Вместо нашей `Times10` мы могли бы использовать вот такое:

```
map(lambda x => 10 * x, [1, 2, 3, 4])
```

Вместо слова `lambda` можно для краткости написать символ обратного слэша «\»:

```
map(\ x => 10 * x, [1, 2, 3, 4])
```

Конструкция `lambda x => 10 * x` (или, что то же самое, `\ x => 10 * x`) представляет собой не что иное, как *выражение*, результатом *вычисления* которого является *функция как объект данных*. Такой объект можно не только передавать в функции, но и возвращать из них; например, функция

```
dec MakeAdder : num -> (num -> num);
--- MakeAdder x <= lambda y => x + y;
```

позволяет построить «прибавлялку» любого заданного числа[37]; так, результатом выражения `MakeAdder 17` станет *функция* одного аргумента, прибавляющая к своему аргументу число 17. При вычислении выражения

```
let plus21 == MakeAdder 21 in plus21 19
```

сначала будет создан объект функции, прибавляющей к своему аргументу `21`, этот объект будет (как значение) связан с переменной `plus21`, после чего будет вычислено выражение `plus21 19`, здесь «прибавлялка 21» будет применена к числу `19`, в результате получится `40`. Это, впрочем, можно записать и короче: выражение `(MakeAdder 21) 19` сделает совершенно то же самое, только без использования локальной переменной.

Отметим, что «операция» `->` в директиве `dec` работает справа налево (правоассоциативна), так что скобки мы при объявлении `MakeAdder` применили исключительно для наглядности. Если ещё заменить слово `lambda` на слэш, получится так:

```
dec MakeAdder : num -> num -> num;
--- MakeAdder x <= \ y => x + y;
```

Стало ли описание `MakeAdder` от этого лучше или хуже — решайте для себя сами. Мы же пока заметим, что, имея произвольную бинарную инфиксную операцию, в том числе `+`, мы в Хоупе можем из неё получить функцию одного аргумента, зафиксировав второй аргумент, и записать это можно в упрощённой форме: конкретно для операции `+` выражение `(+t)` будет синонимом (сокращённой записью) `lambda x => x + t`, а выражение `(t+)` — синонимом `lambda x => t + x`. Результатом выражения `map((-2), [10, 15, 20])` станет список `[8, 13, 18]`, результатом `map((20-), [10, 15, 20])` — список `[10, 5, 0]`. Числовой арифметикой эта возможность не ограничена: например, в роли бинарной операции можно использовать конструктор списков «`::`», и, скажем, результатом `map((::[100]), [10, 15, 20])` будет `[[10, 100], [15, 100], [20, 100]]`. Естественно, с учётом этого наша функция `MakeAdder` оказывается просто не нужна, вместо `MakeAdder` N всегда можно написать `(+`N`)` — но иллюстративной ценности примера это не отменяет.

Раз об этом зашла речь, отметим, что запись `(+)` в Хоупе считается синонимом записи `lambda(x,y)=>x+y`, и это тоже можно применить к любой инфиксной бинарной операции.

Лямбда-функция может состоять не только из одного предложения, но и из нескольких; предложения при этом отделяются символом `|`. Например, если нам потребуется анонимная версия функции `DropFirstEmpty` (см. стр. 168), можно будет написать что-то вроде

```
lambda ([] :: T) => T | (H :: T) => H :: T;
```

Отметим, что **для лямбда-функций в Хоупе невозможно явно указать тип — интерпретатор всегда его выводит сам**.

Сделаем ещё одно интересное замечание. В рассматриваемом нами диалекте выражение вида `let` P `==` E `in` B в точности эквивалентно выражению `(lambda` P `=>` B `)` E.

11.5.10. Операция `letrec`

Имеющаяся в Хоупе операция `letrec` позволяет делать довольно неожиданные вещи и в целом оказывается важна для формирования впечатления о функциональном программировании. Она выглядит (и работает) подобно уже знакомому нам `let` (см. §11.5.7):

`letrec` V `==` E `in` B

Как и для операции `let`, здесь V — выражение с переменными (одна переменная, кортеж переменных, список переменных), E — выражение, задающее то, каким должно быть локальное значение V, а B — тело, в котором введённые локальные значения должны иметь эффект. Важнейшее отличие состоит в том, что *вводимые ею локальные переменные можно использовать не только в её теле B, но и в выражении E*.

Очевидным применением для `letrec` можно считать введение локальных рекурсивных и взаимнорекурсивных функций. Пусть нам, к примеру, потребовалась такая же функция, как `DropElems` (см. стр. 169), но мы почему-то не хотим давать ей глобально видимое имя с помощью `dec` и `---`. Ввести эту функцию в виде безымянной с помощью `lambda` несколько затруднительно, поскольку она обращается сама к себе и должна для этого как-то идентифицировать сама себя. Операция `letrec` позволяет дать функции *локальное* имя, которое будет использовано в том числе в её теле:

```
letrec drop_elems ==
        lambda    (_, []) => []
                | (0, ls) => ls
                | (k, _ :: tail) => drop_elems(k-1, tail)
in
        ! ...
```

Из рассмотренных нами ранее языков `letrec` присутствует также в Scheme, и там область применения этой формы фактически ограничивается введением локальных функций, обращающихся друг к другу, но эта возможность сама по себе не столь интересна, и мы даже не стали рассматривать `letrec`, когда изучали Scheme — там есть вещи поинтереснее.

В Хоупе и других ленивых языках возможности `letrec` намного шире благодаря, собственно говоря, ленивой семантике вычислений. Приведём один простой пример:

```
letrec lst == 1 :: 2 :: lst in FirstN(lst, 5)
```

На всякий случай напомним, что конструктор «`::`» левоассоциативен, так что выражение справа от `==` эквивалентно выражению `1 :: (2 :: lst)`, то есть это список, первыми двумя элементами которого выступают константы `1` и `2`, а остаток равен значению переменной `lst` — но значением этой переменной операция `letrec` сделала сам список целиком. Результатом вычисления приведённого выражения будет список `[1, 2, 1, 2, 1]` — это список из пяти первых элементов циклического списка `[1, 2, 1, 2, 1, 2, ...]`. С аналогичным способом построения циклических списков мы уже знакомы из рассказа о Прологе (см. стр. 147); там вместо `letrec` использовалась операция унификации, но суть совершенно та же — переменная полагается одновременно равной списку целиком и некоторому его остатку.

В Scheme из-за использования энергичной модели вычислений ничего подобного сделать не получится, а в Лиспе вовсе нет примитива `letrec`; циклический список, как мы знаем, можно сделать и там, но для этого нужны *разрушающие операции* — например, построив список из двух элементов `(1 2)`, мы можем заменить объект пустого списка в его конце на сам этот список. В Лиспе это, как мы знаем, делается так (см. стр. 117):

```
(setq lst '(1 2 3))
(rplacd (cdr (cdr lst)) lst)
```

Аналогичный приём можно применить и в Scheme:

```
(set! lst '(1 2 3))
(set-cdr! (cdr (cdr lst)) lst)
```

Но всё это, как мы понимаем, совершенно не то. В Хоупе мы смогли добиться аналогичного эффекта, несмотря на принципиальное отсутствие в нём разрушающих действий; в Прологе разрушающие действия, вообще говоря, есть, но унификация к ним не относится, а для получения циклического списка ничего, кроме унификации, не требуется. Как видим, циклические структуры данных Пролога и Хоупа — явление хотя и похожее, но всё-таки качественно иное, нежели циклические списки в Лиспе и Scheme.

[37] Этот пример мы уже использовали при изучении Лиспа, см. стр. 120.

11.5.11. Редукция списков

Редукция (она же *свёртка*) последовательностей нам уже хорошо знакома — мы неоднократно встречались с этим приёмом, приводя примеры и на чистом Си, и на Си++, и на Лиспе; но, как мы сейчас увидим, ленивая стратегия вычислений привносит в хорошо знакомую нам технику новое качество, которое стоит того, чтобы быть показанным.

Рассматривать в этот раз мы будем только *правую* редукцию. Обсуждая свёртку списков на Лиспе (см. §11.1.11), мы отметили, что реализация левой рекурсии допускает остаточную («хвостовую») оптимизацию, тогда как правая редукция хвостовой быть не может; как ни странно, благодаря ленивым вычислениям мы можем не обращать на это внимания, правая редукция оказывается даже удобнее.

Напомним вкратце, что происходит. У нас имеется список элементов некоторого типа (назовём его `alpha`) и функция `f`, которая принимает на вход первым параметром элемент типа `alpha`, вторым параметром — элемент какого-то ещё типа, в общем случае этот тип может отличаться от `alpha` (пусть это будет тип `beta`); этот же тип имеет возвращаемое функцией `f` значение. Задавшись некоторым начальным значением («затравкой») типа `beta`, мы последовательно применяем функцию `f`: к последнему элементу списка и к затравке, затем к предпоследнему элементу списка и к значению, полученному на предыдущем шаге, и т. д., пока не дойдём до первого элемента списка. Результат применения `f` к первому элементу списка и тому, что она вернула на предыдущем шаге, считается результатом всей редукции.

Чтобы написать на Хоупе функцию `Reduce`, прикинем для начала, какие параметры она должна получать и что будет возвращать. Естественно, функция будет полиморфной, то есть мы воспользуемся ти́повыми переменными, при этом не станем изобретать лишнего — `alpha` и `beta` нас в этой роли вполне устроят. Аргументов для функции `Reduce` нам нужно три: функция `f` (которая имеет тип `alpha # beta -> beta`), редуцируемый список типа `list(alpha)` и затравка типа `beta`. Результатом редукции будет значение типа `beta`. Таким образом, декларация функции `Reduce` получается такая:

```
dec Reduce : (alpha # beta -> beta) # list(alpha) # beta -> beta;
```

Описание (т. е. тело) функции выходит не слишком сложным. В качестве базиса редукции мы воспользуемся тривиальным случаем, когда список пуст; результатом при этом объявляется затравка. Если же список не пуст, мы воспользуемся самой функцией `Reduce`, чтобы для тех же значений функции и затравки свернуть остаток списка, после чего останется только применить `f` в последний раз:

```
--- Reduce(_, [], init) <= init;
--- Reduce(f, h :: t, init) <= f(h, Reduce(f, t, init));
```

Простейшими примерами использования этой функции могут служить, как обычно, сумма и произведение: `Reduce((+),[1,2,3,4],0)` даст[38] значение `10`, `Reduce((*),[1,2,3,4],1)` — значение `24`. Припомнив содержание предыдущих частей книги, читатель наверняка сможет придумать и другие примеры.

Попробуем разобраться, что́ же такого нового дают здесь ленивые вычисления; для этого попытаемся с помощью редукции реализовать функцию `Member`, выясняющую, есть ли в заданном списке заданный элемент. Для начала сделаем так:

```
dec Member : alpha # list(alpha) -> truval;
--- Member(x, ls) <=
        Reduce(\(el, ok) => if ok then true else x = el,
                ls, false);
```

При всей своей очевидности и лаконичности это решение обладает неприятным недостатком; чтобы понять, в чём он заключается, попробуем проследить вычисление выражения `Member(1, [1,2,3])`. Безымянную функцию от параметров `(el, ok)`, используемую в роли первого аргумента `Reduce`, обозначим для краткости «φ», а значение свободной по отношению к φ переменной `x` вынесем в нижний индекс (φ_1 будет означать «φ при `x = 1`»; собственно говоря, значение `x` у нас меняться не будет, но полезно по крайней мере не забывать о его существовании). Выражение `Member(1, [1,2,3])` превратится в

$$\texttt{Reduce(}\varphi_1\texttt{, [1,2,3], false)}$$

Из него в свою очередь получится

$$\varphi_1\texttt{(1, Reduce(}\varphi_1\texttt{, [2,3], false))}$$

Посмотрим, как будет вычисляться φ_1. В роли параметра `el` у нас `1`, в роли `ok` — выражение `Reduce(`φ_1`, [2,3], false)`, т. е. имеем

$$\texttt{if Reduce(}\varphi_1\texttt{, [2,3], false) then true else 1 = 1}$$

Поскольку вычисление `if` невозможно продолжить, не зная, истина в условном выражении или ложь, интерпретатор будет вынужден выполнить рекурсивный вызов `Reduce`, так что у нас последовательно получатся

$$\texttt{if }\varphi_1\texttt{(2, Reduce(}\varphi_1\texttt{, [3], false)) then true else 1 = 1}$$

$$\texttt{if (if Reduce(}\varphi_1\texttt{, [3], false) then true else 1 = 2) then true else 1 = 1}$$

$$\texttt{if (if }\varphi_1\texttt{(3, Reduce(}\varphi_1\texttt{, [], false)) then true else 1 = 2) then true else 1 = 1}$$

$$\texttt{if (if (if Reduce(}\varphi_1\texttt{, [], false) then true else 1 = 3) then true else 1 = 2) then true else 1 = 1}$$

В этом месте для `Reduce` сработает базис и начнётся обратный ход рекурсии:

$$\texttt{if (if (if false then true else 1 = 3) then true else 1 = 2) then true else 1 = 1}$$

$$\texttt{if (if 1 = 3 then true else 1 = 2) then true else 1 = 1}$$

$$\texttt{if 1 = 2 then true else 1 = 1}$$

— и мы получим наконец значение `true`. Как видим, несмотря на то, что искомый элемент находился в самом начале списка, наша реализация `Member` благополучно просмотрела весь список целиком, а происходит это из-за того, что операция `if` вынуждена требовать вычисления своего первого параметра (условия) — как говорят, она *строгая* по этому параметру.

Конечно, такой просмотр всего списка — совсем не лучшая идея; но исправить недостаток нашей реализации оказывается на удивление просто. Перепишем тело `Member` так, чтобы рекурсивный вызов располагался не в условии `if`, а в одной из его ветвей:

```
--- Member(x, ls) <=
        Reduce(\(el, ok) => if x = el then true else ok,
                ls, false);
```

и снова попробуем проследить вычисление `Member(1, [1,2,3])`. Первые два шага будут точно такими же, как для предыдущей реализации:

$$\texttt{Reduce(}\varphi_1\texttt{, [1,2,3], false)}$$

$$\varphi_1\texttt{(1, Reduce(}\varphi_1\texttt{, [2,3], false))}$$

В этот момент проявится отличие новой реализации: получится

$$\texttt{if 1 = 1 then true else Reduce(}\varphi_1\texttt{, [2,3], false)}$$

Вычисление этого выражения завершится в один шаг, рекурсивного вызова `Reduce` не произойдёт. Здесь наглядно видна возросшая благодаря ленивым вычислениям изобразительная мощь редукции: в самом деле, список удалось не просматривать дальше лишь только потому, что выражение `Reduce(`φ_1`, [2,3], false)` интерпретатор не вычислял, пока его значение не требовалось для продолжения работы, и в итоге оно оказалось вовсе не нужно.

Отметим ещё один момент. Чтобы разобраться, чем плох первый вариант реализации и как его следует изменить, нам пришлось проследить работу интерпретатора, то есть фактически использовать императивный, а не функциональный «режим мышления». Программисты, долгое время работающие на «ленивых» языках (чаще всего на Хаскеле), иногда говорят, что с опытом приходит способность видеть всё вычисление целиком, так что подобные пошаговые разборы оказываются не нужны, а изменения, подобные внесённому нами в тело φ, становятся очевидными. По-видимому, это и есть настоящее вхождение в парадигму функционального программирования.

[38] Если вы забыли смысл записи `(+)`, вернитесь к стр. 170.

11.5.12. Бесконечные структуры данных. Примеры

— Сколько отжиманий может сделать Чак Норрис?
— **Все.**

старый анекдот[39]

Как уже говорилось, изюминка ленивых вычислений состоит в их способности работать с «бесконечными» структурами данных; существует даже целый ряд языков программирования, основанных на энергичной семантике обычных вычислений, но поддерживающих при этом ленивые структуры данных. В Хоупе семантика изначально ленивая; воспользуемся этим, чтобы показать, *как* из этого свойства получаются бесконечные списки.

Конструктор списков «::» в Хоупе не вычисляет свои аргументы, так что если они представляли собой невычисленные выражения, то такими они и останутся, пока в ходе дальнейших вычислений их значения не потребуются в явном виде. Говорят, что конструктор списка ***нестрогий*** по своим аргументам. Приведём пример того, как этим можно воспользоваться. Для начала построим список натуральных чисел — вот прямо так, *весь*; точнее говоря, опишем функцию, которая такой список вернёт. Это будет рекурсивная функция с одним параметром — числом, с которого следует начать список; сам возвращаемый список будет состоять из числа, переданного функции параметром, и хвоста списка, который функция вычислит рекурсивно, обратившись сама к себе:

```
dec From : num -> list(num);
--- From(x) <= x :: From(x+1);
```

Теперь выражение `From(1)` как раз и будет равно искомому списку (да, именно: списку *всех* натуральных чисел). Конечно, если попытаться такой список, например, напечатать, ничего хорошего не получится, поскольку вычисление никогда не завершится; но ведь печатать его никто, собственно говоря, не заставляет.

Опишем теперь две вспомогательные функции, одна будет возвращать N-ный элемент списка, вторая — список из N первых элементов заданного списка:

```
dec Nth : list(alpha) # num -> alpha;
--- Nth(x::ls, n) <= if n = 1 then x else Nth(ls, n-1);

dec FirstN : list(alpha) # num -> list(alpha);
--- FirstN(x::y, n) <= if n = 0 then nil else x::FirstN(y, n-1);
```

Теперь, например, выражение `Nth(From(20), 30)` благополучно вычислится и даст 49; выражение `FirstN(From(10), 3)` построит список `[10, 11, 12]`; в обоих случаях «бесконечность» списка, возвращаемого функцией `From`, интерпретатор совершенно не смущает.

Дело тут, разумеется, в том, что список, построенный функцией `From`, никогда и нигде не хранится в явном виде. Физически существующими объектами в памяти становятся только те его элементы, до которых «дотянулись» другие функции, а остаток списка продолжает существовать в виде соответствующего невычисленного обращения к функции `From`. В наших примерах в первом случае будет вычислено 30 первых элементов «бесконечного» списка, во втором — всего три.

Приведём более интересный пример. Хорошо известен алгоритм поиска простых чисел — так называемое решето Эратосфена: берём последовательность целых чисел, начинающуюся с двойки, затем на каждом шаге первое число в последовательности объявляем простым, после чего вычёркиваем из оставшейся последовательности все числа, которые на него делятся. Основной недостаток этого алгоритма — необходимость сразу же ограничиться каким-то фиксированным (конечным) количеством рассматриваемых чисел, например, не рассматривать числа, превосходящие 10 000 (или какое-то ещё число). Если при этом ставится задача найти определённое количество простых чисел, например, выяснить, каковы из них первые сто, этот «потолок» приходится как-то угадывать.

С помощью ленивых вычислений мы можем отказаться от конкретных значений «потолка», рассматривая бесконечный список чисел. Опишем для начала функцию, которая из заданного списка выбрасывает все элементы, делящиеся нацело на заданное число:

```
dec Filter : num # list(num) -> list(num);
--- Filter(n, m::ls) <=
        if (m mod n) = 0
                then Filter(n, ls)
                else m::Filter(n, ls);
```

Теперь мы можем описать само «решето»:

```
dec Sieve : list(num) -> list(num);
--- Sieve(n::ls) <= n::Sieve(Filter(n, ls));
```

Вычислив выражение `FirstN(Sieve(From(2)), 1000)`, мы получим первую тысячу простых чисел, заодно выяснив, что тысячное по счёту простое число — это число 7919. На компьютере автора книги такое вычисление заняло чуть больше двух секунд. К сожалению, реальная вычислительная сложность здесь растёт нелинейно: чтобы вычислить такое же выражение, только уже для 10 000 в роли второго аргумента, потребовалось несколько минут. Десятитысячным простым числом оказалось 104 729; текст этого примера включён в архив под именем `sieve.hop`.

Рассмотрим ещё один пример — вычисление биномиальных коэффициентов C_n^k. Как мы знаем (см. т. 1, §1.3.1), намного эффективнее это делать через треугольник Паскаля, чем через факториалы. Не мудрствуя лукаво, построим *весь* треугольник Паскаля (ну и что, что он бесконечный). Для этого сначала опишем функцию, которая будет строить следующую строку треугольника, получив текущую строку в качестве параметра. Все элементы строки, кроме первого и последнего, можно вычислить как сумму двух элементов исходной строки; последний элемент — точнее, список из него одного — мы объявим базисом рекурсии, а первый элемент просто приделаем к уже готовой строке, для чего нам потребуется вспомогательная функция. Точнее говоря, вспомогательная функция, которую мы назовём `NextLineRest`, будет рекурсивно строить один список по содержимому другого, а основная — `NextLine` — будет к построенному списку спереди добавлять единицу. Всё вместе будет выглядеть так (код этого примера содержится в файле `pascal_t.hop`):

```
dec NextLineRest : list(num) -> list(num);
--- NextLineRest([k]) <= [k];
--- NextLineRest(a::b::ls) <= (a+b) :: NextLineRest(b::ls);

dec NextLine : list(num) -> list(num);
--- NextLine(ls) <= 1::NextLineRest(ls);
```

Если у нас есть очередная строка треугольника Паскаля, то весь бесконечный треугольник, начиная с этой строки, состоит из самой этой строки, а также из остатка треугольника Паскаля, который начинается со следующей его строки. При всей тавтологичности этого утверждения оно прекрасно переводится на язык Хоуп:

```
dec PascalsTriangleBuild : list(num) -> list(list(num));
--- PascalsTriangleBuild(ls) <=
        ls::PascalsTriangleBuild(NextLine(ls));
```

Весь треугольник Паскаля теперь можно построить, начав с его самой первой строки, которая, как мы знаем, состоит из одной единицы, то есть это список `[1]`. Дадим полученной бесконечной структуре данных имя:

```
dec PascalsTriangle : list(list(num));
--- PascalsTriangle <= PascalsTriangleBuild([1]);
```

Теперь биномиальный коэффициент C_n^k представляет собой k-й элемент n-й строки треугольника, если считать, что они занумерованы с нуля. Модифицируем функцию `Nth` из предыдущего примера так, чтобы она считала первый элемент списка имеющим номер ноль:

```
dec Nth : list alpha # num -> alpha;
--- Nth (x :: ls, n) <=
        if n = 0 then x else Nth (ls, n - 1);
```

Собственно говоря, всё, можно вычислять C_n^k. Функцию, которая будет это делать, мы назовём `Cnk`:

```
dec Cnk : num # num -> num;
--- Cnk(n, k) <= Nth(Nth(PascalsTriangle, n), k);
```

[39] В английской версии анекдота используется идиома *all of them*; лаконичный русский перевод, к сожалению, лишён части шарма оригинала.

11.5.13. Ввод-вывод в Хоупе

Скажем сразу же, что с тем интерпретатором, который есть в нашем распоряжении, достичь сколько-нибудь правильного поведения программы в нестандартных случаях (как мы это делали с Лиспом и Прологом) попросту невозможно — этот интерпретатор откровенно экспериментальный и, по-видимому, никогда не предназначался для практического программирования. Поэтому мы даже не станем пытаться «затянуть бантики» и обработать все возможные ошибки.

Свои ограничения здесь налагает и язык Хоуп как таковой, ведь разрушающих функций в нём нет, что называется, по определению, то есть их не просто нет, их в Хоупе *не может быть* в силу его природы. Чтобы в таких условиях организовать ввод-вывод, приходится рассматривать потоки ввода-вывода как нечто *совсем иное*.

К счастью, ленивые вычисления, отобрав у нас всякую возможность работы с модифицирующими действиями и побочными эффектами, при этом кое-что дают нам взамен — те самые «бесконечные» структуры данных, или, лучше сказать, такие структуры данных, с которыми мы можем работать, несмотря на то, что они пока что до конца не вычислены (а в некоторых случаях и не могут быть вычислены ввиду их *бесконечности*). Это даёт нам возможность *рассматривать потоки как «недостроенные» (если угодно, потенциально бесконечные) списки символов*.

Потоки ввода в интерпретаторе Hopeless представлены встроенной функцией `read`, параметром которой служит имя файла, а возвращает она объект типа `list(char)`, то есть список символов — тех самых символов, которые будут прочитаны из потока. Естественно, этот список «недостроен», операции чтения интерпретатор выполняет не раньше, чем программа доберётся до такого места в вычислениях, где требуется *конкретное значение* очередного символа из списка. В этот момент интерпретатор может, что вполне естественно, заблокироваться в ожидании очередной порции информации из потока. Если функции `read` в качестве имени файла передать пустую строку (или, что то же самое, пустой список, то есть сойдёт и `""`, и `[]`), мы получим поток стандартного ввода.

Отметим, что с помощью `read` мы можем работать с несколькими потоками ввода одновременно — достаточно, к примеру, объединить несколько потоков в кортеж, список или другую структуру данных и время от времени «отщеплять» и анализировать очередную порцию символов из того или другого потока.

С потоками вывода всё несколько сложнее. О потоке ввода мы можем думать как о чём-то «уже существующем, просто пока не полностью вычисленном», но поток вывода нам предстоит построить самим, а чтобы не утратить возможность создания интерактивных программ, нужно сделать так, чтобы список символов, который мы строим в ходе выполнения программы, физически выдавался в поток вывода *по мере его построения*, а не весь сразу, когда его построение будет закончено.

В той единственной версии Хоупа, которой мы располагаем, имеется *интерактивная команда* `write`, аргументом которой служит произвольное выражение, имеющее тип `list(char)`. Команда `write` вычисляет свой аргумент и по мере того, как в ходе вычисления становятся доступны очередные его элементы (т. е. символы), отправляет их в поток стандартного вывода. По идее у команды `write` может быть даже дополнительный аргумент, указывающий, *в какой файл* записывать символы, но этот аргумент на самом деле бесполезен, и вот почему.

Команда `write` — это именно *команда* или, если угодно, ***директива***, то есть это не какая-то функция или что-то подобное; встречаться в телах функций она не может, точно так же, как внутри тел функций не могут, к примеру, встречаться слово `dec` или тройное тире. Иначе говоря, `write` нельзя «вызвать», она может находиться только на самом верхнем уровне программы.

Вроде бы мы можем дать в программе эту команду несколько раз, но вычисления, которые произойдут в результате каждого из них, никак не смогут повлиять друг на друга, ведь у нас здесь нет ни глобальных переменных, ни каких-либо других средств для передачи информации из одного независимого вычисления в другое — кроме значений, возвращаемых функциями, но в данном случае значением воспользуется сама команда `write`. Как следствие, единственный осмысленный способ оформления программы на Хоупе (в данной его версии) — это описать в тексте все функции, а потом в самом конце дать *одну* команду `write` для вызова некой «главной» функции. Эта функция должна иметь тип возвращаемого значения `list(char)`; в ходе своего вычисления она будет последовательно слева направо формировать список символов, которые надо напечатать.

Имея в своём распоряжении всего один поток вывода, мы вряд ли захотим принудительно направлять его в файл — в конце концов, пользователь может сделать это сам, перенаправив вывод. Конечно, невозможность работы более чем с одним потоком вывода — это ограничение столь серьёзное, что практической пользы от нашего интерпретатора можно не ожидать.

Отметим ещё один довольно странный момент. В выражении, служащем аргументом команде `write`, можно использовать имя `input` — оно здесь символизирует поток стандартного ввода. В других местах программы это имя недоступно, так что приходится использовать `read("")`.

Чтобы написать осмысленный пример программы, работающей с вводом-выводом, нам потребуется ещё несколько библиотечных функций. Функции `num2str` и `str2num` переводят соответственно число в его строковое представление и строку в число. Операция `<>` производит *конкатенацию* двух списков. Напомним, что слово `argv` обозначает список аргументов командной строки; его тип — `list(list(char))`.

Вернёмся теперь к нашему примеру с треугольником Паскаля (см. стр. 172) и оформим его в виде законченной программы, которая будет работать в трёх режимах: если в командной строке указано два параметра, программа будет рассматривать их как индексы n и k и выдавать значение C_n^k; если аргумент задан только один (n), программа будет печатать коэффициенты разложения для $(a+b)^n$, то есть n-ю строку треугольника Паскаля; если же аргументов не задано вовсе, программа будет работать интерактивно — напечатает первую строку треугольника (то есть просто число `1`), дождётся нажатия Enter, напечатает вторую строку, снова дождётся нажатия Enter и т. д., пока не наступит ситуация «конец файла».

Начнём, как водится, со строки, превращающей наш файл в скрипт:

```
#!/usr/local/bin/hopeless -f
! pascal_t.hop
```

После этого опишем функции для построения треугольника Паскаля и вычисления через него C_n^k, которые мы уже рассматривали в §11.5.12 — `NextLineRest`, `NextLine`, `PascalsTriangleBuild`, `PascalsTriangle`, `Nth` и `Cnk`.

Решить поставленную задачу нам поможет вспомогательная функция, преобразующая список чисел в строку, содержащую запись этих чисел через пробел; ясно, что «напечатать» (то есть попросту приделать к постепенно вычисляемому результату) строку треугольника Паскаля при наличии такой функции будет проще. Напишем её:

```
dec NumList2Str : list(num) -> list(char);
--- NumList2Str([]) <= [];
--- NumList2Str(n :: rest) <=
        num2str n <> " " <> NumList2Str rest;
```

Теперь из трёх режимов работы, которые мы включили в постановку задачи, два — выдача заданного элемента и заданной строки треугольника Паскаля — реализуются тривиально; остаётся последний и самый интересный — интерактивный. Его мы реализуем в виде отдельной функции, которая принимает на вход два параметра — поток ввода (список символов) и сам треугольник Паскаля (список списков чисел), а точнее — *остаток* треугольника Паскаля, который ещё не напечатан. Естественно, при первом вызове в роли этого остатка выступает весь треугольник, а дальше при рекурсивных вызовах от него будут отщепляться строки.

Работать наша функция будет следующим образом. Если поток ввода исчерпан (пустой список), то делать больше ничего не надо, можно просто вернуть пустой список (строго говоря — пустой список символов, но это неважно). Если первый символ на вводе — что угодно, кроме перевода строки, то делать, опять-таки, ничего не нужно, символ следует просто проигнорировать, для этого нашей функции достаточно рекурсивно вызвать саму себя с теми же параметрами, только уже без первого элемента потока ввода. Наконец, если первый символ потока ввода — перевод строки, то результат (то есть то, что выдаётся в поток вывода) будет состоять из текстового представления первой из строк, оставшихся в треугольнике, и результата печати остального треугольника при обработке оставшихся символов потока ввода; этот результат мы тоже получим очевидным рекурсивным вызовом. Функция, которую мы назовём `LineByLine`, получится такая:

```
dec LineByLine : list(char) # list(list(num)) -> list(char);
--- LineByLine([], _) <= [];
--- LineByLine(ch :: inp, h :: t) <=
        if ch = '\n' then
                NumList2Str h <> LineByLine(inp, t)
        else
                LineByLine(inp, h :: t);
```

Можно, впрочем, написать и вот так:

```
dec LineByLine : list(char) # list(list(num)) -> list(char);
--- LineByLine([], _) <= [];
--- LineByLine('\n' :: inp, h :: t) <=
        NumList2Str h <> LineByLine(inp, t);
--- LineByLine(_ :: inp, h :: t) <=
        LineByLine(inp, h :: t);
```

Какой из этих способов предпочесть — дело вкуса. Имея в своём распоряжении функцию `LineByLine`, мы можем написать главную функцию, которую так и назовём — `main`; у неё будет два параметра — список аргументов командной строки и поток стандартного ввода. С реализацией неинтерактивных режимов всё просто, что же касается режима интерактивного, то, если просто вызвать функцию `LineByLine` от потока ввода и треугольника Паскаля, она ничего не напечатает и будет ждать нажатия Enter, и уже после этого на экране появится «нулевая» строка треугольника, потом первая и т. д. Поэтому мы применим небольшую хитрость: к потоку ввода спереди добавим символ перевода строки, чтобы первую (точнее, нулевую) строку треугольника наша программа напечатала сразу же, не дожидаясь действий пользователя.

Итоговая функция `main` будет выглядеть так:

```
dec main : list(list(char)) # list(char) -> list(char);
--- main([], inp) <=
        LineByLine("\n" <> inp, PascalsTriangle);
--- main(n::[], _) <=
        NumList2Str(Nth(PascalsTriangle, str2num n));
--- main(n::k::[], _) <=
        num2str(Cnk(str2num n, str2num k));
--- main(lst, _) <=
        "Too many arguments, sorry";
```

Можно обратить внимание, что ни в одном из случаев наша функция не будет печатать завершающий символ перевода строки. Его мы добавим при вызове функции, который будет выглядеть так:

```
write main(argv, input) <> "\n";
```

Текст нашего скрипта на этом заканчивается. Заметим, что ввиду обсуждавшихся выше особенностей интерпретатора Hopeless *любая* программа, написанная для него, будет заканчиваться именно командой `write`, вызывающей некую главную функцию или, возможно, несколько функций.

11.5.14. Карринг

Этот и следующий параграфы могут показаться чрезмерно заумными. Вы можете их безболезненно пропустить — они нужны в основном для общей эрудиции, а на практике вряд ли потребуются, даже если вы решите всерьёз заняться практическим программированием на Хаскеле или каком-нибудь другом «ленивом» языке.

Отвлечёмся временно от языка Хоуп и рассмотрим функции как некие абстрактные «активные объекты», переводящие несколько аргументов (или, что то же самое, кортеж значений в роли единственного аргумента) в значение некоторого типа; иначе говоря, рассмотрим функции в математическом смысле.

Пусть у нас есть произвольная функция от *двух* аргументов $f(x, y)$, заданная каким-то выражением, в котором, естественно, фигурируют переменные x и y. Если очень захотеть (а зачем — станет ясно чуть позже), вычислять такую функцию для заданных значений x_0 и y_0 можно в два этапа: сначала подставить в выражение значение x_0 вместо x, получив таким образом *новую функцию* $\bar{f}_{x_0}(y) = f(x_0, y)$ (имеющую, заметим, уже *один* аргумент), а затем эту новую функцию применить к значению y_0. Например, для $f(x, y) = 10 * x + y$ в точке $(3, 5)$ имеем: $\bar{f}_3(y) = 30 + y$, $f(3, 5) = \bar{f}_3(5) = 35$.

Некоторые авторы заявляют, что это и есть ***карринг***, но они заблуждаются — мы пока что не ввели «каррированный аналог» функции f. Чтобы понять, о чём идёт речь, сделаем следующее формальное замечание. **Какова бы ни была функция f двух аргументов, ей можно поставить в соответствие функцию f^c *одного* аргумента, *значением* которой в любой точке x_0 будет *функция* (опять же, одного аргумента) $\bar{f}_{x_0}$ такая, что для любых значений x_0, y_0, входящих в область определения функции f, выполняется равенство** $(f^c(x_0))(y_0) = \bar{f}_{x_0}(y_0) = f(x_0, y_0)$. Функцию f^c (а вовсе не $\bar{f}$, как это часто кажется новичкам!) называют *результатом каррирования* исходной функции f. Подчеркнём ещё раз: её значением в точке x_0 является *функция* — как раз эту функцию-значение мы и называли $\bar{f}$. Что касается самого ***карринга*** как операции, то она применяется к функции двух аргументов (в нашем примере — f), а результатом её становится *функция одного аргумента, возвращающая*[40] *функцию одного аргумента*.

Операция карринга или, если угодно, каррирования (англ. *currying*) названа так в честь того же человека, что и язык Хаскель. Его полное имя — Хаскелл Брукс Карри (Haskell Brooks Curry). Интересно, что создатели языков программирования успели воспользоваться всеми тремя частями полного имени Х. Б. Карри — существуют языки программирования Брук (Brook) и Карри (Curry), также названные в его честь, не говоря уже про операцию каррирования. Пожалуй, мы вряд ли найдём другую историческую фигуру, в честь которой в программировании было бы столько всего названо. При этом сам Карри не был, собственно говоря, программистом — он занимался математической логикой, исследовал вопросы теории вычислимости и считается создателем комбинаторной логики, а свои основные работы опубликовал задолго до начала компьютерной эры.

Возвращаясь к Хоупу, мы можем догадаться, что операция каррирования, применимая к произвольной функции двух аргументов, реализуется, в сущности, совершенно тривиально. В самом деле, пусть у нас есть произвольная функция `f` двух аргументов; обозначим их типы через `alpha` и `beta`, а тип возвращаемого значения — через `gamma`, т. е. функция `f` будет иметь тип `alpha # beta -> gamma`. Нам нужно написать такую функцию `Curry`, которая (в терминах, введённых выше) осуществит переход от f к f^c, то есть, имея f в качестве аргумента, построит и вернёт такую *функцию* (ту самую f^c), которая, если ей дать значение первого параметра (типа `alpha`), вернёт *функцию* одного аргумента, принимающую значение типа `beta` и возвращающую значение типа `gamma`. С использованием безымянных функций, создаваемых словом `lambda`, это можно сделать, как говорится, в один ход:

```
dec Curry : (alpha # beta -> gamma) -> (alpha -> (beta -> gamma));
--- Curry f <= (lambda x => (lambda y => f(x, y)));
```

Для наглядности мы здесь расставили скобки, чтобы показать приоритет операций, и не стали сокращать слово `lambda`; но если учесть, что символ `->` в записи типа правоассоциативен, символ `=>` всегда относится к ближайшей «лямбде», символ `<=` имеет приоритет ниже, чем любые операции, включая и операцию `lambda`, а сама «лямбда» может быть обозначена символом обратного слэша, то написать можно и иначе:

```
dec Curry : (alpha # beta -> gamma) -> alpha -> beta -> gamma;
--- Curry f <= \x => \y => f(x, y);
```

Опытные функциональщики обычно так и пишут, и притом утверждают, что так понятнее; правы они или нет — решайте сами.

Чтобы понять, к чему мы в итоге пришли, рассмотрим простенькую функцию двух аргументов:

```
dec Func : num # num -> num;
--- Func(x, y) <= 10 * x + y;
```

Обычный вызов этой функции делается с помощью кортежа — например, выражение `Func(2,3)` даст значение `23`. Вычислив выражение `Curry Func`, мы получим каррированный вариант нашей функции; интерпретатор его напечатает вот в таком виде:

```
lambda x => lambda y => Func (x, y) : num -> num -> num
```

[40] Да простит нас читатель за эту вольность в терминологии. Конечно, в математике функции ничего не «возвращают», это сугубо программистское выражение; но если мы здесь продолжим упорно притворяться математиками, шансы что-то понять от этого только снизятся.

На верхнем уровне тут, как видим, `lambda` с одним аргументом, что полностью соответствует нашим ожиданиям, ведь каррированная функция — это функция одного аргумента, возвращающая функцию. Если её применить к аргументу `2`, то есть вычислить выражение `Curry Func 2`, получим вот что:

```
lambda y => Func (2, y) : num -> num
```

Осталось добавить второй аргумент, то есть вычислить выражение `Curry Func 2 3` — и результатом станет уже знакомое нам `23`.

Мы могли бы написать функцию `Func` сразу в каррированном виде, чтобы не нужно было вызывать `Curry` (не забываем, что «`num -> num -> num`» — это то же самое, что «`num -> (num -> num)`»):

```
dec Func : num -> num -> num;
--- Func x y <= 10 * x + y;
```

Вызывать её, как раньше, «со скобочками» теперь нельзя, зато можно без всяких скобочек: выражение `Func 4 7` даст `47`. Функциями двух аргументов тут дело не ограничивается, следующее описание

```
dec Func3 : num -> num -> num -> num;
--- Func3 x y z <= 100 * x + 10 * y + z;
```

тоже корректно, выражение `Func3 7 3 6` вернёт `736`. Ограничений тут никаких нет, можно в таком виде сделать функцию любого количества аргументов. У людей, привыкших к языкам с энергичной стратегией вычисления, часто вызывает недоумение такой подход к построению функций, ведь здесь для вычисления простенького арифметического выражения были созданы два *функциональных объекта* (***замыкания***; см. §11.1.9), причём созданы они были лишь затем, чтобы тут же исчезнуть, и вроде бы такая «бесхозяйственность» должна быть неэффективна; но вспомним, что здесь *любое* выражение интерпретатор таскает за собой, пока его значение не потребуется в явном виде, и лишь тогда его вычисляет, а такое «невычисленное» выражение само по себе вынуждено быть замыканием, так что от карринга эффективность вряд ли может пострадать *ещё больше*, чем она уже пострадала от ленивости как таковой.

Программисты, работающие на Хаскеле, обычно вообще не применяют кортежи в роли параметров функций, несмотря на то, что кортежи в Хаскеле есть, выглядят точно так же, как в Хоупе, и точно так же позволяют создавать функции многих аргументов; вместо этого они предпочитают писать все функции в исходно каррированном виде, некоторые даже не помнят, что можно как-то иначе.

Впрочем, мы не изучаем Хаскель и не ставим себе цели как-то подготовиться к такому изучению; вообще говоря, мы вполне могли бы обойтись без рассказа о карринге, если бы функции именно в таком виде не требовались нам для демонстрации рекурсии безымянных функций через комбинатор неподвижной точки; этому будет посвящён следующий параграф.

11.5.15. Комбинатор неподвижной точки

В §11.5.10, посвящённом операции `letrec`, мы рассмотрели один из возможных способов устроить рекурсию на безымянных функциях, но у этого способа есть одна особенность: имя-то функции всё же даётся, просто оно локальное.

Математики, создававшие теорию алгоритмов и теорию вычислимости, такой роскоши себе позволить не могли, поскольку сам факт существования имён «портит» используемый формализм; задавать новые функции в формализме, допускающем имена, конечно, проще, но вот исследовать свойства абстрактных объектов и доказывать теоремы (чем, собственно говоря, и занимаются математики — в отличие от программистов) будет, напротив, намного сложнее. Поэтому наиболее популярный формализм, используемый в теории вычислимости — ***лямбда-исчисление***, предложенное Алонсо Чёрчем — оперирует как раз безымянными функциями.

Собственно говоря, загадочное слово `lambda`, обозначающее безымянные функции в Лиспе, Scheme и Хоупе, взято из названия лямбда-исчисления, а само исчисление названо так, потому что в нём греческая буква λ (*лямбда*) используется для обозначения функции. Мы не будем рассматривать лямбда-исчисление, хотя это, конечно, очень интересная (в том числе с программистской точки зрения) математическая абстракция. Прекрасное введение в эту область математики есть, например, в книге [24]; внятные тексты на эту тему можно найти во многих других источниках, в том числе доступных в Интернете.

Рекурсию устроить оказалось возможно и здесь; для этого используются так называемые ***комбинаторы неподвижной точки***. Чтобы понять всю сопутствующую теорию, потребуется довольно много времени и сил, и это в принципе дело на любителя; надо сказать, что такие любители довольно часто встречаются среди апологетов программирования на Хаскеле. Так или иначе, к программированию (даже на Хаскеле) всё это имеет отношение весьма далёкое, да и объём книги не безграничен, поэтому мы ограничимся рассмотрением рекурсии через комбинатор неподвижной точки в сугубо прагматическом аспекте.

Общая идея состоит в том, что функцию, которая должна вызвать сама себя, снабжают дополнительным (обычно первым) параметром, через который ей передают функцию, которую нужно вызвать — то есть, собственно говоря, её саму. Остаётся превратить такую «недоделанную» функцию в ту, которая нам нужна, и как раз для этого применяется пресловутый комбинатор неподвижной точки. На вопрос, что это такое, чаще всего можно получить ответ, что это такая функция высшего порядка (функционал), которая, если ей передать функцию f в качестве аргумента, находит её неподвижную точку, то есть такое значение x, что $f(x) = x$. При этом нам, впрочем, кое-что недоговаривают, но об этом позже.

Кстати, довольно часто «забывают» пояснить и само слово «комбинатор», но с этим как раз проще: комбинаторами в теории вычислимых функций называют произвольные функции, не имеющие в своём теле вхождений свободных переменных. Вообще-то этот термин применяют не ко всем функциям, удовлетворяющим такому определению, но ничего более конкретного вы от любителей всей этой экзотики всё равно не добьётесь.

Вооружившись определением неподвижной точки и обозначив искомый комбинатор буквой F, получим, что $F(f) = x$, где $x = f(x)$; подставив первое в обе части второго, получим $F(f) = f(F(f))$. От x мы избавились, у нас остались только функции. Полученное выражение вполне можно перевести на язык Хоуп, только сам комбинатор мы в этот раз обозначим `Fix`:

```
--- Fix f <= f(Fix f);
```

Попытаемся сообразить, каков должен быть тип функции `Fix`, ведь он нам нужен для директивы `dec`. На входе у нас функция, причём из определения неподвижной точки следует, что тип аргумента и тип значения у этой функции совпадают; обозначим этот тип за `alpha`. Тогда получится, что сам «комбинатор» тоже будет возвращать значение типа `alpha`, ведь его возвращаемым значением служит то, что вернула функция `f`. Получаем

```
dec Fix : (alpha -> alpha) -> alpha;
```

Если эти директивы поместить в программу (естественно, в обратном порядке — сначала `dec`, потом `---`), то полученная версия комбинатора неподвижной точки *будет работать*, превращая безымянную функцию двух аргументов (правда, только заданную в каррированной форме) в функцию одного аргумента частичным применением к значению, в роли которого сама эта функция — сколь бы странным это ни казалось. Чтобы в этом убедиться, попробуем посчитать с помощью безымянной функции сумму списка чисел[41].

Безымянную «функцию-заготовку», которую потом надо будет подать комбинатору в качестве аргумента, мы напишем в каррированном виде, как это было описано в предыдущем параграфе. Выше было сказано, что она должна получать дополнительный аргумент, через который ей будет передаваться та функция, которую надо вызвать для решения «чуть более простой задачи» (в данном случае — для суммирования остатков списка). Обозначим этот аргумент словом `fn`; тогда собственно функция, которая считает сумму списка, обращаясь к некой внешней функции `fn` за подсчётом остатка суммы, должна выглядеть так: `lambda [] => 0 | a::d => a + fn d`. Чтобы в окружении всегда присутствовала переменная `fn`, мы поместим это выражение ещё в одну операцию `lambda`, у которой `fn` будет выступать в роли имени параметра; получим такое выражение:

```
lambda fn => lambda [] => 0 | a::d => a + fn d
```

Это и есть искомая заготовка. Осталось применить к ней наш комбинатор — и можно будет считать сумму списка. Например, выражение

[41] Этот пример (с этой же целью) приводят авторы книги [24]; в подавляющем большинстве других текстов, посвящённых комбинатору неподвижной точки, используется рекурсивный подсчёт факториала, но у автора этих строк факториал как пример рекурсии вызывает отвращение; причины этого см. в §9.2.2.

```
Fix (lambda fn => lambda [] => 0 | a::d => a + fn d) [1,2,3]
```

даст результат `6`. Попробуем проследить, как это получилось; `lambda`-выражение в круглых скобках обозначим для краткости φ. Итак, на первом шаге имеем

$$\varphi\texttt{(Fix }\varphi\texttt{) [1,2,3]}$$

Поскольку функция φ (точнее, внешняя из её лямбд) — нестрогая по аргументу, на следующем шаге появится замыкание с телом от внутренней лямбды, которое в качестве значения `fn` получит невычисленное `(Fix` φ`)`. Замыкания, полученные из внутренней лямбды, мы обозначим буквой ψ, а значение `fn` будем указывать в её нижнем индексе. Итак, на следующем шаге у нас будет вот что:

$$\psi_{\texttt{Fix }\varphi}\ \texttt{[1,2,3]}$$

(где ψ соответствует `lambda [] => 0 | a::d => a + fn d`). Этой функции одного аргумента ничто не мешает примениться к списку, а поскольку он не пуст, сработает вторая ветка лямбда-выражения, переменная `a` получит значение `1`, переменная `d` — значение `[2,3]`, `fn` у нас по-прежнему `Fix` φ, так что получаем:

$$\texttt{1 + ((Fix }\varphi\texttt{) [2,3])}$$

Круглые скобки мы здесь расставили для наглядности, в действительности они не нужны, поскольку у сложения приоритет ниже, чем у операции применения функции, а сама операция применения левоассоциативна. Следующим должно выполняться сложение, но оно строгое по обоим аргументам, так что сначала должно быть вычислено выражение справа от него. Дальнейшие шаги нам уже знакомы, только список теперь на один элемент короче:

$$\texttt{1 + }\varphi\texttt{(Fix }\varphi\texttt{) [2,3]}$$

$$\texttt{1 + }\psi_{\texttt{Fix }\varphi}\ \texttt{[2,3]}$$

$$\texttt{1 + 2 + ((Fix }\varphi\texttt{) [3])}$$

Осталось ещё два круга:

$$\texttt{1 + 2 + }\varphi\texttt{(Fix }\varphi\texttt{) [3]}$$

$$\texttt{1 + 2 + }\psi_{\texttt{Fix }\varphi}\ \texttt{[3]}$$

$$\texttt{1 + 2 + 3 + ((Fix }\varphi\texttt{) [])}$$

$$\texttt{1 + 2 + 3 + }\varphi\texttt{(Fix }\varphi\texttt{) []}$$

$$\texttt{1 + 2 + 3 + }\psi_{\texttt{Fix }\varphi}\ \texttt{[]}$$

В этот раз список пуст, так что внутри ψ сработает первая ветка, которая без всякой рекурсии даст `0`; из нашего выражения исчезло всё сложное, осталось только `1 + 2 + 3 + 0`, из которого и получится искомое `6`.

Как показывает опыт, даже этот довольно подробный разбор происходящего не всем и не всегда помогает понять творящуюся магию. Попробуем подойти к объяснению с другой стороны. Если начать целенаправленно вычислять функцию `Fix`, не обращая внимания на ленивость стратегии, мы получим что-то вроде следующего:

```
Fix f = f(Fix f) = f(f(Fix f)) = f(f(f(Fix f))) = ...
        ... = f(f(f(...(Fix f)...))) = ...
```

и так до бесконечности. Как можно заметить, комбинатор неподвижной точки просто в силу своего определения превращает исходно нерекурсивную функцию `f` в цепочку вызовов, напоминающих последовательность рекурсивных обращений её к самой себе. Кстати, именно «напоминающих», ведь в действительности она здесь сама к себе не обращается, вместо этого `f` *получает на вход результат её же работы* (над каким-то другим аргументом). По своей структуре полученная последовательность вызовов скорее напоминает свёртку, чем собственно рекурсивные вычисления.

Следует подчеркнуть, что получаемая последовательность вызовов — именно бесконечная, а не какая-то другая. Будь функция `f` строгой по своему аргументу или будь она хотя бы просто функцией одного аргумента, *зависящей* от этого своего аргумента — что автоматически означает её строгость по своему единственному аргументу — применение к ней функционала `Fix` неминуемо привело бы к бесконечной рекурсии и, как следствие, к аварийному завершению программы.

Мы смогли заставить всё это заработать, потому что, во-первых, мы используем ленивый язык и, во-вторых, наша функция `f` имеет два аргумента, при этом построена так, что её можно применить к одному из них без второго (спасибо каррингу), и к тому же она по этому аргументу нестрогая. В результате потенциально бесконечная цепочка дальнейших вызовов `f`, построенная с помощью комбинатора `Fix`, оказывается на некоторое время сохранена в виде невычисленного выражения, пока функция `f` анализирует свой второй аргумент и решает, нужно ли ей вообще значение аргумента первого — того, который используется в роли функции, к которой следует обратиться для решения «чуть более простой задачи» *подобно* рекурсивному вызову (да, строго говоря, здесь нет рекурсивного вызова! — единственная рекурсия здесь имеет место внутри `Fix`). В какой-то момент задача оказывается настолько простой, что для её решения никакая внешняя помощь уже не требуется (аналог базиса рекурсии, в нашем случае — вычисление суммы пустого списка), так что оставшаяся (по-прежнему бесконечная) цепочка вызовов `f` исчезает, так и оставшись невычисленной.

Пожалуй, отдельного обсуждения заслуживает само по себе название «комбинатор неподвижной точки». Много где встречается набившее оскомину утверждение вроде того, что это «функционал, который находит неподвижную точку заданной функции»; иногда для верности добавляют, что это «наименьшая» неподвижная точка. Можете, впрочем, быть уверены, что какую бы вы функцию ни взяли, никакой неподвижной точки для неё этот комбинатор не найдёт — вместо этого он устроит вам очередную аварию по переполнению стека, и на этом всё кончится. Например, для функции `\x=>20-x` неподвижной точкой в обычном математическом смысле будет значение `x=10`, но `Fix` этой функцией подавится, как и любой другой числовой функцией, пусть даже имеющей (сколь угодно очевидную) неподвижную точку. Единственное исключение — константы, то есть функции одного аргумента, возвращающие всегда одно и то же значение вне зависимости от аргумента; очевидно, что применение `Fix` к такой функции приведёт к отбрасыванию всей бесконечной «пирамиды» невычисленных вызовов на первом же шаге; но такой вырожденный случай не слишком интересен.

Комбинатор неподвижной точки оправдывает своё название лишь в пространстве функций, введённых в так называемой теории доменов; без изучения этого не слишком известного широкой публике раздела математики оказывается довольно трудно привести пример вида «смотрите, вот функция, вот у неё неподвижная точка, вот мы применили к функции комбинатор и он нам неподвижную точку нашёл» (если только нас не устроят константы). Интересно, что с учётом теории доменов `Fix` оказывается вполне себе применим даже к вышеупомянутой `\x=>20-x`, просто «бесконечное зацикливание» там рассматривается как одно из допустимых значений, притом ме́ньшее любого другого, а применение любой всюду определённой неконстантной функции одного аргумента к этому «значению», очевидно, даст его же само — если уж вычисление аргумента зацикливается, то вычисление применения функции к такому аргументу зациклится и подавно. Константа в этом плане оказывается «самым хитрым зверем в лесу»: для неё «зацикливание» не является неподвижной точкой, ведь ей всё равно, к чему её применили; но всем остальным функциям *одного* аргумента требуется вычислить значение своего аргумента — просто чтобы *не быть константами*.

Впрочем, возможно, всё не так плохо. Вернёмся к нашей функции

$$\varphi = \texttt{lambda fn => lambda [] => 0 | a::d => a + fn d}$$

Если считать её функцией одного (первого) аргумента, возвращающей функцию, то внезапно функция `Sumlist`, считающая сумму элементов списка и написанная самым простым из возможных способов (с учётом отсутствия циклов):

```
dec Sumlist : list(num) -> num;
--- Sumlist [] <= 0;
--- Sumlist (x :: r) <= x + Sumlist r;
```

окажется именно что неподвижной точкой для φ. В самом деле, значением выражения

```
(lambda fn => lambda [] => 0 | a::d => a + fn d) Sumlist
```

будет новая безымянная функция — вот такая:

```
lambda [] => 0 | a :: d => a + Sumlist d
```

Нетрудно видеть, что для любого списка чисел эта функция вернёт такое же значение, как и просто `Sumlist`, а значит, *в математическом смысле* это она и есть, т. е. φ`(Sumlist) = Sumlist` в полном соответствии с определением неподвижной точки. Между тем, выражение `Fix` φ нам тоже построило функцию, вычисляющую сумму списка, то есть в математическом смысле равную функции `Sumlist` — иначе говоря, `Fix` всё-таки нашёл неподвижную точку своего аргумента (функции φ). Спасибо и на том.

Возвращаясь к операции `letrec` из §11.5.10, отметим ещё один любопытный факт. Согласно утверждению Росса Патерсона из [26], выражение

$$\texttt{letrec}\ V\ \texttt{==}\ E\ \texttt{in}\ B$$

семантически эквивалентно выражению

$$\texttt{(lambda}\ V\ \texttt{=>}\ B\texttt{)(Fix (lambda}\ V\ \texttt{=>}\ E\texttt{))}$$

— только более эффективно. Проверить это утверждение предложим читателю в качестве упражнения.

Часть 12

Компиляция, интерпретация, скриптинг

К настоящему моменту мы успели познакомиться с довольно внушительным набором языков программирования. Некоторые из них — Паскаль, Си, Си++ и, конечно, язык ассемблера — вне всякого сомнения относятся к компилируемым. С другими языками, как мы уже обсуждали, ситуация сложнее. Тот же SBCL вроде бы позволяет получить исполняемый файл, но такого размера, что на практике это бессмысленно, один исполняемый файл получается больше, чем вся инсталляция SBCL вместе с исходниками вашей программы. Очевидно, что создатели SBCL считают основным способом выполнения программ их запуск через SBCL, но при этом настойчиво утверждают, что их реализация — компилирующая, основывая это утверждение на том, что функции Лиспа перед их выполнением переводятся в машинный код. Реализации Scheme обычно действительно *компилируют* программу, используя язык Си в роли промежуточного, но при этом получаемые исполняемые файлы либо зависят от динамических библиотек, включающих в себя целиком все компоненты интерпретатора, либо — если удаётся задействовать статическую сборку — включают в себя изрядную часть реализации Scheme и тоже оказываются довольно велики по объёму. Вопрос, можно ли считать это полноценной компиляцией (ведь интерпретатор всё равно присутствует во время исполнения, просто он включён внутрь исполняемого файла, пусть даже и не весь), оставим пока открытым.

Так или иначе, способ исполнения программы, как будет показано ниже, несомненно влияет на то, как программист воспринимает свою программу, и это позволяет говорить о компиляции и интерпретации как об особого рода *парадигмах*. Этот вопрос мы обязательно обсудим позже, но пока для полноты картины рассмотрим класс языков программирования, интерпретируемая сущность которых не вызывает сомнений — так называемые ***командно-скриптовые*** языки программирования.

12.1. Характерные особенности скриптовых языков

Изначально командно-скриптовые языки программирования предназначались для *управления заданиями*, то есть для указания операционной системе, что и в каком виде нужно исполнить. С одним таким языком мы уже хорошо знакомы, причём встретились с ним ещё в первой части первого тома, то есть раньше, чем с Паскалем, и активно использовали его в работе на протяжении всех томов нашей книги, при этом часто забывая, что вообще имеем дело с языком программирования. Имеется в виду язык командного интерпретатора — Bourne Shell. Именно на нём мы всегда давали команды компьютеру, средствами этого языка мы запускали редакторы текстов, компиляторы, отладчики и сами написанные нами программы.

Как мы видели в первой части (см. т. 1, §1.2.15), на этом языке можно создавать ***скрипты***, то есть попросту *программы*, состоящие из команд и призванные автоматизировать часто выполняемые в системе нетривиальные процедуры; для этого Bourne Shell предусматривает переменные и оператор присваивания, а также конструкции для ветвления и цикла (всё это мы успели рассмотреть). Добавим, что возможности скриптового программирования на Bourne Shell мы рассмотрели далеко не полностью. Так, цикл мы рассматривали только один (`while`), но их там на самом деле три — есть ещё `until` и `for`, хотя на аналоги из других языков они совершенно не похожи; имеется оператор выбора (`case`); есть даже подпрограммы (функции), позволяющие при желании устроить рекурсию, хотя лучше этого здесь не делать — на каждый вызов функции интерпретатор порождает процесс.

Пример Bourne Shell прекрасно иллюстрирует основные особенности командно-скриптового программирования как явления. Прежде всего следует выделить интерпретируемое исполнение: эффективность здесь мало кого волнует, ведь программа на таком языке *управляет другими программами* (во всяком случае, тот же Bourne Shell предназначен именно для этого), так что время, затраченное интерпретатором на исполнение конструкций программы («скрипта»), в любом случае будет пренебрежимо мало в сравнении со временем выполнения запускаемых внешних программ, ради которых всё и делается. Отсюда довольно естественно вытекает второй принцип — «всё есть строка»: в типичных командно-скриптовых языках нет никаких иных данных, кроме текстовых строк. Информация любого рода, будь то числа или что-то ещё, представляется именно строками. В таких условиях практически неизбежно появляются развитые средства анализа и преобразования строк; мало того, обычно в командно-скриптовых языках *строка обозначает сама себя*, то есть не нужно (во всяком случае, не обязательно) заключать строку в какие-то кавычки или апострофы; лишь некоторые символы имеют особый смысл, причём не всегда. Когда строка обозначает команду, она при этом *остаётся строкой*, просто её помещают в такой контекст, в котором интерпретатор выполняет её в качестве команды; вспомним, что в лиспоподобных языках S-выражение тоже остаётся S-выражением, когда его вычисляют в качестве формы. Всевозможные кавычки, апострофы и прочие ограничители в командно-скриптовых языках тоже применяются, но обозначают некие особые случаи строк: например, в Bourne Shell двойные кавычки позволяют объединить несколько слов в одну строку, то есть временно лишить пробел его «разделяющего» свойства, одиночные апострофы блокируют эффект спецсимволов (подстановку имён файлов, обращения к переменным и прочее), а обратные апострофы используются для подстановки результата выполнения команды, когда текст, выданный командой, становится частью командной строки для другой команды.

Из того, что строки обозначают сами себя, вытекает довольно любопытное следствие, о природе которого программисты часто не задумываются. **Во многих командно-скриптовых языках *имя переменной* и *обращение к переменной* выглядят по-разному**, или, если говорить совсем строго, **обращение к переменной представляет собой операцию, которую программист должен указать явно**. Традиционно для обозначения этой операции используется символ `$`. Дело тут в том, что *имя переменной — это тоже строка*, и она может потребоваться сама по себе, например, для вывода на экран или для любых других целей. В частности, в Bourne Shell для присваивания переменной используется только её имя, примерно так:

```
MY_VARIABLE=256
```

(здесь очень важно отсутствие пробелов вокруг знака равенства, иначе интерпретатор воспримет и имя, и сам знак равенства как обычные строки); при этом ***обращение*** к этой переменной можно записать как `$MY_VARIABLE` или `${MY_VARIABLE}` — например, команда

```
echo $MY_VARIABLE
```

напечатает *значение* переменной, то есть в нашем случае `256`, тогда как команда

```
echo MY_VARIABLE
```

просто напечатает строку `MY_VARIABLE`, не воспринимая её как имя переменной. Необходимость явного обозначения обращений к переменным выглядит непривычно для программистов, ранее не имевших опыта скриптового программирования; но, если подумать, это даже логичнее, чем поступать, как поступают трансляторы других языков — догадываться из контекста, что имеется в виду не сама переменная, а её значение. В самом деле, переход от переменной к её значению — это *действие* (например, загрузка информации из соответствующей области памяти). Вспомните, что при работе на языке ассемблера NASM нам тоже приходилось явным образом отличать обращение к переменной, т. е. к области памяти, помеченной меткой, от самой метки — мы использовали для этого квадратные скобки, обозначая таким образом операнды типа «память».

Поскольку сама программа — это тоже текст, в командно-скриптовых языках обычно присутствует уже знакомый нам по диалектам Лиспа `EVAL` —

вызов интерпретатора для произвольного текста, в том числе созданного во время выполнения. В Лиспе и Scheme эта возможность тоже вытекает из единства представления программы и данных, но там в роли этого представления выступают S-выражения, которые хоть как-то структурированы; задачи управления в таком структурировании не нуждаются. Вообще ***метапрограммирование*** для командно-скриптовых языков — нечто само собой разумеющееся, и здесь не требуется пространных рассуждений о цене, которая за это будет заплачена (см. стр. 118), ведь исполнение здесь в любом случае интерпретируемое.

Наконец, природа «командных» задач требует максимально простого (с точки зрения трудоёмкости) доступа к внешним программам — к их запуску, всевозможному комбинированию, перенаправлению ввода-вывода и анализу выдаваемой работающими программами информации. Эти возможности сохраняются в том числе и в тех языках командно-скриптовой группы, которые не предназначены для использования в диалоговых интерпретаторах командной строки: по-видимому, дело здесь в том, что текст как универсальное представление данных (и самой программы) идеально сочетается с классическим принципом unix-систем «всё есть текст» в применении к обычным программам, на каких бы языках они ни были написаны.

12.2. Язык Tcl

Название языка Tcl образовано от слов *tool command language*[1], но, хотя это и аббревиатура, пишется название только с одной заглавной буквой *T*, буквы *c* и *l* всегда оставляют строчными. Название произносится как «тикль» или как «ти-си-эль», оба прочтения традиционно считаются одинаково допустимыми.

Язык создан Джоном Оустерхаутом (John Ousterhout); первая версия была представлена публике в 1988 году в уже знакомом читателю университете Беркли (напомним, что семейство unix-систем под общим названием BSD тоже появилось в Беркли). Язык исходно предназначался для встраивания в качестве управляющего в приложения, для настройки которых не хватает обычных конфигурационных файлов, но достаточно быстро стал применяться и в других областях. Популярность Tcl существенно возросла с появлением библиотеки Tk, предназначенной для быстрого создания графических пользовательских интерфейсов; эта библиотека часто используется в программах на языках, отличных от Tcl, таких как Python, Perl, Haskell и других, при этом доступ к возможностям Tk производится через интерпретатор Tcl.

К настоящему моменту кроме основной реализации Tcl существует ещё несколько альтернатив, таких как Jim и TinyTcl, но ни сам язык Tcl, ни связка Tcl/Tk пока что не привлекли внимания стандартизаторов. К сожалению, это обстоятельство не смогло предохранить инструмент от разбухания и утраты концептуальной целостности; версии 8.5 и 8.6, вышедшие соответственно в 2007 и 2012 гг., содержат множество чужеродных возможностей и производят довольно мрачное впечатление.

К счастью, авторы Tcl внимательно относятся к сохранению обратной совместимости, так что ныне поддерживаемые версии интерпретаторов прекрасно работают со скриптами, написанными для предыдущих версий. Это позволит нам ограничиться лишь «классическими» возможностями самого Tcl и его библиотеки Tk в том виде, в котором они существовали в версии 8.4.

12.2.1. Интерпретатор и простейшие программы

Интерпретатором языка Tcl служит программа, обычно называющаяся `tclsh` (*Tcl shell*); чтобы узнать, где она находится в вашей системе, можно дать команду `which tclsh`; в большинстве случаев искомый полный путь — `/usr/bin/tclsh`:

```
avst@host:~$ which tclsh
/usr/bin/tclsh
```

Итак, программы (скрипты), написанные на этом языке, можно начинать со строчки `#!/usr/bin/tclsh`, чтобы система автоматически запускала нужный интерпретатор. Начнём с программы «Hello, world»; для печати строки воспользуемся командой `puts`:

```
#!/usr/bin/tclsh
# hello.tcl
puts "Hello, world"
```

Отметим, что интерпретатор Tcl считает строки, первым непробельным символом в которых выступает #, комментариями. Сохранив этот текст в файле `hello.tcl` и сделав, как обычно в таких случаях, наш файл исполняемым с помощью команды `chmod +x hello.tcl`, запустим полученную программу:

```
avst@host:~$ vim hello.tcl
avst@host:~$ cat hello.tcl
#!/usr/bin/tclsh
# hello.tcl
puts "Hello, world"
avst@host:~$ chmod +x hello.tcl
avst@host:~$ ./hello.tcl
Hello, world
avst@host:~$
```

В тексте man-страницы к интерпретатору `tclsh` приведена несколько неожиданная рекомендация по оформлению таких скриптов: вместо явного указания пути к интерпретатору `tclsh` его авторы предлагают использовать стандартный `/bin/sh`, а вторую строку скрипта сделать «комментарием», причём её конец «заэкранировать» символом «\», а следующей строкой приказать `sh`'у выполнить `exec`, запустив `tclsh` вместо себя. Выглядит это примерно так:

```
#!/bin/sh
# hello2.tcl \
exec tclsh "$0" "$@"

puts "Hello, world"
```

Эта конструкция основывается на том, что «экранирование» конца строки, содержащей комментарий, предусмотрено в Tcl, но не предусмотрено в Bourne Shell. Поэтому `/bin/sh` считает комментарием только вторую строку, а третью рассматривает как обычную команду — в данном случае `exec`. Эта команда в Bourne Shell делает то же, что и знакомые нам по языку Си функции семейства `exec` — заменяет исполняющуюся программу на другую, указанную в параметрах. Если вспомнить, что `$0` в Bourne Shell означает имя программы («нулевой» параметр командной строки), а `$@` — строку, составленную из всех аргументов командной строки, за исключением «нулевого», мы поймём, что `sh` вместо себя запустит в данном случае `tclsh`, причём параметрами ему передаст имя скрипта и все остальные аргументы, если они есть. Когда же `tclsh` получит управление и прочитает файл скрипта, он первые *три* строки проигнорирует как комментарии: первые две — поскольку они начинаются с #, а третью — потому что она с его точки зрения является продолжением предыдущей (то есть продолжением комментария).

Авторы man-страницы утверждают, что этот подход лучше обычного по трём причинам. Во-первых, полный путь к `tclsh` при этом не нужно указывать в скрипте, так что скрипт будет работать, если его перенести в систему, где этот путь отличается (отметим, что в системе FreeBSD, скорее всего, программа окажется в другом месте — `/usr/local/bin/tclsh`; то же самое произойдёт и в Linux, если вам придёт в голову собрать `tclsh` из исходных текстов и инсталлировать с параметрами по умолчанию) — нужно только, чтобы `tclsh` находился в одной из директорий, перечисленных в `PATH`. Во-вторых, как мы знаем, длина первой строки скрипта с указанием интерпретатора может быть ограничена, причём в некоторых системах ограничение составляет всего 32 символа (см. т. 1, §1.2.15); это может стать проблемой, например, в Mac OS X, где часто используются весьма длинные имена директорий. Кроме того, в таком варианте мы можем передать интерпретатору `tclsh` произвольное количество опций командной строки, тогда как при обычном подходе мы ограничены всего одним параметром. Наконец, в некоторых системах программа `tclsh` сама представляет собой скрипт, а ядро ОС может запрещать использовать скрипт в роли интерпретатора (хотя, прямо скажем, автору этих строк такое ни разу не встречалось — во-первых, обычно под именем `tclsh` в системе установлен сам интерпретатор, а во-вторых, тот же Linux этого ограничения не имеет).

В дальнейших примерах мы будем использовать подход, рекомендованный авторами Tcl, но если он вам не вполне понятен — можете его не применять; в большинстве случаев нет ничего страшного в указании `tclsh` интерпретатором напрямую.

Как можно заметить, команды в Tcl отделяются друг от друга переводом строки, то есть, попросту говоря, каждая строка программы-скрипта считается отдельной командой, если этому явным образом не противодействовать. При желании можно разместить несколько команд в одной строке, разделив их символом «`;`»:

```
puts "Hello, world" ; puts "Good bye, world"
```

[1] Это может быть переведено с английского как «инструментальный командный язык» либо как «язык инструментальных команд».

Если для записи команды не хватает одной строки, можно объявить следующую строку продолжением текущей, «заэкранировав» перевод строки символом «\», причём, как мы видели, это работает в том числе и для комментариев. Если возникает желание написать комментарий *к строке*, не выделяя для него отдельную строку, это можно сделать благодаря всё той же точке с запятой — показав, что команда закончилась, а затем написав символ «#», примерно так:

```
puts "Hello" ; # this will print ‘‘Hello’’
```

Без точки с запятой это не сработает — символ решётки должен быть первым значащим символом очередной команды, в противном случае он рассматривается как обычный символ. Например, команда

```
puts #
```

выдаст символ решётки на стандартный вывод.

Интерпретатор `tclsh` можно использовать не только для выполнения скриптов, но и в интерактивном режиме. Для этого его запускают без параметров. Для удобства рекомендуем читателю воспользоваться уже знакомой нам командой `rlwrap` (см. стр. 110).

Интерактивная работа с интерпретатором напоминает то, как это делалось с Лиспом и Scheme: интерпретатор печатает приглашение (обычно символ `%`, но это можно изменить); мы набираем *команду* (признаком конца команды считается символ перевода строки, если только мы не начали набирать какую-нибудь многострочную конструкцию), интерпретатор выполняет её и выдаёт нам *результат* — что может быть несколько неожиданно для пользователя, привыкшего к Bourne Shell. В этом плане *выполнение команды* Tcl напоминает скорее *вычисление S-выражения* в Лиспе: у каждой команды есть *результат*, представленный строкой текста (впрочем, это только потому, что ничего другого, кроме строк, в Tcl нет).

Например, команда `expr` вычисляет арифметическое выражение, представленное в виде одной или нескольких строк:

```
% expr 2*3
6
% expr "10 + 20"
30
% expr "10 + 20" + 30
60
% expr "10 + 20" "+ 30"
60
%
```

Сама команда `expr` ничего не печатает (что и понятно, ведь это не её дело); то, что мы видим — это результат выполнения команды, *напечатанный интерпретатором*.

Другой интересный пример команды — `exec`; она исполняет произвольную команду системы. К сожалению, запускать таким способом полноэкранные программы вроде редакторов текстов не получится, причина этого станет понятна чуть позже. Ещё нужно помнить, что процесс не может изменить текущую директорию другого процесса, так что команды вроде «`exec cd mydir`» или «`exec cd ..`» результата не дадут; впрочем, в Tcl есть своя команда `cd`, просто не надо её предварять словом `exec`.

Результатом выполнения команды можно воспользоваться в качестве аргумента для другой команды; для этого команду заключают в квадратные скобки. Например, если мы хотим, чтобы наш скрипт напечатал значение выражения, вычисленного с помощью `expr`, придётся написать что-то вроде

```
puts [expr 10+20]
```

Выполняя эту строку, интерпретатор сначала исполнит команду `expr 10+20`, получит результат `30`, подставит его вместо всего выражения `[expr 10+20]`, получит команду `puts 30` и выполнит её уже в таком виде. Стоит обратить внимание, что печать строки — это ***побочный эффект*** команды `puts`, а её *результат* — пустая строка.

Подстановка результата выполнения команды с использованием квадратных скобок работает в том числе и внутри двойных кавычек; это позволяет легко включать результаты команд в состав более сложных текстов.

Интересно, что для команды `exec` то, что выдала системная команда, становится *результатом* — иначе говоря, интерпретатор, выполняя `exec`, перехватывает поток стандартного вывода запускаемой внешней команды. Поэтому если мы напишем `exec ls` в скрипте, то ничего не увидим; чтобы скрипт *напечатал* то, что выдала команда `ls`, нужно написать, например,

```
puts [exec ls]
```

— но чаще всего так не делают; выдачу внешних команд в большинстве случаев интереснее проанализировать, а пользователю показать уже готовые результаты анализа. Как раз поэтому `exec` перехватывает вывод запускаемых внешних команд.

Кстати, теперь понятно, почему мы не можем запускать с помощью `exec` полноэкранные программы. Впрочем, `tclsh` позволяет запускать любые внешние программы без всякого `exec`: если имя введённой вами команды «верхнего уровня» (такой, которая набрана непосредственно в ответ на приглашение к вводу) не соответствует ни одной из команд, известных интерпретатору, он попытается рассмотреть полученную строку как имя и аргументы для запуска внешней программы. Изначально это задумывалось, чтобы `tclsh` можно было использовать в роли обычного командного интерпретатора, но, по правде говоря, идея подменить им тот же `bash` или другую оболочку выглядит несколько сомнительно.

12.2.2. Переменные и их значения

Как и во многих других языках командно-скриптового назначения, в Tcl единственным типом данных является строка текста — во всяком случае в той семантике Tcl, которая видна пользователю; интерпретатор в целях оптимизации может где-то «за кулисами» хранить, например, числовые значения, соответствующие некоторым строкам, чтобы не нужно было их снова и снова вычислять, но извне этого не видно. Строки могут обозначать сами себя, могут представлять числа, логические значения, они же служат в роли имён команд и переменных. Более того, и программа целиком, и любая законченная конструкция в её составе — это тоже строки.

Строки, применяемые для идентификации именованных сущностей — команд (встроенных и введённых пользователем), а равно и переменных — мы, как и в других языках, можем назвать ***идентификаторами***, хотя при описании Tcl этот термин не обязателен. Идентификатором может служить едва ли не любая строка, но лучше всё же сдерживать собственную фантазию и ограничиться, как в Си, идентификаторами, начинающимися с латинской буквы или знака подчёркивания и содержащими те же буквы, подчёркивание и цифры. Более изощрённые имена идентификаторов потребуют особого подхода, о котором, кстати, знают далеко не все программисты, пишущие на Tcl; читаемости вашей программе такое не добавит.

Значения присваиваются переменным с помощью команды `set`, имеющей ровно два аргумента — имя переменной и присваиваемое значение, например:

```
set x 25
set str "abra schwabra kadabra"
set n [expr n+1]
```

Обратиться к переменной, то есть узнать значение по её имени, можно с помощью знака `$`. Например, команда

```
puts $str
```

напечатает значение переменной `str` (в нашем примере — строку `abra schwabra kadabra`). Подстановка значений переменных работает в том числе внутри двойных кавычек, так что команда

```
puts "the value of x is $x"
```

напечатает `the value of x is 25`.

Команду `set` можно дать с одним аргументом — именем переменной; в этом случае её *значением* станет текущее значение указанной переменной, то есть это *тоже запись обращения к переменной*. Так, предыдущий пример команды можно переписать без знака `$`:

```
puts "the value of x is [set x]"
```

Встречается утверждение, что запись вида `$x` есть *сокращённая форма записи* `[set x]`.

Любопытно, что Tcl позволяет рассматривать строку, содержащую пробелы, как ***список***. Обращаться к отдельным элементам списка позволяет команда `lindex`, имеющая два аргумента: первый — собственно «список», второй — номер нужного элемента. В роли номера выступает целое число, причём первому слову строки соответствует `0`, второму — `1` и т. д. Например, результатом команды `lindex $str 1` в нашем примере будет слово `schwabra`. Элементом такого списка может, в свою очередь, быть список; для этого несколько слов группируются в одно с помощью кавычек (точно так же, как мы делали это в командной строке и скриптах на Bourne Shell). Чтобы «загнать» кавычки внутрь строки, их можно заэкранировать символом «\». Так, после

```
set str "abra \"schwab ra\" kadabra"
```

значением переменной `str` будет строка, содержащая кавычки: `abra "schwab ra" kadabra`, тогда как результатом выполнения команды `lindex $str 1` станет строка `schwab ra` (без кавычек).

Ещё одна интересная команда для работы с данными так и называется `string`; она предназначена для выполнения разнообразных операций над строками как таковыми — если угодно, над строками, которые здесь и сейчас рассматриваются как строки, а не что-то другое. Первый параметр команды `string` всегда представляет собой имя *субкоманды* и показывает, чего мы, собственно говоря, хотим. Например, субкоманда `index` позволяет извлечь из строки её символ по заданному номеру (позиции); так, *результатом* команды

```
string index abrakadabra 4
```

будет строка из одной буквы `k`, т. е., например,

```
set c [string index abrakadabra 4]
```

занесёт значение «`k`» в переменную `c`. Субкоманда `range` требует на один параметр больше и позволяет извлечь из заданной строки подстроку с такой-то по такую-то позицию (включительно); в частности, значением

```
string range abrakadabra 4 6
```

будет строка «`kad`». Для обеих субкоманд индексация, как видим, начинается с нуля; специальное значение `end` обозначает позицию последнего символа строки, значение `end-1` (без пробелов) — предпоследнего, `end-12` — двенадцатого с конца. Субкоманда `length` позволяет получить длину строки — например, результатом `string length abrakadabra` будет `11`. Среди прочих субкоманд можно найти `toupper` и `tolower` для приведения строки соответственно к верхнему регистру и к нижнему, `trim`, `trimleft` и `trimright` для удаления пробельных символов спереди и сзади, `replace` для замены части строки на другую строку, `equal` и `compare` для сравнения строк (с учётом или без учёта регистра, с возможностью сравнения только заданного количества первых символов и т. д.; сами эти субкоманды различаются возвращаемым значением); субкоманда `is` позволяет проверить принадлежность строки к одному из нескольких классов — например, является ли строка представлением целого числа, дробного числа, логического значения, состоит ли она только из заглавных букв или только из строчных, и т. п. Есть и другие субкоманды, про которые можно узнать из технической документации.

В дальнейших примерах мы будем активно использовать аргументы командной строки. Для доступа к ним используются три встроенные переменные: `argc`, `argv` и `argv0`. Пусть вас не обманывают названия этих переменных; работают они не совсем так, как аналогичные параметры в Си. Переменная `argc` содержит число (точнее, *строковое представление* числа, поскольку чисел как таковых в Tcl нет), равное количеству параметров командной строки, *не считая имени программы* (!); `argv` содержит *список* параметров командной строки — и тоже без имени программы. Если имя программы нам всё же потребуется, то, как легко догадаться, оно доступно через переменную `argv0`.

Например, аналог программы `echo` реализуется на Tcl в одну команду `puts $argv`. Следующий скрипт получает один аргумент командной строки, в котором должно быть арифметическое выражение, и вычисляет его:

```
#!/bin/sh
# calc0.tcl \
exec tclsh "$0" "$@"
puts [expr [lindex $argv 0]]
```

Почему так писать не надо, мы разберёмся в следующем параграфе.

12.2.3. Ветвления, циклы... и строки

Пример, которым мы завершили предыдущий параграф, обладает недостатком столь же очевидным, сколь и фатальным. Если пользователь, не зная, что делать с данной конкретной программой, попробует запустить её без параметров, он получит пачку диагностических сообщений от интерпретатора Tcl:

```
syntax error in expression "": premature end of expression
    while executing
"expr [lindex $argv 0]"
    invoked from within
"puts [expr [lindex $argv 0]]"
    (file "./calc0.tcl" line 4)
```

Нам, авторам скрипта, эта диагностика могла бы оказаться полезной, но конечному пользователю, который, скорее всего, никогда не слышал ни о каком языке Tcl и тем более о командах `lindex` и `expr`, подобная белиберда уж точно не понравится.

Здесь можно вернуться к предыдущей части книги и припомнить, что по итогам рассмотрения имеющихся реализаций Common Lisp мы в §11.1.15 сформулировали некий вывод или, лучше сказать, требование к транслятору, будь то интерпретатор или компилятор: поведение программы должна определять она сама, а не тот, кто служит посредником между программой и машиной. Из этого вытекает очевидное следствие: правильно написанная программа должна сама реагировать на ошибки пользователя, не доверяя и не делегируя это кому бы то ни было ещё. Если пользователь увидел диагностику интерпретатора и вообще любые сообщения, выданные кем-то, кроме самой программы, то в роли главного идиота выступает автор программы.

Естественно, реагировать на ошибки надо так, чтобы пользователю было понятно, в чём проблема и как её решить, и диагностика интерпретатора, предназначенная не пользователю, а программисту, здесь может только мешать. Как мы убедились в ходе борьбы с интерпретаторами Лиспа, Scheme и Пролога, там с возможностью написания программ, правильно обрабатывающих ошибки, дела обстоят из рук вон плохо — хуже всего ситуация в реализациях Common Lisp, чуть получше с этим в Chicken Scheme и SWI-Прологе, но всё ещё отвратительно. К счастью, автор Tcl думал прежде всего о практическом применении своего творения, а не об абстрактных высоких материях, так что здесь *есть* возможность написать программу правильно.

Чтобы устранить недостаток скрипта, нам потребуется конструкция ***ветвления***. В Tcl она сама по себе очень простая, но её описание потянет за собой длинное обсуждение, касающееся, как ни странно, *строк*, а ещё команд, стандартной библиотеки и вообще скриптинга как парадигмы. Прежде чем к нему приступить, мы просто покажем, как будет выглядеть исправленный скрипт:

```
#!/bin/sh
# calc.tcl \
exec tclsh "$0" "$@"

if { $argc == 1 } {
    puts [expr [lindex $argv 0]]
} else {
    puts stderr "Please specify an arithmetic expression"
    puts stderr "as the first parameter, such as"
    puts stderr "  $argv0 '2*2'"
    exit 1
}
```

На самом деле этот скрипт не сильно лучше предыдущего: если пользователь укажет параметр, но вместо выражения в этом параметре напишет, к примеру, слово «abrakadabra», всё снова кончится диагностикой интерпретатора. Справиться с этим несколько сложнее, но можно — вот так:

```
#!/bin/sh
# calc1.tcl \
exec tclsh "$0" "$@"
if { $argc == 1 } {
    set x [lindex $argv 0]
    set code [catch { expr $x } result]
    if { $code } {
```

```
        puts stderr "Couldn't evaluate ``$x''"
        exit 1
    } else {
        puts $result
    }
} else {
    puts stderr "Please specify an arithmetic expression"
    puts stderr "as the first parameter, such as"
    puts stderr "  $argv0 '2*2'"
    exit 1
}
```

Слово `catch` нам хорошо знакомо; действительно, Tcl предусматривает обработку исключений, в том числе для ошибок, которые констатировал сам. Эту возможность мы подробно обсудим позже.

Итак, что в наших примерах нового? Слово `stderr` мы в Tcl ещё не видели, но что́ оно означает — прекрасно знаем из опыта работы на других языках. Операция сравнения — двойное равенство, как в Си — нам тоже знакома (отметим заодно, что одиночный знак равенства в Tcl вообще не используется). Остаётся оператор `if`, который тоже выглядит совсем как в Си, вот разве что условие почему-то взято в фигурные скобки, а не в круглые... и вот тут нам стоит остановиться и не спешить.

Автор этих строк вынужден признаться, что когда-то давно (если память не изменяет, в 2001 году), впервые столкнувшись с Tcl с целью быстро слепить несколько простеньких программ с GUI на Tcl/Tk, не дал себе труда разобраться в происходящем и почти год писал на Tcl, не ведая, что творит. *Не делайте так!* Во-первых, подобная практика отвратительна сама по себе. Во-вторых, в нашем разговоре о парадигмах самая интересная особенность языка Tcl — как раз та, о которой сейчас пойдёт речь.

Итак, прежде всего отметим, что фигурные скобки в Tcl — это не какие-то там «группирующие символы», «составные операторы» или что-то подобное; **нечто, заключённое в фигурные скобки — это просто строка**, то есть буквально *кусок текста*. Чтобы проиллюстрировать эту мысль, заметим, что присваивание переменной `str` в одном из недавно рассмотренных примеров мы могли бы записать так:

```
set str {abra schwabra kadabra}
```

От двойных кавычек фигурные скобки отличаются одним фундаментальным свойством: **внутри фигурных скобок все символы обозначают сами себя**; в частности, в фигурных скобках Tcl не подставляет значения переменных вместо токенов вроде `$var`, он так и оставляет, как написано — знак доллара и всё, что за ним следует. Даже перевод строки, встреченный в фигурных скобках, остаётся самим собой.

Никакие символы внутри фигурных скобок не имеют никакого особого значения; из этого правила есть исключение, которое не все осознают — интерпретатор внутри фигурных скобок особым образом обрабатывает символы фигурных скобок, делая возможной своего рода вложенность строк в строки. Подчеркнём, что встреченные открывающие и закрывающие фигурные скобки продолжают обозначать сами себя, они, как и все другие символы, ни на что не заменяются, Tcl принципиально ничего не меняет в таких строках; просто интерпретатор ведёт подсчёт открывающих и закрывающих фигурных скобок и считает, что строковый литерал закончился, лишь когда количество закрывающих скобок сравняется с количеством открывающих. Между прочим, фигурную скобку (да, внутри фигурных скобок!) можно заэкранировать символом «\», тогда интерпретатор исключит её из подсчёта (это может понадобиться, чтобы, например, вывести непарный символ фигурной скобки на экран), но даже при этом и «\», и скобка останутся самими собой.

Коль скоро мы всё равно уже обсуждаем строки, отметим, что внутри литералов в двойных кавычках интерпретатор обрабатывает привычные нам по языку Си обозначения спецсимволов, такие как `\n`, `\r`, `\t` и некоторые другие; внутри фигурных скобок такие двухсимвольные комбинации останутся ровно тем, чем они и были — двухсимвольными комбинациями, что не исключает их интерпретации в дальнейшем, если содержимое фигурных скобок будет исполнено в качестве фрагмента программы.

Вернёмся теперь к «оператору» `if` и осознаем, что никакой он не оператор, **в Tcl вообще нет операторов**, во всяком случае, в том смысле, в котором они есть в Паскале или Си, а само слово `if` — никоим образом не «ключевое». В действительности `if` — это *просто имя встроенной команды*, такой же, как знакомые нам `expr`, `puts`, `exec` и прочие. Интерпретатор выполняет все команды, включая и `if`, по одним и тем же правилам: где нужно, выполняются подстановки (например, значений переменных), после чего первое слово «логической строки» рассматривается как имя команды, последующие — как аргументы; дальнейшее сделает уже реализация команды, и базовый интерпретатор может не строить никаких предположений о её особенностях.

Может сложиться такое впечатление, что предыдущий абзац описывает частные детали реализации интерпретатора и, как следствие, не имеет отношения к делу, если только мы не собираемся вносить изменения в интерпретатор. В действительности всё сложнее. Интерпретаторы обычно можно *расширять*, добавляя новые возможности; вспомним теперь, что в Лиспе написать новую спецформу невозможно — для этого придётся переделать сам базовый интерпретатор. Так вот, в Tcl мы можем написать свою собственную управляющую конструкцию, причём даже на самом Tcl, если захотим — или на Си, если это покажется нам удобнее. Интерпретатор при этом трогать будет не нужно.

Больше того, в Tcl отсутствуют *ключевые слова* — то есть вообще нет такой сущности. Смысл любой команды мы можем, если захотим, изменить, и к управляющим конструкциям это тоже относится: например, если нам не нужна конструкция `switch`, мы можем описать свою собственную процедуру с таким именем; делать так не рекомендуется, но никаких препятствий нам интерпретатор чинить не станет. С `if`'ом тоже вполне можно это проделать, но здесь всё же есть одна сложность: в остальной программе нам придётся тогда обойтись без привычного ветвления, что вряд ли удобно. Впрочем, в Tcl имеется команда `rename`, с помощью которой можно *переименовать* любую из существующих команд, в том числе `if`; если так сделать, мы освободим имя `if` для какой-то своей процедуры, но сама команда ветвления останется нам доступна под другим именем. Об этой возможности полезно знать, чтобы понимать, как устроен Tcl, но прежде чем реально сделать что-то подобное, стоит хорошо подумать.

Итак, `if` — это команда, а всё, что за ней следует — её параметры, каждый из которых представляет собой строку. Первая строка — текст условия; в нашем случае, как и в большинстве обращений к `if`, строка заключена в фигурные скобки, но теперь это нас не обманет, строка и есть строка. Это условие команда вычисляет, считая его арифметическим выражением, точно так же, как это делает, например, команда `expr` — реализация интерпретатора Tcl предусматривает для этого специальный метод. Полученный результат используется для принятия решения, какую из веток выполнить.

Примечательно, что Tcl допускает несколько различных представлений для логических значений. Значением арифметического выражения, на верхнем уровне которого находится операция сравнения, может стать, как и в языке Си, `0` или `1`, и, как и в языке Си, в Tcl число `0` (как целое, так и дробное, вроде `0.0`) считается обозначением лжи, а любое другое — обозначением истины. Но при этом истину можно также обозначить строками `true`, `on` и `yes`, а ложь — строками `false`, `off` и `no`; все эти строки можно сокращать, так что за истину сойдут `y`, `ye`, `tr` и т. п., за ложь — `f`, `fal` или `of`; ещё более неожиданной оказывается нечувствительность Tcl к регистру букв в этих строках — всевозможные `TRUE`, `tRuE`, `YeS` и прочее в таком духе остаются валидными обозначениями логической истины, при том что в именах команд и переменных Tcl чувствителен к регистру, т. е., например, `If` — это не то же самое, что `if`. Следует обратить внимание, что *произвольные* строки в роли логических значений использовать нельзя — будет выдана ошибка.

Следующий параметр команды `if` — опять же строка (а что же ещё!), но она уже рассматривается не как выражение, а как ***скрипт***, то есть фрагмент текста на языке Tcl — собственно, именно текст на Tcl там и есть. В наших примерах эта строка содержит внутри себя переводы строк, т. е. в нашем тексте она записана на нескольких строках, но с этим будет всё в порядке, ведь пока строка, начавшаяся с открывающей фигурной скобки, не закончится соответствующей закрывающей скобкой, все символы означают сами себя, и перевода строки это тоже касается. Этот скрипт команда `if` выполнит в том и только в том случае, если ранее вычисленное условие оказалось истинно, то есть его значением стало число, отличное от нуля. Далее следует слово `else`, его команда `if` рассматривает как «ключевое», но это не ключевое слово языка Tcl (как мы уже говорили, таких тут вообще нет), это просто вот так написана реализация команды `if` — где-то в потрохах её реализации есть сравнение переданного параметра со строкой `"else"`. Ну а после `else` идёт ещё одна строка в фигурных скобках, и это тоже скрипт, но предназначенный, чтобы выполнить его, если условие ложно — то есть в результате его вычисления получился ноль. Для выполнения первого или второго скрипта команда `if` запускает интерпретатор Tcl.

Между прочим, слово `else` во всех наших примерах необязательное — если его убрать, работать всё будет точно так же. Мы могли бы ещё вставить `then` после условия, команда `if` это тоже позволяет. Чтобы окончательно

развеять впечатление, что местный `if` чем-то похож на оператор ветвления из Си, рассмотрим следующий пример:

```
set b off ; if $b "puts YES" "puts NO"
```

В таком виде сия конструкция напечатает `NO`, если же сменить `off` на `on` или какое-то другое обозначение истины — будет напечатано уже `YES`. От сходства с Си, как видим, не осталось и следа (разве что само слово `if`, но оно и в Лиспе используется — никто ведь не скажет, что Лисп похож на Си?).

По правде говоря, в этом примере мы смогли использовать кавычки вместо фигурных скобок лишь по причине крайней простоты обеих веток `if`'а. Будь тела чуть посложнее, мы бы устали экранировать каждый символ доллара и каждую кавычку, а читать получившийся шедевр было бы совершенно невозможно. Если бы там ещё оказались вложенные конструкции вроде того же `if` или каких-нибудь циклов, всё стало бы совсем плохо — экранирования пришлось бы делать «кратными» и внимательно считать, где какой уровень вложенности. Фигурные скобки за счёт своих особенностей все эти проблемы снимают.

Кстати, команда `if`, как и другие команды, имеет *результат* — для неё он равен результату последней команды из выполнявшейся ветки. Чтобы это обнаружить, попробуем сделать так:

```
set b 100
set tt [if $b "puts YES ; expr 10" "puts NO ; expr 20"]
puts "result is $tt"
```

Напечатано теперь будет *две* строки. Если в `b` ненулевое значение — это будут строки `YES` и `result is 10`, в противном случае — соответственно `NO` и `result is 20`. Вспомнив Лисп и его спецформу `if`, мы определённо обнаружим сходство — там тоже вычислялось значение; разница между Лиспом и Tcl оказывается во многом терминологической: там были *формы* и они *вычислялись*, тогда как здесь у нас *команды* и они *выполняются*. Конечно, содержательная разница тоже есть — мы к ней ещё вернёмся.

Аналогично в Tcl устроены и конструкции циклов — всё это команды, принимающие строки в качестве параметров. Самым тривиальным оказывается, как обычно, цикл `while` — у этой команды два параметра, первый задаёт выражение для условия, второй — скрипт в качестве тела цикла. Например, распечатать аргументы командной строки можно и вот так:

```
set n 0
while { $n < $argc } {
    puts "parameter $n is [lindex $argv $n]"
    set n [expr $n + 1]
}
```

Между прочим, на примере `while` можно наглядно продемонстрировать разницу между использованием кавычек и фигурных скобок, особенно в условных выражениях. Внутри фигурных скобок интерпретатор не подставляет вместо переменных их значения, поскольку `$` воспринимается просто как символ; как следствие, команда `while` получает первым параметром заданное условное выражение как оно есть и перед каждой итерацией *сама вычисляет* это выражение, в том числе подставляет значения переменных, а поскольку они меняются от итерации к итерации, выражение вычисляется каждый раз новое, что, собственно, и требуется. Если бы мы попытались использовать кавычки, интерпретатор подставил бы *числа* вместо `$n` и `$argc`, получилась бы строка вроде `0 < 3` (если предположить, что аргументов командной строки у нас три), и именно такую строку получила бы команда `while` — но ведь эта строка вообще не зависит ни от каких переменных, ноль всегда меньше, чем три, так что цикл бы получился *бесконечным*. Для `if` это не так заметно, поскольку там условие вычисляется только один раз, и не имеет значения, будет оно меняться или нет.

Цикл `for` в Tcl очень похож на привычную нам конструкцию из языка Си. Параметров у этой команды четыре — инициализация, условие, итерация и тело; вначале команда выполнит скрипт, переданный первым параметром, перед каждым прогоном тела она будет вычислять условие (второй параметр) и решать, работать или выйти, а после каждого выполнения тела (то есть четвёртого параметра) будет выполнять ещё и третий параметр. Например, приведённый выше цикл `while` мы могли бы записать и так:

```
for { set n 0 } { $n < $argc } { set n [expr $n + 1] } {
    puts "Parameter $n is [lindex $argv $n]"
}
```

Отметим, что вместо громоздкого `set n [expr $n + 1]` можно воспользоваться куда как более компактной командой `incr` (от слова *increment*), которая рассматривает свой единственный параметр как имя переменной и увеличивает значение этой переменной на единицу. Заголовок нашего цикла с использованием `incr` можно переписать так:

```
for { set n 0 } { $n < $argc } { incr n } {
```

Цикла с постусловием в Tcl не предусмотрено, зато есть довольно хитрая команда `foreach`, специально предназначенная для построения циклов по спискам. В простейшем варианте эта команда имеет три аргумента: имя переменной цикла, список значений и тело. Пройдёмся по командной строке ещё одним способом:

```
foreach a $argv {
    puts "We've got $a as a command line argument"
}
```

Простейшей формой возможности `foreach` не ограничиваются. Для начала расскажем, что эта команда позволяет проходить списки не по одному элементу, а по два, по три и т. д., используя при этом соответствующее количество переменных. В этом случае первым параметром указывают не имя переменной, а список из нескольких имён. Проще всего показать это на примере:

```
# greek.tcl
set greek {
   alpha beta gamma delta epsilon zeta eta
   theta iota kappa lambda mu nu xi omicron
   pi rho sigma tau upsilon phi chi psi omega
}
foreach { x y z } $greek {
   puts "$x $y $z"
}
```

На первой итерации цикла переменные `x`, `y` и `z` получат соответственно значения `alpha`, `beta` и `gamma`, на второй итерации — значения `delta`, `epsilon` и `zeta`, и так далее, пока в списке `$greek` не кончатся элементы. В итоге скрипт распечатает список английских названий букв греческого алфавита по три в строке:

```
alpha beta gamma
delta epsilon zeta
eta theta iota
kappa lambda mu
nu xi omicron
pi rho sigma
tau upsilon phi
chi psi omega
```

Но и это ещё не всё. Команда `foreach` позволяет пройти несколько независимых списков параллельно, например:

```
set digits {0 1 2 3 4 5 6 7 8 9}
set dignames {
    zero one two three four
    five six seven eight nine
}
foreach d $digits n $dignames {
    puts "$d reads ``$n''"
}
```

Формального ограничения на количество таких списков в одном цикле нет, но, пожалуй, три списка и три переменные представить себе ещё можно, а вот четыре уже сделают всю конструкцию чрезмерно громоздкой. Оба подхода можно скомбинировать, например, проходя два списка с шагом по три элемента, или один список проходить по одному элементу за итерацию, а другой (одновременно) — по два элемента (или по три, или по сколько хотите). Здесь автор вынужден признать, что его фантазия иссякла и придумать осмысленный пример такого `foreach` он не смог.

Для досрочного завершения циклов предусмотрены команды `break` и `continue`; как и в языке Си, первая из них производит немедленный выход из ближайшего объемлющего цикла, а вторая досрочно завершает один проход тела цикла.

Отметим, что в качестве своего *результата* команды `while`, `for` и `foreach` всегда возвращают пустую строку, так же как, например, `puts`; попросту говоря, эти команды не предназначены для использования в составе выражений, их следует использовать только ради побочных эффектов.

Аналогом оператора выбора в Tcl служит команда `switch`; оставим её заинтересованным читателям для самостоятельного освоения.

Следует обратить внимание на один момент, вытекающий из отношения интерпретатора Tcl к строкам: **язык Tcl допускает только «честную» интерпретацию строкового представления программ и их частей**, никакие промежуточные представления тут не годятся, ведь даже тела циклов и ветвлений *чисто семантически* должны быть проинтерпретированы в тот момент, когда принято решение об их исполнении. До этого момента у интерпретатора попросту недостаточно информации, чтобы узнать окончательный вид кода.

12.2.4. Процедуры и видимость переменных

Подпрограммы в Tcl называются ***процедурами***, хотя на самом деле они, как и встроенные возможности, похожи скорее то ли на команды, то ли на функции. Собственно говоря, в Tcl нет никаких иных «действующих лиц», нежели эти (уже знакомые нам) команды-функции. Для описания процедуры — то есть фактически команды Tcl, написанной на самом Tcl — тоже применяется *команда* (с именем `proc`), эффект которой — добавить в интерпретатор новую команду с заданным именем, заданным списком параметров и заданным телом. Разумеется, и имя, и список параметров, и тело — это просто строки, ведь в Tcl нет никаких других данных, кроме строк. Итак, команда `proc` принимает три аргумента: имя, параметры и тело. Если параметры не требуются, вторым аргументом указывают пустую строку.

Для примера опишем команду `hello`, которая будет печатать строку «Hello, world»:

```
proc hello {} {
    puts "Hello, world"
}
```

Чтобы у читателя не возникало неправильных ощущений, подчеркнём, что то же самое можно написать иначе, например, так:

```
proc "hello" "" "puts \"Hello, world\""
```

Здесь истинная сущность слова `proc` видна более наглядно.

Список параметров (второй аргумент команды `proc`) может быть непустым — он задаёт одновременно и количество параметров, и то, как они будут именоваться в теле новой процедуры. Например:

```
proc greet { name } {
    puts "Hello, dear $name, I'm pleased to meet you!"
}
```

Здесь создаётся команда `greet`, имеющая один аргумент. С интерпретатором, в котором введена такая процедура (то есть исполнена показанная выше команда `proc`), может состояться следующий диалог:

```
% greet Vasya
Hello, dear Vasya, I'm pleased to meet you!
% greet {Amvrosiy Ambruazovich}
Hello, dear Amvrosiy Ambruazovich, I'm pleased to meet you!
% greet "Mister X"
Hello, dear Mister X, I'm pleased to meet you!
```

Как и «встроенные» (то есть просто написанные на Си) команды Tcl, процедуры, введённые с помощью `proc`, могут возвращать значения. Сделать это можно двумя способами — явным и неявным. Явно указать возвращаемое значение позволяет команда `return`, предполагающая один аргумент (собственно возвращаемое значение); как и, например, в языке Си, команда `return` заодно прекращает выполнение тела процедуры, так что её можно использовать для досрочного выхода. Так, следующая процедура *печатает* строку `Hello`, при этом *возвращает в качестве значения* строку `Ok`:

```
proc hello_ok {} {
    puts "Hello"
    return "Ok"
}
```

Если же процедура не указывает своего возвращаемого значения явно, то есть не выполняет команду `return`, то значением, возвращённым из процедуры, становится значение, возвращённое *последней командой, выполненной в её теле*. Например, следующая процедура возводит заданное число в квадрат:

```
proc quad { x } { expr $x * $x }
```

Того же эффекта можно достичь и с помощью `return`:

```
proc quad { x } {
    return [expr $x * $x]
}
```

Tcl позволяет создавать вариадические процедуры, т. е. процедуры с переменным числом параметров. Сделано это с помощью «волшебного» имени параметра — `args`. Параметр с таким именем должен быть или единственным (в этом случае его обычно для наглядности не заключают в скобки), или последним в списке параметров. Значением переменной `args` в теле процедуры становится *список* «неиспользованных» фактических параметров.

Например, следующая процедура считает сумму своих параметров, сколько бы их ни было (если не указать ни одного, она вернёт `0`):

```
proc sumargs args {
    set s 0
    foreach x $args {
        set s [expr $s + $x]
    }
    return $s
}
```

Следующий пример — поиск максимума среди заданных чисел — чуть более сложен: для пустого множества максимум не определён, так что хотя бы один параметр должен быть задан обязательно, и именно он выступает в роли начального значения максимума:

```
proc maximum { x args } {
    set max $x
    foreach t $args {
        if { $t > $max} {
            set max $t
        }
    }
    return $max
}
```

Отдельного обсуждения заслуживает видимость переменных; авторы Tcl подошли к этому вопросу несколько неожиданным способом. **В Tcl любая переменная, упоминаемая внутри процедуры, по умолчанию считается локальной** — если программист явным образом не укажет иное. Так, попытка обратиться из процедуры к глобальной переменной без предварительно принятых специальных мер приведёт к ошибке:

```
% set x "abra kadabra"
abra kadabra
% proc p1 {} { puts $x }
% p1
can't read "x": no such variable
%
```

Здесь мы создали в глобальной области видимости переменную `x`, после чего наглядно убедились, что внутри процедуры `p1` эта переменная недоступна. Точно так же попытка присвоить внутри процедуры переменную, имя которой совпадает с именем глобальной переменной, никак на эту глобальную переменную не повлияет:

```
% set x "abra kadabra"
abra kadabra
% proc p2 {} { set x "foo bar" }
% p2
foo bar
% puts $x
abra kadabra
%
```

Чтобы в процедуре работать с глобальной переменной, нужно это намерение задекларировать в явном виде, применив команду `global`; параметрами этой команде служат имена переменных. С момента выполнения команды `global` (обязательно в теле процедуры) в текущем контексте вместе с локальными переменными появляются *синонимы* указанных глобальных переменных под теми же именами. Опишем для примера процедуру без параметров, которая вычисляет сумму глобальных переменных `x` и `y`:

```
proc x_plus_y {} {
    global x y
    expr $x + $y
}
```

После выполнения команды `global` *локальные* имена `x` и `y` становятся синонимами глобальных переменных с теми же именами, так что `expr` обратится уже к глобальным переменным:

```
% set x 25
25
% set y 36
36
% x_plus_y
61
%
```

Используя эту возможность, сохраняйте благоразумие: **помните, глобальные переменные — абсолютное вселенское зло!** Процедуру `x_plus_y` мы привели лишь для иллюстрации.

Теоретически параметр команды `global` может быть, как и любое другое значение, вычислен во время работы. Например, процедура

```
proc print_and_clear { name } {
    global $name
    eval " puts \$$name "
    set $name ""
}
```

принимает на вход *имя глобальной переменной*, печатает её значение, а самой переменной присваивает пустую строку:

```
% set x "abra kadabra"
abra kadabra
% print_and_clear x
abra kadabra
% puts ($x)
()
%
```

По правде говоря, так никогда не делают, и тому есть целых две причины. Во-первых, обратите внимание, к каким ухищрениям нам пришлось прибегнуть, чтобы обратиться к глобальной переменной, имя которой находится в другой (локальной) переменной; разгадать ребус с командой `eval`, кавычками (а не фигурными скобками, здесь это важно) и экранированным знаком доллара предоставим читателю самостоятельно. Во-вторых, польза от такой процедуры довольно сомнительна, ведь она может «печатать и очищать» *только глобальные переменные* — с локальной переменной другой процедуры у неё ничего не получится.

Если возникает необходимость «передать в процедуру переменную» (в отличие от её значения), это тоже вполне можно устроить. Tcl предоставляет для этого способ штатный и, в отличие от вышеприведённого трюка, корректный. Команда `upvar` позволяет ввести в контексте процедуры *синонимы для переменных из объемлющего контекста*, то есть, попросту говоря, из того контекста, откуда процедура была вызвана. Эта команда принимает чётное число аргументов — два или более; аргументы попарно задают имена переменных из внешнего контекста и соответствующие имена локальных синонимов. Например,

```
upvar x out_x y out_y z out_z
```

сделает переменные `x`, `y` и `z` из объемлющего контекста доступными в текущей процедуре под именами `out_x`, `out_y` и `out_z`. Обычно эта возможность используется, чтобы организовать *передачу параметра по имени* — аналог var-параметра Паскаля, передачи адреса из Си или передачи по ссылке из Си++. Для примера перепишем нашу команду `print_and_clear`:

```
proc print_and_clear { name } {
    upvar $name var
    puts $var
    set var ""
}
```

Здесь мы предполагаем, что через параметр `name` нам передали имя переменной, которая существует в контексте вызывающего; для работы с этой переменной мы вводим локальный синоним `var` и дальше пользуемся этим синонимом как обычной переменной. В отличие от предыдущей версии, этот вариант процедуры `print_and_clear` будет прекрасно работать с локальными переменными в других процедурах, да и реализация стала проще, понятнее и вообще приятнее: всё-таки конструкция с `eval`, кавычками и долларом представляла собой неочевидный и некрасивый хак, а здесь он нам уже не требуется.

Приведём ещё один пример — процедуру, меняющую местами значения двух переменных:

```
proc swap { a b } {
    upvar $a p $b q
    set t $p
    set p $q
    set q $t
    return {}
}
```

Проверяем:

```
% set x 25 ; set y 36
36
% puts "$x $y"
25 36
% swap x y
% puts "$x $y"
36 25
%
```

Вообще-то `upvar` позволяет обратиться не только к переменным из контекста вызывающего, но и *из любого другого*, вплоть до глобального. Для этого применяется ещё один параметр, указываемый первым; этот параметр должен быть либо натуральным числом 1, 2 и т. д., которое означает, *на сколько фреймов* следует подняться по стеку, либо *абсолютным номером стекового фрейма* — #0 (глобальная область видимости), #1 (контекст процедуры, вызванной на верхнем уровне) и т. д. В реальной жизни изредка используется вариант #0 для создания синонимов глобальных переменных под другими именами; значение 1 (естественно, без знака «#») соответствует контексту вызывающего, его нет надобности указывать явно — именно оно используется по умолчанию. Про использование значений, отличных от #0 и 1, в одном из описаний, попавшихся автору, было сказано лаконичное «you're probably asking for trouble»[2].

Коль скоро нам пришлось обсудить команду `upvar`, упомянем заодно родственную ей команду `uplevel`, которая позволяет в контексте одного из объемлющих фреймов *выполнить целый скрипт*. Первый её аргумент устроен точно так же, как первый аргумент `upvar`, и точно так же настоятельно рекомендуется воздержаться от использования значений, отличных от #0 и 1; по умолчанию используется как раз 1, но опустить первый аргумент можно лишь в случае, если следующий аргумент не начинается ни с цифры, ни со знака #. Все остальные аргументы, сколько бы их ни было, объединяются в один (как списки) и выполняются в качестве скрипта в контексте заданного фрейма. Эта команда чрезвычайно полезна при создании новых управляющих структур — например, если вам придёт в голову всё-таки ввести в Tcl цикл с постусловием; больше она практически ни для чего не нужна.

Возможно, читатель уже догадался, что переменная в Tcl *создаётся* при первом присваивании; эта операция обратима — переменную можно ликвидировать с помощью команды `unset`. Tcl позволяет узнать, существует ли в текущем контексте переменная с заданным именем — для этого используется команда `info` с подкомандой `exists`; например, `info exists x` возвращает 1, если переменная `x` в текущем контексте есть, и 0 в противном случае. Команда `info` вообще позволяет многое узнать о текущем состоянии интерпретатора; подробности читатель найдёт в технической документации.

[2] «Вы, вероятно, напрашиваетесь на неприятности» (англ.).

12.2.5. Обработка особых ситуаций

Выше мы в одном из примеров вынуждены были прибегнуть к обработке ошибки с помощью команды `catch`; разберёмся с ней подробнее. Как мы знаем, команда в Tcl, завершив выполнение, возвращает некое значение. Кроме того, некоторые команды оказывают влияние на выполнение того фрагмента программы, в котором они были вызваны: это команды `return`, `break` и `continue`. Про `return` нам известно, что она завершает выполнение процедуры; добавим, что она также способна завершить команду верхнего уровня (например, если вне процедур дать команду цикла, а в его теле встретится `return`) и выполнение внешнего скрипта, вызванного командой `source`. Команда `continue` досрочно завершает выполнение тела цикла (одной итерации), команда `break` — выполнение цикла целиком.

Добавим к этому перечню команду `error`, которая обычно вызывается с одним аргументом — сообщением об ошибке; в этом случае интерпретатор «разматывает» стек вызванных и пока не завершившихся команд, пока либо не встретится `catch`, либо стек не опустеет; в последнем случае интерпретатор сам выдаст сообщение об ошибке — ровно то, которое указано параметром команды `error`, и плюс к этому информацию о «размотанном» стеке, то есть какие команды откуда были вызваны на момент выполнения `error`.

Для понимания работы команды `catch` нам придётся разобраться, *как именно* (каким образом) некоторые команды ухитряются влиять на то, что происходит вокруг них, изменяя (на самом деле — просто прерывая) последовательность исполнения команд на одном или нескольких внешних уровнях вложенности. Впрочем, всё оказывается гораздо проще, чем могло бы быть: **кроме своего результата, любая команда в Tcl генерирует ещё и *код завершения***. Такой код представляет собой целое число; для пяти чисел, используемых базовыми средствами Tcl, предусмотрены названия: `ok` (0), `error` (1), `return` (2), `break` (3) и `continue` (4). Большинство команд использует код `0`, означающий нормальное завершение с возвратом результата. Интересно, что Tcl поддерживает произвольные числа в качестве таких кодов; это сделано, чтобы дать возможность пользователю вводить свои собственные управляющие структуры, использующие альтернативные модели передачи управления.

Когда интерпретатор выполняет одну за другой команды, составляющие скрипт (либо весь целиком, либо тело какой-нибудь управляющей конструкции), он после завершения каждой отдельной команды проверяет код, с которым эта команда завершилась. Как можно заметить, только код `0` допускает выполнение следующей команды из скрипта; любой ненулевой код заставляет интерпретатор немедленно прекратить выполнение на текущем уровне.

Дальнейшее зависит от того, как устроены объемлющие контексты выполнения. Интерпретатор в любом случае будет подниматься по стеку этих контекстов, пока не найдёт такой, в котором имеется обработчик соответствующего кода завершения. В частности, *все циклы* обрамляют выполнение своего тела обработкой значений `3` и `4`, благодаря чему в них работают `break` и `continue`. Процедуры, введённые командой `proc`, обвешивают своё тело обработчиком, перехватывающим код `2`, и то же самое делает сам интерпретатор для команд верхнего уровня, а команда `source` — для текста внешнего скрипта, благодаря чему в них во всех правильно срабатывает команда `return`. Код `1` не обрабатывает никто, так что если мы сами не вставим, где нужно, команду `catch`, стек будет размотан до конца и мы увидим диагностику, выданную интерпретатором. Что-то похожее произойдёт, если применить `break` или `continue` вне цикла, хотя «размотки до конца» здесь не будет: коды `3` и `4` «отловит» любой из обработчиков, сделанный для `return` (вокруг тела функции, команды верхнего уровня или в команде `script`), а поскольку такая ситуация ошибочна, дальнейшая размотка стека пойдёт уже с кодом `1` (т. е. обработчик выполнит команду `error` или сделает аналогичное ей действие).

Команда `return` позволяет завершить текущую процедуру (или объект, приравненный к ней) не только с кодом `0`, но и с любым другим. Для этого используется опция `-code`, параметром которой может служить как число (`0`, `1` и т. д. — вообще говоря, произвольное целое неотрицательное), так и условные обозначения — слова `ok`, `error`, `return`, `break` и `continue` (соответственно для значений 0, 1, 2, 3 и 4). Теперь мы знаем, как написать свои собственные команды вместо `break` и `continue`:

```
proc my_break {} { return -code 3 }
proc my_cont {} { return -code 4 }
```

или так:

```
proc my_break {} { return -code break }
proc my_cont {} { return -code continue }
```

Надо сказать, что команда `return` написана так, чтобы понимать строковые обозначения кодов, но больше нигде их использовать нельзя, во всех остальных местах программы в любом случае придётся пользоваться числами.

Работать `my_break` и `my_cont` будут абсолютно так же, как и их встроенные прототипы. Намного интереснее ситуация с кодом `2` (`return`): этот код позволяет написать *процедуру, которую можно использовать вместо `return`*. Например:

```
proc retABC {} { return -code 2 ABC }
```

Если теперь написать процедуру и дать в ней эту команду, то такая процедура вернёт строку `ABC`. Так, процедура `p`:

```
proc p { x } {
    if { $x == 1 } {
        retABC
    } else {
        return {}
    }
}
```

будет возвращать строку `ABC`, если её аргумент — число `1`, и пустую строку — для любых других чисел.

Вернёмся теперь к тому, ради чего мы затеяли всё это обсуждение — к команде `catch`. Она принимает один или два аргумента[3]; первый из них — собственно скрипт, который может кончиться «как-то не так», второй (необязательный) — имя переменной для занесения результата. В большинстве случаев второй аргумент обязательно нужен, ведь иначе результат выполнения скрипта (если, к примеру, он выполнится успешно) мы уже никаким способом не узнаем.

Пусть, к примеру, у нас есть процедура `suspicious`, которую нам надо вызвать с параметром `42` и которая при этом может «плохо кончиться», то есть где-то в ходе её выполнения может возникнуть ошибка, которую мы хотели бы перехватить и обработать. Соответствующая команда `catch` будет выглядеть так:

```
catch { suspicious 42 } result
```

Здесь `result` — имя переменной, в которую следует занести результат выполнения `{ suspicious 42 }`. Сама команда вернёт *код*, с которым завершится скрипт, то есть, скорее всего, число `0` (всё в порядке) или `1` (возникла ошибка). Значение, возвращённое командой `catch`, нам, очевидно, важно, так что вызывать команду `catch` нужно как-нибудь так:

```
set code [catch { suspicious 42 } result]
```

После выполнения такой строчки в переменной `code` может быть значение `0` — в этом случае вызов процедуры `suspicious` прошёл успешно, результат её выполнения находится в переменной `result`; либо `code` может содержать `1` — тогда значением `result` будет текст сообщения об ошибке.

Остальных значений здесь можно не ожидать, хотя, конечно, если задаться целью такое устроить, то всё возможно: для любых значений, кроме 3 и 4, достаточно вызвать из `suspicious` (прямо или косвенно) какую-нибудь процедуру, которая выполнит `return -code` с соответствующим кодом; для 3 и 4 такой `return` придётся расположить прямо в теле `suspicious`, иначе его отловят раньше и превратят в код `1`.

Дальше в нашей программе, по-видимому, должен стоять какой-нибудь `if` вроде такого:

```
if { $code == 1 } {
    puts stderr "Error: <<$result>>"
} else {
    puts "Success; the result is $result"
}
```

Можно, конечно, обойтись без `set` и переменной `code`, запихнув `catch` прямо в условие `if`'а; в принципе, это дело вкуса, но яснее программа от этого не станет.

[3] В версии Tcl 8.5 был добавлен ещё и третий аргумент для `catch`, но мы условились обсуждать «классическую» версию 8.4; к тому же подробное описание этого третьего аргумента заняло бы слишком много места, а объяснение, как им пользоваться — ещё больше, и, что самое обидное, всё равно эти знания не имеют шансов вам пригодиться.

12.2.6. Файлы, потоки и внешние команды

Набор команд Tcl, отвечающий за работу с файлами или, точнее говоря, с *потоками ввода-вывода*, на первый взгляд напоминает интерфейс, привычный нам по языку Си; на второй взгляд этот набор команд предстаёт неким гибридом низкоуровневого и высокоуровневого ввода-вывода (см. т. 2, гл. 4.4 и §5.2.3), но и это лишь начало истории. Команда `open`, предназначенная для открытия файла, умеет заодно ещё и запускать внешние команды, перехватывая их стандартный ввод или вывод, но и это не слишком удивительно для тех, кто знаком с функциями `popen` и `pclose` из библиотеки Си. Настоящие приключения начнутся, если вам придёт в голову обрабатывать на Tcl не текстовые, а бинарные данные. Как мы помним, в языке Tcl есть только строки, никаких других типов данных нет; работать с бинарными данными при этом возможно, просто их приходится представлять в виде строк. Но — обо всём по порядку.

Для открытия файла служит команда `open`, принимающая от одного до трёх параметров. С её первым параметром всё вроде бы просто — это имя файла; но к нему мы ещё вернёмся. Второй параметр, если он есть, задаёт режим открытия файла, причём этот режим можно задать по аналогии со вторым параметром функции `fopen` (см. §4.4.3) — строка `"r"` означает чтение, `"r+"` — чтение и запись существующего файла, `"w"` — только запись, `"w+"` — чтение и запись, но в отличие от `"r+"` в этом случае файл создаётся, а если он уже есть, его содержимое сбрасывается; `"a"` (от слова *append*) открывает файл на добавление в конец. Как и во втором параметре `fopen`, здесь можно использовать букву `b` для указания, что поток будет обрабатываться как бинарный, но, как читатель, возможно, помнит, функция `fopen` эту букву игнорирует; в Tcl же бинарные потоки ввода-вывода от текстовых отличаются, и весьма существенно.

Есть и другой способ задания второго параметра `open`: это может быть *список*, составленный из слов `RDONLY`, `WRONLY`, `RDWR`, `APPEND`, `BINARY`, `CREAT`, `EXCL`, `NOCTTY`, `NONBLOCK` и `TRUNC`, имеющих тот же смысл, что и соответствующие константы, используемые с *системным вызовом* `open` — `O_RDONLY`, `O_TRUNC` и т. д. (см. §5.2.3), только здесь они записываются без префикса `O_`. Например, строка `{WRONLY CREAT TRUNC}` задаёт комбинацию флагов, хорошо известную нам ещё со времён изучения программирования на ассемблере — открыть файл на запись, создать, если его нет, а если есть — сбросить его содержимое и начать его записывать «с чистого листа».

Смысл третьего параметра в точности совпадает со смыслом третьего параметра системного вызова `open` — это права доступа к файлу, которые следует установить, если файл создаётся; более того, и задаётся он точно так же — числом, обычно восьмеричным, которое, как мы помним, в большинстве случаев равно `0666`, намного реже — `0600`, а другие значения вообще почти никогда не используются. Если второй параметр не подразумевает создания файла, то третий можно (и нужно) опустить; более того, значение `0666` команда `open` использует по умолчанию, то есть и в этом случае явно указывать третий параметр не нужно. Можно с хорошей вероятностью предсказать, что вам этот параметр вообще никогда не потребуется.

Отметим, что и второй параметр можно опустить — по умолчанию файл открывается в режиме «только чтение». Таким образом, команда

```
open "file.txt"
```

откроет файл `file.txt` из текущей директории на чтение, а две команды, имеющие абсолютно одинаковый смысл:

```
open "file.txt" "w"
open "file.txt" {WRONLY CREAT TRUNC}
```

откроют тот же файл на запись, при необходимости создав новый файл или опустошив существующий. Как читатель уже, можно надеяться, знает из своего опыта, эти два варианта покрывают подавляющее большинство случаев, возникающих на практике — если не считать того, что мы нигде не указали букву `b`, то есть работаем с текстовыми файлами. Впрочем, если вы пишете скрипт на Tcl и вам вдруг потребовался бинарный ввод-вывод — подумайте, правильно ли вы выбрали язык программирования.

Команда `open` возвращает в качестве своего результата Tcl'евский аналог файлового дескриптора, который, естественно, тоже представляет собой строку. Это могут быть, например, строки вроде `file5`, `file12` и т. п., но нас это, по идее, волновать не должно: полученную строку нужно сохранить в переменной, чтобы потом использовать для операций ввода-вывода и для закрытия файла, а что́ это конкретно за строка — какая нам разница. Вызов команды `open` может выглядеть как-то так:

```
set fd [open "file.txt"]
```

Отметим, что для обозначения стандартных потоков ввода, вывода и диагностики можно использовать соответственно слова `stdin`, `stdout` и `stderr`.

Этот вариант оформления `open` не учитывает возможности возникновения ошибки. Если же ошибка возникнет при выполнении скрипта в квадратных скобках (в данном случае это как раз команда `open`), то до выполнения объемлющей команды (в данном случае `set`) дело не дойдёт. Правильнее будет поступить примерно так:

```
if { [catch { set fd [open "file.txt"] } openError] } {
    puts stderr "$openError"
    return false;
}
```

Здесь подразумевается, что дело происходит внутри какой-нибудь процедуры, откуда можно выйти, вернув значение «ложь» как индикацию происшедшей ошибки; ваша программа может быть организована совершенно иначе. В любом случае строка диагностики окажется в переменной `openError`, и этим можно будет воспользоваться. Вернёмся теперь к первому параметру команды `open`, который задаёт, как мы уже сказали, имя файла. Если значение этого параметра будет начинаться с символа `|`, подразумевается, что вместо операции открытия файла нужно произвести запуск внешней программы с перехватом её потока вывода (если второй параметр `open` подразумевает чтение) или ввода (если поток открывается на запись). Например,

```
open "|/bin/ls"
```

запустит программу `/bin/ls`, перехватив её поток вывода, и вернёт нам дескриптор потока, из которого весь этот вывод можно будет прочитать; команда

```
open "|grep -v abc" "w"
```

запустит программу `grep` с параметрами `-v` и `abc`, перехватив её поток ввода, и создаст для нас поток вывода, через который мы сможем подать запущенной программе данные на вход.

Как ни странно, в отличие от `popen`, Tcl поддерживает запуск внешней программы с перехватом обоих её стандартных потоков — именно это произойдёт, если первый аргумент `open` начинается с `|`, а в качестве второго указано `r+` или `w+`. В большинстве случаев от такого использования никакого проку не будет; свой вывод, поступающий на вход внешней программе, вы ещё можете заставить уйти в канал, невзирая на буферизацию — с помощью команды `flush`; но программа, которую вы запустили в роли внешней, ничего не знает о необходимости очищать свой буфер вывода, так что вы из полученного канала ничего не прочитаете, пока внешняя программа либо не завершится, либо не переполнит свой буфер вывода. Начинающие часто пытаются организовать с внешней программой подобие диалога — «строчку записать, строчку прочитать» — так, как обычно с программами работает пользователь в терминале. Разумеется, из этого ничего не выйдет, ведь вытеснение буферов при работе с каналами устроено не так, как при работе с терминалом.

Возможно, вы от внешней программы чего-то добьётесь, если запишете в поток связи с ней всё, что вы хотели ей подать на вход, после чего поток «закроете наполовину». К этому вопросу мы вернёмся при рассмотрении команды `close`. Другой подход состоит в применении неблокирующего ввода-вывода. Подумайте хорошенько, надо ли оно вам.

Для чтения из потока ввода используются команды `read` и `gets`, для записи данных в поток вывода — команда `puts`; поскольку она нам уже знакома, с неё мы и начнём. Ранее мы использовали эту команду всегда с одним параметром; в этом случае она выдаёт свой аргумент в стандартный поток вывода, добавив после него ещё перевод строки. Добавление перевода строки можно отключить, указав флажок `-nonewline`, например:

```
puts -nonewline "Your name, please: "
```

Чтобы заставить `puts` работать с потоком, отличным от стандартного, нужно добавить ещё один аргумент — собственно Tcl'евский дескриптор потока, полученный от `open` (или как-то ещё). Например, если дескриптор содержится в переменной `fd`, можно сделать так:

```
puts $fd "This string goes to file"
```

или так:

```
puts -nonewline $fd "This string goes to file with no EOL"
```

Отметим довольно неожиданный момент: никаких других средств файлового вывода в Tcl не предусмотрено.

В отличие от `puts`, которая по умолчанию работает со стандартным потоком, обе команды чтения — как `read`, так и `gets` — требуют обязательного указания дескриптора потока; если нужен стандартный ввод, следует использовать слово `stdin`.

Команда `gets` читает из заданного потока ввода символы, пока не встретит перевод строки, и полученную строку (уже без символа перевода строки) возвращает. Команда может использоваться с одним параметром или с двумя. В первом случае прочитанная строка возвращается в качестве результата команды. Во втором случае второй параметр задаёт имя переменной, в которую и записывается прочитанная строка, сама же команда возвращает *количество прочитанных символов* (не включая символ перевода строки) или `-1`, если прочитать ничего не удалось (например, если возникла ситуация «конец файла»). Если не использовать переменную, то «конец файла» приводит ровно к тому же, что и прочтение пустой строки — команда возвращает пустую строку; это вполне может оказаться нормальным, если совместно с `gets` использовать команду `eof`, речь о которой пойдёт ниже.

Команда `read` первым аргументом принимает дескриптор потока; можно также задать второй аргумент — число, указывающее, сколько символов следует прочитать. Если этот аргумент не задан, команда прочитает всё содержимое файла до конца; в частности, если читать из стандартного потока ввода, команда `read stdin` не вернёт управление, пока не настанет ситуация «конец файла» (в применении к чтению с терминала — пока пользователь не нажмёт `Ctrl-D`). Пробуя эту возможность в интерактивном режиме интерпретатора, учтите, что `read` и сам интерпретатор используют один и тот же поток со всей его обвеской, так что «конец файла», который вы устроите для `read`, заодно может заставить завершиться и ваш интерпретатор, хотя это происходит не во всех версиях.

Поведение `read` при заданном количестве символов может оказаться неожиданным для людей, привыкших к одноимённому системному вызову: команда *блокируется, пока не прочитает сколько сказано* или пока не настанет «конец файла». Заставить её вести себя так же, как системный вызов — отдавать всё, что можно отдать прямо сейчас, если есть хотя бы один байт — можно, переведя поток в неблокирующий режим (задав флажок `NONBLOCK` при открытии файла или воспользовавшись командой `fconfigure`, которую мы вскоре рассмотрим).

В отличие от `gets`, команда `read` всегда возвращает прочитанное в виде своего результата, записывать данные в переменные она не умеет.

Узнать, не возникла ли ситуация конца файла, позволяет команда `eof`, принимающая на вход дескриптор потока и возвращающая `0` или `1`. Интересно, что эта команда вообще ничего не делает с потоком самим по себе — она проверяет, *возник ли «конец файла» при выполнении на данном потоке последней операции ввода*. Иначе говоря, **команду `eof` бессмысленно использовать перед чтением из потока** — нужно использовать её только после попытки чтения.

Аналогично — проверяя результаты последней операции — работает ещё одна команда, `fblocked`, применять которую имеет смысл только на потоках, работающих в неблокирующем режиме. Она позволяет узнать, не вернула ли последняя команда чтения меньше данных, чем от неё ожидалось, из-за того, что нужного количества данных не было в наличии.

Команда `close` позволяет закрыть поток ввода-вывода, и с этой командой связан целый ряд сюрпризов. Прежде всего, **если закрываемый поток ввода-вывода связан с внешней командой и не является неблокирующим, `close` будет ждать завершения выполнения этой внешней команды**. Такое свойство `close` может показаться довольно неприятным, но дело-то даже не в ней: что действительно неприятно, так это то, что мы никак не можем воздействовать на запущенную с помощью `open` внешнюю программу — Tcl не предоставляет возможности ни узнать её pid, ни убить её, и на это обстоятельство люди склонны не обращать внимания, пока не столкнутся с поведением `close`. Подчеркнём ещё раз, что ругать саму команду `close` тут бессмысленно, её поведение лишь высвечивает фундаментальный недочёт всего подхода.

Со вторым сюрпризом можно столкнуться при закрытии неблокирующего потока вывода. Когда команда `close` закрывает поток вывода, она должна вытеснить всё содержимое буфера. Имея дело с блокирующим потоком, она просто производит вывод содержимого буфера, не возвращая управление, пока вывод не закончится и буфер не опустеет; но с *неблокирующим* потоком так поступить нельзя. Поэтому `close` возвращает управление немедленно. В документации написано, что буфер будет вытеснен (и поток после этого окончательно закрыт) «в фоновом режиме», но в действительности это означает, что мы в своей программе должны время от времени запускать *событийный цикл*, реализованный командой `vwait`. Основное его предназначение — работа с сокетами; к счастью, нас не удивить циклом обработки событий — читатель, можно надеяться, уже представляет, что конкретно может скрываться в потрохах этой `vwait`. Иной вопрос, что задействование таких возможностей представляется чрезмерным и вполне может рассматриваться как достаточный повод для отказа от использования неблокирующего вывода.

Третий сюрприз состоит в том, что у команды обнаруживается дополнительный аргумент, позволяющий «закрыть поток наполовину»; в качестве этого аргумента можно передать слова `read` и `write`, или просто буквы `r` или `w`. Для сокетов в этом случае выполняется системный вызов `shutdown`, и это ещё вполне понятно; но, оказывается, есть второй вид потоков, поддерживающих «закрытие наполовину» — это каналы, связанные с запущенной внешней программой в режиме чтения и записи.

Чтобы оценить масштабы бедствия, расскажем по секрету, что этот вариант потока реализуется *двумя* каналами (`pipe`, см. т. 2, 5.3.15) — один на чтение, другой на запись, ну а «закрытие наполовину» закрывает только один из двух дескрипторов. Надо сказать, что создатели Tcl приложили здесь довольно серьёзные усилия — слишком серьёзные, если учесть, что, как отмечалось выше, добиться какого-то внятного толка от запущенной таким способом внешней программы удаётся очень редко.

Проведём краткий обзор оставшихся команд, связанных с файловым вводом-выводом. Подробности по этим командам мы приводить не будем; всю недостающую информацию можно, как обычно, почерпнуть из документации к ним.

Команда `fconfigure` позволяет настраивать существующий поток — переводить его в неблокирующий режим и обратно, устанавливать кодировку данных (да, Tcl умеет переводить текст из одной кодировки в другую; лучше бы он этого не умел), включать (и, что обычно полезнее, *выключать*) преобразование представления конца строки из односимвольного (`LF`) в двухсимвольный (`CR LF`) и обратно, а также управлять буферизацией.

Как уже упоминалось выше, Tcl в принципе позволяет вводить и выводить бинарные данные (в противоположность текстовым). Массивы бинарных данных приходится представлять в виде строк; работать с ними позволяет команда `binary`: подкоманда `binary format` позволяет *синтезировать* представление бинарных данных (для последующей записи в файл или поток), а подкоманда `binary scan` — анализировать такое представление (как правило, прочитанное из файла или потока). Команда `binary` предусматривает ещё две полезные подкоманды — `encode` и `decode`, которые, к счастью, не имеют никакого отношения к текстовым кодировкам: они позволяют перевести бинарные данные в форматы base64, uuencode и hex (простое шестнадцатеричное представление каждого байта) и обратно.

Команда `socket` позволяет работать с TCP-сокетами, причём как с клиентскими, так и с серверными. Если с клиентскими сокетами всё и так более-менее ясно, то серверные сокеты, как мы знаем, порождают *проблему очерёдности действий* (см. т. 2, §6.4.1). Tcl эту проблему решает довольно своеобразно: при создании серверного сокета нужно указать команду (обычно это имя процедуры), которая будет выполняться всякий раз при принятии очередного соединения, причём одним из её аргументов будет дескриптор канала, связанного с новым клиентом. Эта процедура может связать с клиентским дескриптором скрипты, подлежащие выполнению, когда канал оказывается готов к чтению или к записи; это делается с помощью команды `fileevent`. Из таких скриптов выстраивается знакомая нам система обработчиков событий на сокетах, остаётся лишь организовать ***главный цикл***; для этого применяется команда `vwait`, которая запускает цикл обработки событий и выполняет его до тех пор, пока какой-нибудь из обработчиков не изменит глобальную переменную, указанную команде в качестве параметра. Фактически Tcl предлагает нам написать событийно-ориентированный TCP-сервер, при этом его главный цикл оказывается спрятан в команду, но нам всё равно необходимо понимать, *что именно* там, в этой команде, находится.

Последняя команда, которую хотелось бы упомянуть, называется просто `file`. Она позволяет выполнять разнообразные действия над файлом как единым объектом — копировать, переименовывать, удалять файлы, узнавать их тип, размер и всевозможные атрибуты вроде прав доступа и дат создания/модификации, а также и изменять их, создавать и удалять директории, создавать символические и жёсткие ссылки, и много чего ещё.

12.2.7. Ассоциативные массивы

В роли массива в обычном смысле, когда элементы доступны по номерам, можно использовать обычный список, обращаясь к его элементам через команду `lindex` (см. стр. 181); вопреки возможным ожиданиям, `lindex` работает достаточно эффективно: для строки, которая хотя бы раз рассматривалась как список, интерпретатор хранит вспомогательные (невидимые) структуры данных, позволяющие производить индексирование за константное время.

На семантике программы эти невидимые данные никак не сказываются, единственный способ догадаться об их существовании — померить быстродействие на сравнительно больших списках, содержащих элементы существенно разного размера, и убедиться, что «лобовое» решение таких результатов дать не может.

Ассоциативные массивы, которые мы сейчас рассмотрим, предназначены для другой цели; в сущности это вообще не массивы, а скорее хеш-таблицы[4], поскольку в роли «индекса» при доступе к их элементам могут выступать произвольные строки. Использовать их как традиционные массивы, где в роли индексов выступают натуральные числа и элементы расположены достаточно плотно (т. е. массив не является разреженным), нет никакого смысла: ассоциативные массивы работают *медленнее*, чем списки и команда `lindex`.

На самом деле при введении ассоциативных массивов речь идёт об *особой форме имён переменных*. Такое имя состоит из двух частей: имени массива и ключа, при этом ключ заключается в круглые скобки. Примерами таких имён переменных служат, например, `a(1)`, `abra(kadabra)`, `foo(bar,bur)` и т. п.; обращаться с ними вполне можно как с обычными переменными — так, команда

```
set foo(bar,bur) "abrakadabra"
```

присвоит значение `abrakadabra` элементу массива `foo` с индексом `bar,bur`, а обратиться к этой переменной (получить её значение) можно, как всегда, с помощью знака `$`: `$foo(bar,bur)`. Подчеркнём, что индекс здесь по-прежнему один, а запятая — просто один из символов строки, используемой в качестве индекса.

Основная прелесть таких «сложносочинённых» имён переменных состоит в возможности вычислить индексную часть имени, применяя выражения вроде `$foo($a,$b)` или `$abra([f x])`. На первый взгляд, что-то подобное можно сделать и с обычными переменными; например, следующие команды:

```
set i 15
set m$i 1000
```

присвоят значение `1000` переменной `m15`. Проблемы здесь начнутся при попытке *обратиться* к переменной таким способом: `$m15` интерпретатор воспримет ровно так, как мы могли бы ожидать, а вот в выражении `$m$i` мы уже никак не сможем объяснить интерпретатору, что значение `$i` следует рассматривать как часть имени переменной для предыдущего знака `$`. Конечно, если всерьёз упереться, можно придумать для этого выражение вроде

```
eval return [string cat {$m} "$i"]
```

— но писать такое каждый раз при обращении к массиву нам быстро надоест. Есть более экономный способ: ввести процедуру

```
proc getv { name } { upvar $name n ; return $n }
```

и обращаться к переменным с хитрыми именами через неё:

```
set t [getv m$i]
```

Это, во-первых, всё ещё достаточно громоздко; во-вторых, работать это будет ещё медленнее, ведь вызов процедуры, тело которой представлено в виде текста (напомним, семантика Tcl требует честной интерпретации) — штука уж точно не дешёвая. Но есть и более серьёзный аргумент в пользу ассоциативных массивов: интерпретатор знает, что́ это такое, и включает для них специальную поддержку.

Во-первых, массивы поддерживаются командой `upvar`: с её точки зрения имя переменной в применении к массиву — это имя массива (без скобок и ключа). Для иллюстрации введём следующую процедуру:

```
proc arr { a i } {
    upvar $a an
    return $an($i)
}
```

Как видим, эта процедура получает два параметра — имя массива и значение индекса — и возвращает значение заданного таким образом элемента, например:

```
set abra(kadabra) schwabra
set x [arr abra kadabra]     ;# x получает значение schwabra
```

Вернувшись к телу процедуры `arr`, обратите внимание на аргументы команды `upvar`: локальное имя `an` становится синонимом *имени массива*, переданного через параметр `a`, и само, естественно, используется как массив.

Кроме того, в Tcl имеется команда `array` для работы с массивами. Например, команда `array names abra` вернёт список всех строк, используемых в массиве `abra` в роли индексов; команда `array get` возвращает все элементы массива в виде списка пар имя/значение, `array set` принимает такой список как аргумент и присваивает указанные значения элементам с соответствующими индексами. Команда `array` также позволяет производить в массиве поиск и выполнять много других полезных операций.

Подчеркнём один важный факт, который следует понять относительно только что введённых возможностей: **ассоциативные массивы в Tcl не являются не только особым типом данных, но и вообще данными как таковыми: массив не может рассматриваться как единый объект.** В частности, массив как целое нельзя передать в процедуру или вернуть из неё (хотя можно передать *имя* как обычную строку, а затем воспользоваться `upvar`, как в примере выше), массив (как единый объект) не может быть значением переменной. Можно считать, что массив представляет собой просто набор отдельных переменных, не объединённых ничем, кроме первой части своего имени.

В более поздних версиях Tcl были введены так называемые *словари* (*dictionaries*), интерфейсом к которым служит команда `dict`. В отличие от ассоциативных массивов такой «словарь» представляет собой единый объект данных; как и для всех данных в Tcl, для словарей единственное *видимое* представление — строка текста, хотя, конечно, «за кулисами» для его хранения используются намного более эффективные структуры данных. Подробно рассматривать эту возможность мы не будем.

12.2.8. Если сравнить Tcl с Лиспом

На первый взгляд Tcl и Лисп совершенно не похожи друг на друга; если смотреть внимательнее, то они, наоборот, начинают казаться *очень* похожими: единый носитель представления программ и данных, примитив `EVAL`, интерпретируемая сущность семантики — все эти признаки объединяют скриптовые языки с языками лиспоподобными. Даже вызов подпрограммы здесь делается немножко похоже — берётся список, в котором первый элемент идентифицирует вызываемого, а остальные служат параметрами, и вычисляется; разве что скобки при этом в Лиспе круглые, а в Tcl — квадратные.

Поскольку наш основной предмет обсуждения — парадигмы, стоит обратить внимание на одно ключевое различие между вычислительными моделями Лиспа и Tcl, на которое, как ни странно, мало кто обращает внимание, а зря: как мы увидим ниже, именно это отличие оказывается очень важным. Итак, **в Лиспе всё по умолчанию вычисляется, так что приходится явным образом блокировать вычисления, когда требуется просто выражение как таковое; в Tcl, напротив, ничего не вычисляется, пока вычисление не будет явно затребовано**. Для такого явного требования в Tcl служат *операция* обращения к переменной (знак `$`) и *подстановка результата выполнения команды* — квадратные скобки.

Может показаться, что это не так уж и важно — тут явно обозначаем одно, там другое, какая разница. А разница, между прочим, начинается с того, что **в Tcl не нужны никакие аналоги лисповских спецформ и макро-**

[4] Краткий рассказ о хеш-таблицах был приведён в первом томе, §2.10.8.

сов — как мы видели, все управляющие конструкции вроде `if`, `while` и т. п. представляют собой простые команды, выполняющиеся точно так же, как любые другие, и в программе можно описать свои собственные аналоги для управляющих команд, чего никак нельзя было сделать в Лиспе; на стр. 113 мы отметили, что набор ***спецформ*** для каждого диалекта Лиспа закрытый, то есть расширить его программист не может, а на стр. 120 подробно объяснили почему. Если Лисп часто называют «языком без синтаксиса», но при этом всё-таки вспоминают, что вместо синтаксиса там есть спецформы, решающие фактически те же проблемы, то Tcl в этом плане оказывается концептуально чище — в нём *действительно* нет синтаксиса.

Второе принципиальное семантическое отличие Tcl от Лиспа (на самом деле тесно связанное с первым) состоит в подходе к обработке кода как данных. В Лиспе поддерживаются *функции* как первоклассные объекты — а точнее, не сами функции, а ***замыкания***, включающие в себя, кроме тела функции, ещё значения свободных переменных (или, точнее говоря, *связи* переменных со значениями, при том что сами эти значения могут изменяться). В Tcl, как мы видели, свободных переменных просто нет (см. стр. 184). Отметим, что процедуры — местный аналог функций — здесь не являются «первоклассными объектами данных», процедура тут есть не более чем связь её имени со списком формальных параметров и телом — то есть, попросту говоря, связь одной строки с двумя другими. Впрочем, жить это совершенно не мешает, ведь «первоклассным» (и вообще единственным) объектом тут является строка, в том числе и строка кода, которую можно передавать кому и когда угодно через параметры и возвраты из процедур — и в любой момент можно «вычислить» (выполнить) как код. Аналог замыкания здесь достигается *подстановкой* значений переменных в строку кода перед её передачей получателю.

12.3. Интерпретатор Tcl и язык Си

Автор Tcl изначально позиционировал этот язык как предназначенный для *встраивания* в программы на Си и, возможно, других языках; больше того, при ответе на вопрос, что такое Tcl, Джон Оустерхаут обычно говорил, что это *библиотека для Си*, включающая в себя поддержку языка для скриптинга (см. статью [27]). Практически все функции интерпретатора Tcl действительно оформлены в виде библиотеки, `tclsh` их просто использует. Мы сами тоже можем так сделать, и, как мы вскоре увидим, это не так уж сложно — важно лишь не увлекаться и помнить, *зачем* мы это делаем.

Надо сказать, что задачи и предметные области, действительно требующие встраиваемого интерпретатора — не просто так, а по делу — возникают довольно редко. Автору этих строк такая задача встретилась всего один раз за всю его практику. Несомненно, стоит иметь в виду саму по себе возможность встроить в программу интерпретатор, чтобы, если нужда вас всё-таки заставит это сделать, вы знали, что́ вам нужно и куда за этим обратиться. Чего точно делать не следует — так это искать такие задачи специально. Известно, что, когда в руках молоток, все окружающие предметы начинают казаться гвоздями[5]; встраивание интерпретаторов относится к «молоткам» такого рода, которые лучше засунуть в самый дальний угол шкафа с инструментами и не доставать, пока практика вас к этому не вынудит. Пока без этого *можно* обойтись, без этого *нужно* обходиться.

Tcl реализован так, что его можно не только *встроить* в программу на Си, но и *расширить* командами, написанными в виде функций Си. С расширением интерпретатора ситуация не столь жёсткая, как с его встраиванием, нужно только помнить, что Tcl — язык скриптовой и не предназначен для написания программ в десятки тысяч строк (да и в одну-две тысячи, откровенно говоря, тоже). Но если вставшая перед вами задача требует от силы двух-трёх сотен строк кода, не налагает жёстких требований по эффективности и вы готовы терпеть существование интерпретатора на машине конечного пользователя, то почему бы и не сэкономить немного трудозатрат.

Представьте себе, что вы написали программу, предназначенную для работы на сервере и, естественно, не имеющую ничего даже отдалённо похожего на пользовательский интерфейс. При этом программе требуются некие конфигурационные или другие файлы, которые (по смыслу стоящей задачи) должен создать (сгенерировать) пользователь, поставивший вашу программу на свой сервер. Задача генерации таких файлов может оказаться многократно, в сотни раз проще, чем задача, которую решает ваша серверная программа; это тот самый случай, когда вполне оправданно графический пользовательский интерфейс сделать на Tcl/Tk.

Допустим, при этом у вас уже есть библиотека функций (написанных на Си или Си++), позволяющих работать со злосчастными файлами; возможно, часть этих функций или даже все они используются в вашей серверной программе. Переписывать эти функции на Tcl может оказаться занятием долгим и неблагодарным, особенно если обрабатываемые файлы имеют бинарный формат. Будет намного проще, написав небольшой «переходник», начать использовать ваши (уже имеющиеся) функции прямо из скриптов, написанных на Tcl. Разумеется, бывают и другие случаи; генерация настроечных файлов для серверной программы здесь приведена лишь в качестве простого примера.

В этой главе мы попытаемся создать общее представление о том, как именно всё это делается. Конечно, мы не сможем привести подробного описания всей библиотеки Tcl, включающей несколько сотен функций, но это и не нужно: технические детали вам не потребуются, пока дело не дойдёт до их практического использования, ну а если до этого всё же дойдёт, то источники, позволяющие освоить техническую конкретику, имеются в изобилии.

Чтобы попробовать примеры из этой главы, вам потребуется установить в системе откомпилированные файлы библиотеки `libtcl` и заголовочные файлы для неё. В подавляющем большинстве дистрибутивов Linux есть соответствующий пакет; в частности, в дистрибутивах семейства Debian (в том числе и Devuan, и Ubuntu) этот пакет называется `tcl-dev`.

12.3.1. Встраиваемый Tcl

Для демонстрации мы напишем программу, которая сама работает как интерпретатор Tcl — конечно, очень упрощённый. Общая идея реализации такого интерпретатора следующая. В программе нужно подключить заголовочный файл `tcl/tcl.h` (возможно, в вашей системе можно использовать просто `tcl.h`, но это не везде так). Далее нужно создать *объект интерпретатора Tcl*; это делается с помощью функции `Tcl_CreateInterp`, которая возвращает значение типа `Tcl_Interp*`; объект интерпретатора хранит имена и тела команд, значения глобальных переменных, стек вычислительных контекстов и всю прочую информацию, которая нужна для выполнения программ на Tcl.

Далее можно будет читать с клавиатуры команды и выполнять их с помощью функции `Tcl_Eval`; сама эта функция возвращает знакомый нам *код* завершения команды, причём заголовочник `tcl.h` предоставляет нам символические имена (макросы) для пяти возможных значений этого кода: `TCL_OK` (`0`), `TCL_ERROR` (`1`), `TCL_RETURN` (`2`), `TCL_BREAK` (`3`) и `TCL_CONTINUE` (`4`). Результат выполненной команды — строку, которую она вернула (при завершении с кодом `TCL_OK` или `TCL_RETURN`), или сообщение об ошибке (для кода `TCL_ERROR`) можно получить с помощью функции `Tcl_GetStringResult`.

При чтении команд с клавиатуры недостаточно просто прочитать одну строку, отдать её интерпретатору и так до бесконечности. Дело в том, что команда может, как мы неоднократно видели, располагаться на нескольких строках: для этого достаточно открыть и не закрыть фигурную скобку. Для корректной обработки этого нужно продолжать читать и «склеивать» строки, пока не получится команда целиком — такая, в которой всё открытое закрылось. К счастью, для этого не нужно писать свой лексический анализатор, поскольку, естественно, такая возможность есть в интерпретаторе Tcl и, как и любая другая его возможность, она доступна нам через функцию; в данном случае это функция `Tcl_CommandComplete`, которая принимает на вход строку и выдаёт число `1` (истина), если строка представляет одну или несколько законченных команд Tcl, а в противном случае возвращает `0`.

С чтения команды мы и начнём. Наша функция будет принимать на вход указатель на буфер (массив типа `char*`), в котором нужно расположить прочитанную команду, и длину этого буфера; возвращать наша функция будет `1`, если команда успешно считана, и `0`, если прочитать команду по той или иной причине не удалось. В случае, если вводимая пользователем команда в буфер не поместится, функция выдаст диагностическое сообщение, дочитает поток стандартного ввода до перевода строки (или конца файла) и вернёт `0`. Обнаружив конец файла, функция вернёт `0` сразу же и без всякой диагностики.

Чтение строки мы будем выполнять с помощью `fgets`, при этом воспользуемся тем, что сразу после прочитанной информации она в переданный ей буфер заносит нулевой байт, а все оставшиеся элементы массива не трогает; как следствие, достаточно перед обращением к `fgets` занести ноль в предпоследний байт буфера, и если после чтения строки он там останется — значит,

[5] Это т. н. *закон инструмента*, он же *принцип золотого молотка*; его приписывают Абрахаму Маслоу, который также известен своей *пирамидой потребностей*.

строка прочитана целиком, то есть памяти хватило. Формально говоря, если там не ноль, а символ перевода строки — это тоже должно значить, что памяти хватило (как говорится, тик в тик), но мы проверять этот вариант не будем; из-за этого допустимая длина команды у нас будет на один байт меньше, но это вряд ли очень страшно. Полностью функция чтения одной команды будет выглядеть так:

```
static int read_command(char *cmd_buf, int bufsize)
{
    int len = 0;
    *cmd_buf = 0;
    do {
        const char *r;
        if(len > bufsize-3) {  /* nowhere to read "}\n" to */
            fprintf(stderr, "Command too long\n");
            return 0;
        }
        cmd_buf[bufsize-2] = 0;
        r = fgets(cmd_buf+len, bufsize-len, stdin);
        if(!r)    /* EOF */
            return 0;
        if(cmd_buf[bufsize-2]) {
            int c;
            fprintf(stderr, "Command too long\n");
            while((c = getchar()) != EOF && c != '\n')
                {}
            return 0;
        }
        len += strlen(cmd_buf+len);
    } while(!Tcl_CommandComplete(cmd_buf));
    return 1;
}
```

Следующей мы напишем функцию, реализующую знакомый нам цикл чтения-выполнения-печати (*read-eval-print loop*, ***REPL***). Этой функции уже потребуется объект интерпретатора; кроме того, мы передадим ей строку, которую она будет выдавать как приглашение к вводу. Длину массива для хранения команды мы зададим глобальной константой:

```
enum { maxcmd = 32 * 1024 };
```

Наша функция будет обращаться к уже написанной функции `read_command`, и если та вернула истину, то выполнять прочитанную команду через интерпретатор Tcl, анализировать результат и выдавать возвращённое значение, если оно есть, после чего начинать всё сначала — и так пока `read_command` не вернёт ложь.

Для выполнения команды мы воспользуемся функцией `Tcl_Eval`, которая принимает всего два аргумента: указатель на объект интерпретатора и собственно строку, которую надо исполнить; возвращает она, как уже говорилось, код завершения команды. Прежде чем обратиться к этой функции, на всякий случай попросим интерпретатор очистить хранилище результата — это делается с помощью функции `Tcl_ResetResult`. Когда функция `Tcl_Eval` закончит выполнение команды и вернёт управление, нужно будет проверить, с каким кодом она завершилась, и действовать соответственно. По коду `TCL_OK` нужно проверить, не пуст ли результат, и если не пуст, напечатать его (если результат — пустая строка, мы не будем её печатать, чтобы на экране не появлялось лишних переводов строки). По коду `TCL_ERROR` нужно напечатать сообщение об ошибке, которое находится там же, где мог бы быть результат. Если же нам вернули какой-то ещё код, остаётся только очень сильно удивиться. Текст функции, которую мы назовём `rep_loop`, получается таким:

```
static void rep_loop(Tcl_Interp *interp, const char *prompt)
{
    char *buf;
    buf = malloc(maxcmd);
    fputs(prompt, stdout);
    while(read_command(buf, maxcmd)) {
        int code;
        const char *resptr;
        Tcl_ResetResult(interp);
        code = Tcl_Eval(interp, buf);
        switch(code) {
        case TCL_OK:
            resptr = Tcl_GetStringResult(interp);
            if(*resptr)
                printf("%s\n", resptr);
            break;
        case TCL_ERROR:
            resptr = Tcl_GetStringResult(interp);
            fprintf(stderr, "TCL ERROR: %s\n", resptr);
            break;
        case TCL_RETURN:
            fprintf(stderr, "return?! what the hell?\n");
            break;
        case TCL_BREAK:
        case TCL_CONTINUE:
            fprintf(stderr,
                    "strange to use break/continue this way\n");
            break;
        default:
            fprintf(stderr, "unknown Tcl ret.code %d\n", code);
        }
        fputs(prompt, stdout);
    }
}
```

Осталось только добавить функцию `main`. Всё, что она должна сделать — это создать интерпретатор и вызвать только что написанную `rep_loop`. Приглашение к вводу мы специально сделаем не таким, как в `tclsh` — воспользуемся строкой `"> "`:

```
int main()
{
    Tcl_Interp *interp = Tcl_CreateInterp();
    rep_loop(interp, "> ");
    return 0;
}
```

Откомпилировать и скомпоновать полученную программу можно командой вроде

```
gcc -Wall -g tclembed.c -ltcl -o tclembed
```

(обратите внимание на флаг `-ltcl`, который заставит редактор связей подключить библиотеку `libtcl`). Всё, запускаем её и убеждаемся, что можно вводить команды Tcl: они тут же будут выполняться.

Пример получился, откровенно говоря, не очень интересный, но это только начало. Давайте посмотрим, как добавить в интерпретатор свою собственную команду. Именно такие команды дают скрипту, написанному на встроенном варианте Tcl, доступ к управлению вашей программой, то есть только благодаря им достигается единственная осмысленная цель встраивания интерпретаторов в программы. Поскольку наша программа сугубо демонстрационная, команду для Tcl мы придумаем тоже иллюстративную: она будет принимать ровно один аргумент (как всегда, строку) и возвращать «утроенную» строку — например, получив `abc`, она вернёт `abcabcabc`. Мы назовём её `triple`.

Добавление в интерпретатор Tcl команды, реализованной на Си, выглядит довольно просто. Для начала надо написать функцию, которая реализует эту команду. Функция должна возвращать `int` — знакомый нам *код* (обычно либо `TCL_OK`, либо `TCL_ERROR`), а на вход принимать четыре параметра. Первый параметр, имеющий тип `void*` — так называемые пользовательские данные; мы этот параметр никак не задействуем, но вообще он позволяет реализовать много разных команд одной функцией. Второй параметр — указатель на интерпретатор; он нужен, чтобы вернуть результат, а ещё он позволяет самой команде выполнять какие-то ещё команды Tcl (вспомним, что `if` — это тоже команда). Последние два параметра называются `argc` и `argv`; разумеется, эти слова нам хорошо знакомы. Параметр `argc` показывает, сколько аргументов в интерпретаторе получила наша команда, а массив указателей на строки `argv` содержит адреса аргументов в памяти (причём `argv[0]` — это имя команды, аналогия с функцией `main` тут полная). Заголовок нашей функции, которую мы назовём `proc_triple`, получается такой:

```
static int proc_triple(ClientData cd, Tcl_Interp *interp,
                       int argc, const char **argv)
```

Чтобы написать её тело, нам потребуется знать, что возврат результата команды (в отличие от кода, который функция просто возвращает как своё значение) производится через объект интерпретатора; для этого проще всего воспользоваться функцией `Tcl_SetResult`. Параметров эта функция принимает три: первый — указатель на объект интерпретатора, второй — собственно строка результата, а третий задаёт то, как интерпретатор должен/может этой строкой распорядиться. Вариантов тут четыре. Если в качестве результата мы возвращаем строковую константу или какую-то ещё строку, которая — это мы точно знаем — не изменится, то интерпретатору совершенно ни к чему создавать копию этой строки, о чём мы ему и сообщаем, передав значение `TCL_STATIC`. Второй вариант — прямо противоположный: мы требуем, чтобы интерпретатор создал для себя копию той строки, которую мы ему передаём, поскольку сама она нам нужна для каких-то других целей. Это обозначается константой `TCL_VOLATILE`. Третий вариант — `TCL_DYNAMIC`; он означает, что интерпретатору предлагается стать владельцем передаваемой ему строки и в нужный момент ликвидировать её (освободить память). Делать это он будет с помощью функции `Tcl_Free`, и это значит, что выделить такую память нам нужно с помощью `Tcl_Alloc`; именно этим способом мы и воспользуемся. В принципе, есть ещё четвёртый вариант: в качестве этого параметра можно передать указатель на функцию, с помощью которой интерпретатор должен удалить нашу строку, когда она ему больше не будет нужна. Этот вариант может пригодиться, если в программе используется какой-то свой менеджер динамической памяти; но это не наш случай. Итак, пишем:

```
static int proc_triple(ClientData cd, Tcl_Interp *interp,
                       int argc, const char **argv)
{
    int len;
    char *res;
    if(argc != 2) {
        Tcl_SetResult(interp,
                      "must give exactly 1 parameter",
                      TCL_STATIC);
        return TCL_ERROR;
    }
    len = strlen(argv[1]);
    res = Tcl_Alloc(len*3+1);
    strcpy(res, argv[1]);
    strcpy(res+len, argv[1]);
    strcpy(res+2*len, argv[1]);
    Tcl_SetResult(interp, res, TCL_DYNAMIC);
    return TCL_OK;
}
```

Теперь осталось только зарегистрировать новую функцию в объекте интерпретатора; это делается с помощью `Tcl_CreateCommand`, которая принимает пять параметров: интерпретатор, имя команды (строку), адрес функции, которая реализует команду, и ещё два, которые в нашем случае будут нулевыми указателями: первый — те самые клиентские данные, которые нужно передать в нашу функцию `proc_triple`, а второй — указатель на функцию, которую нужно вызвать, если наша новая команда будет удалена из интерпретатора (это позволяет сделать для команды что-то вроде деструктора). Мы вызовем `Tcl_CreateCommand` из функции `main`, которая теперь станет такой:

```
int main()
{
    Tcl_Interp *interp = Tcl_CreateInterp();
    Tcl_CreateCommand(interp, "triple", proc_triple, NULL, NULL);
    rep_loop(interp, "> ");
    return 0;
}
```

Откомпилируем новую версию программы и опробуем её:

```
avst@host:~$ gcc -Wall -g tclembed.c -ltcl -o tclembed
avst@host:~$ ./tclembed
> triple abrakadabra
abrakadabraabrakadabraabrakadabra
>
```

Попробуем «проявить» сообщения из функции `rep_loop`, которые мы ещё не видели. С ошибками проблем не будет, достаточно дать какую-нибудь несуществующую команду. Попытаемся теперь сделать на верхнем уровне `break`, `continue` или `return -code 2` (напомним, это код `return` — не того, который мы выполнили, а того, который должен произойти уровнем выше). Результаты могут нас разочаровать: соответствующих сообщений, которые мы так старательно писали, мы не увидим, `Tcl_Eval` «заботливо» превратит их все в обычные ошибки.

Эта особенность `Tcl_Eval` известна и документирована: сама `Tcl_Eval` подсчитывает, на каком уровне она вызвана, и, зная, что сейчас она обслуживает верхний уровень вложенности (то есть её вызов не вложен ни в какой другой её вызов для того же самого интерпретатора), она действительно сама обрабатывает все коды, отличные от `0` и `1`, превращая их в код `1`. Но мы можем перехитрить её и всё-таки увидеть наши драгоценные сообщения. Для этого придётся добавить ещё одну команду в интерпретатор; эта команда будет запускать функцию `rep_loop`, используя заданный (параметром команды) вид приглашения, чтобы мы наглядно видели, что это *другой экземпляр* нашего REPL. Команду мы назовём `ENTER` (вот прямо так, большими буквами — обычно команды называют маленькими буквами, так что эта специфическая команда ни с чем не будет конфликтовать). Поскольку сама команда `ENTER` будет вызвана изнутри интерпретатора, все последующие экземпляры REPL будут *вложены* в тот вызов `Tcl_Eval`, который исполняет команду `ENTER`; в результате вызовы `Tcl_Eval`, исполняющие команды «внутри» `ENTER`, уже не будут вызовами верхнего уровня.

Функция для реализации `ENTER` будет довольно простой. Если аргументов не задано, в качестве нового приглашения она будет использовать строку `">> "`; если аргументов задано больше одного, лишние она просто проигнорирует, так что можно не делать обработку ошибок. Единственная хитрость состоит в том, что завершаться вложенный цикл выполнения команд будет по концу файла, и чтобы можно было выйти, нажав `Ctrl-D`, именно из одного уровня цикла, а не из всей программы, мы после завершения `rep_loop` будем сбрасывать флажок ситуации «конец файла», вызвав функцию `clearerr`. Всё вместе будет выглядеть так:

```
static int proc_enter(ClientData cd, Tcl_Interp *interp,
                      int argc, const char **argv)
{
    const char *prompt = argc > 1 ? argv[1] : ">> ";
    rep_loop(interp, prompt);
    clearerr(stdin);
    Tcl_SetResult(interp, "", TCL_STATIC);
    return TCL_OK;
}
```

Добавим в `main` регистрацию новой команды:

```
    Tcl_CreateCommand(interp, "ENTER", proc_enter, NULL, NULL);
```

— и дело сделано. Компилируем программу и пробуем:

```
> break
TCL ERROR: invoked "break" outside of a loop
> ENTER {::: }
::: puts { We're in! }
 We're in!
::: break
strange to use break/continue this way
::: return 1
return?! what the hell?
::: ^D
>
```

Нашему интерпретатору не хватает ещё одной возможности — при вызове его из командной строки сначала считывать указанные аргументами файлы (например, содержащие описания процедур на Tcl) а уже потом входить или не входить в REPL — в зависимости от указаний, полученных через те же аргументы. Без этого пользоваться полученным интерпретатором мы толком не сможем. Проблем реализовать это всё, в общем, нет, особенно если знать, что для исполнения целиком файла на Tcl есть специальная функция `Tcl_EvalFile` (параметры — указатель на объект интерпретатора и имя файла; остальное она сделает сама). Оставим это читателю в надежде, что примеров, которые мы уже успели привести, для понимания происходящего достаточно.

12.3.2. Расширение набора команд tclsh

Как уже говорилось, интерпретатор `tclsh` (а равно и интерпретатор `wish`, применяемый для Tcl/Tk, который мы рассмотрим в следующей главе) можно расширить командами, написанными на Си или Си++, причём, естественно, без перекомпиляции самих интерпретаторов. Общая идея тут в том, что нужный набор функций собирается в *динамическую библиотеку* (под *nix-системами файлы таких библиотек обычно имеют суффикс `.so` от слов *shared object*). Эту динамическую библиотеку затем можно подгрузить из работающего интерпретатора, и в нём станут доступны команды, реализованные в библиотеке. Такие загружаемые расширения для программ (в данном случае для интерпретатора) обычно называют ***плагинами*** (от английского *plug-in*).

Команды, которые вы хотите видеть в Tcl, оформляются точно так же, как мы это делали в предыдущем параграфе — в виде функций с четырьмя аргументами; в вашу разделяемую библиотеку, которую вы будете использовать в роли плагина, также должны войти те функции, которые *используются* (прямо или косвенно) реализацией ваших команд.

Для связи с интерпретатором Tcl нужно предусмотреть в тексте плагина стартовую функцию, которая имеет имя, составленное из имени вашей библиотеки (без суффикса `.so`) с первой буквой, приведённой к верхнему регистру, и суффикса `_Init`. В нашем примере плагин будет называться `tclembed.so`, так что стартовую функцию нам придётся назвать `Tclembed_Init`. Эта функция принимает на вход один параметр — указатель на интерпретатор, а вернуть должна целое число — либо `TCL_OK`, если всё в порядке, либо `TCL_ERROR`, если произошла ошибка. Пользуясь переданным ей указателем на объект интерпретатора, стартовая функция регистрирует в нём все команды, реализованные в библиотеке, с помощью уже хорошо знакомой нам функции `Tcl_CreateCommand`.

В примерах предыдущего параграфа мы реализовали две команды — `triple` и `ENTER`. Не будем изобретать ничего нового, а просто превратим имеющуюся программу (которая в её текущем виде реализует наш собственный «куцый» интерпретатор Tcl) в плагин. Для этого рядом с имеющейся функцией `main` (которая, естественно, для нашего плагина не нужна) придётся написать стартовую функцию плагина. Чтобы не городить лишних файлов, воспользуемся условной компиляцией для выбора между компиляцией в интерпретатор и компиляцией в плагин: если в командной строке определён макросимвол `PLUGIN`, наш исходный текст будет компилироваться со стартовой функцией для плагина, а в противном случае — с функцией `main`. Выглядеть это будет так:

```
#ifdef PLUGIN
int Tclembed_Init(Tcl_Interp *interp)
{
    Tcl_CreateCommand(interp, "triple", proc_triple, NULL, NULL);
    Tcl_CreateCommand(interp, "ENTER", proc_enter, NULL, NULL);
    return TCL_OK;
}
#else
int main()
     /* ... */
#endif
```

Всё остальное содержимое файла нашего примера мы оставим без изменений, включая секцию директив `#include`. Для создания собственно файла плагина нам потребуется следующая команда:

```
gcc -shared -fpic -Wall -g -DPLUGIN tclembed.c -o tclembed.so
```

Поясним, что флаг `-shared` означает создание разделяемой библиотеки, флаг `-fpic` требует генерации кода, не привязанного к конкретным адресам (ведь на момент сборки неизвестно, в какой области виртуальных адресов будет работать машинный код, составляющий библиотеку); `-DPLUGIN` определяет макросимвол `PLUGIN`, на который мы завязали директивы условной компиляции.

Получив файл плагина (`tclembed.so`), запустим интерпретатор `tclsh`, загрузим плагин командой `load ./tclembed.so` и попробуем его с новыми командами:

```
avst@host:~$ rlwrap tclsh
% load ./tclembed.so
% triple foobar
foobarfoobarfoobar
% ENTER {#> }
#> break
it's a bit strange to use break/continue this way
#> ^D
%
```

Окончательную версию нашего интерпретатора, совмещённого с плагином, читатель найдёт в архиве примеров под именем `tclembed.c`.

12.4. Графические интерфейсы на Tcl/Tk

12.4.1. Библиотека Tk и интерпретатор `wish`

Библиотека Tk позволяет довольно быстро создавать программы с графическим пользовательским интерфейсом, который, как и в знакомой нам библиотеке FLTK для Си++, строится из ***виджетов***[6]. В принципе, эту библиотеку можно подгрузить из обычного интерпретатора Tcl, как, впрочем, и из программы на Си/Си++, но проще будет воспользоваться специально предназначенным для этой цели интерпретатором `wish` (от слов *Window Shell*); этот интерпретатор уже содержит возможности Tk и плюс к тому корректно обрабатывает аргументы командной строки и переменные окружения, имеющие отношение к рисованию окон (для ОС Unix — к работе с протоколом X). У `wish` есть и другие специфические особенности, облегчающие работу с окнами; о некоторых из них мы узнаем позже.

Интерпретатор `wish` можно запустить в интерактивном режиме; при этом он будет читать команды из потока стандартного ввода. Этим можно воспользоваться, чтобы быстро опробовать ту или иную возможность Tcl/Tk. Средствами редактирования вводимой строки интерпретатор не оснащён, так что для интерактивного использования будет удобнее запускать его через `rlwrap`. Сразу после запуска интерпретатор создаст главное окно, не содержащее ничего, и выдаст вам (в терминале) приглашение к вводу (обычно это символ `%`).

По умолчанию заголовок окна будет содержать слово `wish`, но это можно изменить, указав интерпретатору при запуске опцию `-name`:

```
wish -name myApplication
```

Изменить заголовок окна можно и потом, дав в интерпретаторе команду `wm title`:

```
avst@host:~$ wish -name myApplication
% wm title . "Another Title"
```

Именно так лучше всего и поступать. Дело в том, что опция `-name` влияет не только на заголовок окна — она задаёт его *имя*, используемое, например, в подсистеме настроек, которая понадобится нам в §12.4.7. В роли такого имени, т. е. параметра для `-name`, крайне желательно использовать слово-идентификатор без пробелов, начинающееся со строчной латинской буквы; например, имя может совпадать с именем вашей программы.

В приведённом примере заслуживает внимания второй параметр команды `wm`, в качестве которого в нашем примере задана точка «`.`». Этот параметр идентифицирует окно, к которому относится команда; точкой в Tcl/Tk обозначается главное окно вашего приложения.

Коль скоро об этом зашла речь, обсудим систему идентификации окон, тем более что без понимания этой системы сделать мы ничего толком не сможем. Как мы помним, виджеты в составе графического интерфейса *образуют иерархию* в том смысле, что определённые виджеты могут выступать в роли *ведущих* (*master*) для других виджетов, которые в этом случае называются *ведомыми* или *подчинёнными* (*slave*); подчинённые рисуются *внутри* окна своего ведущего и в определённом смысле являются его частью.

В Tcl/Tk кроме отношения «ведущий/ведомый» окна (виджеты) могут находиться ещё и в отношении «родительский/дочерний» (*parent/child*), и это порождает определённую путаницу. *По умолчанию* эти две иерархии окон друг друга дублируют, то есть два виджета, находящиеся в соотношении «ро-

[6] Интересно, что в применении к Tk термины *widget* и *window* означают примерно одно и то же, причём в документации можно встретить их оба; как мы помним (см. §10.11.1), для FLTK виджет и окно — сущности совершенно разные.

дительский/дочерний», находятся также и в отношении «ведущий/ведомый», так что некоторые программисты вообще не видят разницы между этими понятиями; беда в том, что такое дублирование имеет место отнюдь не всегда. В общем же случае отношение «родительский/дочерний» вообще не влияет на *положение* виджетов (один внутри другого или какое бы то ни было ещё) — оно определяет совершенно другое свойство виджетов: при уничтожении родительского окна (это делается командой `destroy`) уничтожаются также и все его потомки.

Родитель в Tk есть у каждого виджета, за исключением главного окна; даже если вы решите вывести на экран несколько независимых окон, имеющих свои рамки, заголовки и т. п. (они называются окнами верхнего уровня, *toplevel windows*, и создаются командой `toplevel`), с точки зрения логики Tcl/Tk они будут дочерними для главного окна. Каждый виджет имеет имя, *начинающееся* так же, как имя его родителя; имена образуются добавлением к имени родителя нового идентификатора через точку, причём идентификатор должен обязательно начинаться со строчной латинской буквы (дальше можно использовать как строчные буквы, так и заглавные, а также цифры и некоторые из знаков препинания). Например, виджеты, родителем которых выступает главное окно, могут называться `.btn`, `.lab12`, `.abraKadabra`, `.box_for_text` или даже просто `.x`; если какой-то из них (пусть он называется `.grp`) сам имеет потомков, то они будут называться `.grp.btn1`, `.grp.list` и т. п. Отметим, что уровни в иерархии виджетов не могут быть пропущены, так что, если у вас есть виджет с именем `.x.y.z.t`, то существуют и виджеты `.x.y.z`, `.x.y` и `.x`.

Создать виджет можно командой, имя которой совпадает с типом виджета; при этом указывается имя виджета и, возможно, дополнительные параметры. Например, создать обычную кнопку можно так:

```
% button .bt -text "Press Me"
.bt
```

Заметим, что *результатом* выполнения команды `button` стало имя созданного виджета (в данном случае `.bt`); как обычно в Tcl, это просто строка, не более того.

Создание кнопки само по себе не приведёт к её появлению на экране; нужно ещё объяснить интерпретатору, как мы хотим её расположить, каковы должны быть её собственные размеры и как всё это должно меняться (если вообще должно) при изменении размеров родительского окна. Этот вопрос не так прост, как хотелось бы; подробности мы рассмотрим позже, а пока прибегнем к самому тривиальному (и, как водится, совершенно непригодному для большинства практических ситуаций) способу: заявим, что кнопка должна быть расположена в фиксированном положении вне зависимости от обстоятельств:

```
% place .bt -x 10 -y 10
```

Результат этой команды — пустой (так же, как, например, для команды `puts`). Теперь кнопка появится на экране — точнее, внутри главного окна; её верхний левый угол будет находиться на 10 пикселей правее и ниже верхнего левого угла самого окна.

Обратим теперь внимание читателя на одну из особенностей `wish`: **имя любого виджета само по себе является командой**, с помощью которой мы можем с этим виджетом выполнять различные действия. Набор действий определяется типом виджета. Общими для всех виджетов являются всего два действия: `cget` — получить значение параметра (любого из тех, что могут быть заданы при создании виджета) и `configure` — установить новое значение (это уже возможно не для всех параметров, некоторые параметры после создания виджета изменить нельзя). Например, мы можем узнать, какой в настоящее время для нашей кнопки установлен текст:

```
% .bt cget -text
Press Me
```

Текст, который мы увидели — это *результат* выполнения команды, так что мы можем, например, занести его в переменную, как это обычно делается в Tcl:

```
% set bt_txt [.bt cget -text]
Press Me
% puts $bt_txt
Press Me
```

Теперь изменим текст кнопки, причём, чтобы убить разом двух зайцев, сделаем его состоящим из двух строк, которые существенно длиннее той, что была:

```
% .bt configure -text "Don't press me! \n I don't like it"
```

Кнопка после этого *изменит свои размеры* как по горизонтали, так и по вертикали, чтобы в ней поместился новый текст. Расстояние от текста до краёв кнопки останется прежним: это так называемый *паддинг*; при желании его тоже можно изменить, например

```
% .bt configure -padx 0 -pady 0
```

вообще уберёт промежутки вокруг текста, уменьшив кнопку так, что текст в ней будет едва помещаться.

Кнопка (как тип виджета) наряду с общими `cget` и `configure` поддерживает ещё два действия: `flash` — мигнуть изображением кнопки на экране и `invoke` — сделать всё так, будто пользователь нажал на кнопку (например, щёлкнул по ней мышкой).

Для контраста рассмотрим другой тип виджета — `label` (*метку*). В отличие от кнопок, метки *пассивны*, то есть никак не реагируют на действия пользователя. Создадим метку и выведем её на экран, но, в отличие от предыдущего раза, воспользуемся более интересным способом указания её расположения — потребуем, чтобы она занимала весь низ главного окна:

```
% label .lab -text "This is a label"
.lab
% pack .lab -side bottom -fill x
```

Изменить текст метки можно точно так же, как и текст кнопки:

```
% .lab configure -text "This is a new text"
```

Метка, созданная и выведенная на экран таким способом, будет по горизонтали занимать всю ширину окна, но по вертикали её размер по-прежнему будет определяться высотой текста; чтобы понять, о чём идёт речь, попробуйте вставить в текст метки несколько переводов строки (`\n`).

Интересно, что обращение к виджету по его имени работает и для главного окна. Поскольку оно имеет имя «`.`» (точка), так же будет называться и команда; например, вот так можно покрасить свободное пространство главного окна в красный цвет:

```
% . configure -bg #ff0000
```

В завершение экспериментов с интерактивным режимом интерпретатора `wish` снабдим нашу кнопку тем, ради чего обычно создаются кнопки — действием, которое должно выполняться в ответ на её нажатие. Действие будет довольно простое: при первом нажатии кнопки текст нашей метки сменится числом `1`, при последующих нажатиях это число будет каждый раз увеличиваться на единицу. Для этого мы заведём переменную `i` с начальным значением `0`, а для кнопки воспользуемся конфигурационным параметром `-command`:

```
% set i 0
0
% .bt configure -command {
   set i [expr $i + 1]
   .lab configure -text $i
}
```

Как видим, в роли команды выступает *скрипт* — строка, содержащая команды Tcl; как обычно, мы заключили её в фигурные скобки. Первая команда увеличивает на единицу значение `i`, вторая — заменяет текст метки `.lab` новым значением переменной `i`. Напомним ещё раз: в Tcl *всё есть строка*, и чисел это тоже касается — они хранятся в текстовом представлении[7], так что

[7] Вообще-то, если совсем честно, то это давно уже не так — интерпретатор Tcl часто вместе со строками хранит и обычное представление чисел; но на поведении команд это не сказывается никак, кроме некоторой прибавки в эффективности, так что семантически мы сей факт обнаружить не можем.

никаких преобразований нам не потребовалось. Конечно, команды, составляющие этот скрипт, можно было бы записать в одну строчку, разделив их символом «;», но это хуже читается и к тому же не влезло бы по горизонтали в полосу набора.

Интерактивный интерпретатор можно использовать не только для экспериментов, подобных нашим; например, если вам нужно визуализировать какой-нибудь процесс вычислений, то один из самых простых способов сделать это — заставить вашу программу выдавать последовательность команд для `wish`, после чего запустить её и `wish` конвейером. Автор книги как-то раз вёл практикум по Прологу и в качестве одной из задач предложил студентам головоломку «ход конём»: найти для шахматной доски заданного размера такую последовательностей ходов шахматного коня, чтобы он при этом обошёл всю доску, побывав в каждой клетке ровно один раз, и вернулся на исходную позицию. Через пару дней двое студентов продемонстрировали программу на Прологе, выдающую последовательность команд для `wish`; повинуясь этим командам, интерпретатор рисовал доску с клетками, текущее положение коня, помечал посещённые клетки, а при попадании в тупиковую ситуацию визуализировался откат, так что было наглядно видно, как именно программа на Прологе перебирает варианты.

Разумеется, в большинстве случаев интерактивный интерпретатор нам не нужен — требуется законченная программа, запускаемая обычным способом. Для этого нам потребуется оформить текст программы как ***скрипт*** в смысле ОС Unix, использующий `wish` в качестве интерпретатора. Мы воспользуемся той же техникой, которую применяли для скриптов на Tcl — интерпретатором скрипта для операционной системы обозначим обыкновенный `/bin/sh`, а нужный нам `wish` запустим с использованием техники *продолженного комментария*. Сам скрипт будет представлять собой простую демонстрационную программу: он покажет в графическом окне свои аргументы командной строки, нарисует кнопку `Ok`, а при нажатии на неё — завершит работу. Выглядеть это будет так:

```
#!/bin/sh
# x11echo.tcl
# The next line starts wish \
exec wish "$0" -name "x11_echo" "$@"

wm title . "tcl/tk x11 echo demo"
label .lb -text "$argv0 $argv" -padx 10 -pady 10
pack .lb -side top -fill x
button .ok -text "Ok" -command exit
pack .ok -side bottom -fill x
```

Приём, используемый здесь при запуске интерпретатора `wish`, мы подробно обсудили на стр. 179; в отличие от запуска `tclsh`, здесь без этой «магии» обойтись уже не получается, поскольку нужно передать интерпретатору *два* параметра командной строки — `-name` и `"x11_echo"`. Можно было бы работать и без этой опции, но в дальнейшем это приведёт к определённым сложностям.

Как видим, обращения к аргументам командной строки производятся точно так же, как и в скриптах для `tclsh`; впрочем, это и не удивительно, ведь `wish` — это практически тот же интерпретатор, просто с дополнительной функциональностью.

Отметим ещё один интересный момент. **Когда интерпретатор `wish` используется в неинтерактивном режиме, завершение выполнения команд, составляющих скрипт, не завершает выполнение интерпретатора**. В действительности можно сказать, что в этот момент — когда последняя команда в скрипте будет обработана — всё только начинается, то есть именно тогда на экране появляется главное окно и запускается *главный цикл*, о существовании которого, впрочем, мы можем только догадываться. В интерактивном режиме работы интерпретатора ничего подобного не происходит: окно появляется (и цикл обработки событий запускается) сразу после запуска интерпретатора, а ситуация «конец файла» в интерактивном интерпретаторе приводит к его завершению.

12.4.2. Взаимодействие с оконным менеджером

Напомним, что ***оконный менеджер*** — это та программа, которая рисует рамки и заголовки окон; именно она позволяет пользователю перемещать окна по экрану и менять их размеры. Вашей программе (например, в ответ на действия пользователя) может потребоваться выполнить функцию, относящуюся к прерогативам оконного менеджера, или запросить от него информацию; для этого Tk предусматривает команду `wm`; как несложно догадаться, это сокращение для слов *window manager*.

С одной из подкоманд команды `wm` — `wm title` — мы уже знакомы, она позволяет задать/изменить заголовок окна. Следующая команда

```
wm attributes . -fullscreen 1
```

распахнёт ваше главное окно на весь экран. Отменить эффект, вернув окну обычный размер и поведение, можно той же командой, заменив `1` на `0`; если у вас есть другие окна верхнего уровня, можно вместо «`.`» использовать их имена, чтобы указать, какое конкретно окно вы желаете сделать полноэкранным.

С помощью команды `wm state` можно убрать окно с экрана, свернуть его в иконку и восстановить обратно:

```
wm state . withdrawn
wm state . iconic
wm state . normal
```

Если последний параметр (собственно сам статус) не указать, команда *возвращает* статус окна. Кроме трёх перечисленных, она может вернуть также статус `icon`, если окно *является* иконкой для какого-то другого окна (которое, в свою очередь, в настоящий момент находится в статусе `iconic`), но установить этот статус нельзя. Аналогичного эффекта можно добиться с помощью команд `wm withdraw`, `wm iconify` и `wm deiconify`; они отличаются от `wm state` тем, что их можно применять к окну, которое ещё никогда не показывалось на экране (`wm state` с такими окнами не работает).

Для управления внешним видом и положением иконки используются подкоманды `iconbitmap`, `iconmask`, `iconname`, `iconphoto`, `iconposition` и `iconwindow`; рассматривать их мы не будем ради экономии места.

Подкоманда `wm geometry` позволяет изменить размер и/или позицию окна на экране. Она принимает два параметра — имя окна (естественно, окно должно быть верхнего уровня, например «`.`») и так называемую *геометрическую строку*, которая как раз и задаёт размер и позицию. Например, команда

```
wm geometry . 200x300
```

установит вашему окну ширину 200 и высоту 300 пикселей; команда

```
wm geometry . 200x300+50+70
```

сделает то же самое, но вдобавок переместит окно так, чтобы его верхний левый угол находился на 50 пикселей правее левого края экрана и на 70 пикселей ниже; замена плюса на минус позволяет отсчитывать эти значения *в другую сторону* — так, команда

```
wm geometry . 250x340-100-200
```

установит окну размер 250 на 340 пикселей, расположив его так, чтобы *нижний правый* угол его находился на 100 пикселей левее правого края экрана и на 200 пикселей выше нижнего. Размер можно опустить; например,

```
wm geometry . -150+50
```

расположит главное окно на 150 пикселей левее правого края и на 50 ниже верхнего, при этом размер окна не изменится. Надо сказать, что геометрические строки — не особенность Tcl/Tk, а традиция X Window: едва ли не любое оконное приложение для X, на чём бы оно ни было написано, понимает опцию командной строки `-geometry`, параметром которой служит именно такая вот строчка, задающая геометрические свойства окна.

Команда `wm` позволяет управлять возможностью изменения размера окна. По каждой из двух координат можно задать минимальный и максимальный размер, а также запретить изменение размера. Например,

```
wm minsize . 100 150
wm maxsize . 200 250
```

зададут для главного окна возможность изменения ширины от 100 до 200 пикселей, а высоты — от 150 до 250. Команда

```
wm resizable . yes no
```

для всё того же главного окна изменение высоты вообще запретит, а возможность изменения ширины сохранит (первый логический параметр отвечает за ширину, второй за высоту, комбинации допускаются любые).

Интересно, что длина и ширина окна может измеряться не только в пикселях; для этого нужно сообщить оконному менеджеру, что данное окно относится к числу *gridded windows*, то есть с его внутренней частью связана некая сетка с фиксированным размером ячеек. Делается это командой `wm grid` или стандартной опцией `-setgrid`. Например, к числу таких окон обычно относятся эмуляторы терминалов, ячейки их внутренней сетки соответствуют знакоместам. Оконный менеджер при изменении размера такого окна всегда оставляет длину и ширину кратной размерам ячейки сетки; больше того, длина и ширина такого окна задаётся уже не в пикселях, а в ячейках сетки — это верно и для `geometry`, и для `minsize`/`maxsize`. Чтобы научиться этим пользоваться, следует прочитать документацию на опцию `-setgrid` и, конечно, поэкспериментировать.

Команда `wm` поддерживает целый ряд других возможностей, в том числе таких, которые работают не со всеми оконными менеджерами. При желании вы можете с ними ознакомиться, обратившись к документации; с другой стороны, если вам потребовалось что-то настолько нетривиальное, имеет смысл подумать, правильно ли вы выбрали язык реализации — всё-таки командно-скриптовое программирование не предназначено для создания больших и сложных программ.

12.4.3. Основные виджеты

Мы уже знакомы с тремя типами виджетов: окнами верхнего уровня (`toplevel`), кнопками (`button`) и метками (`label`). Кроме них, Tk поддерживает также галочки (`checkbutton`), кнопки для выбора из нескольких вариантов (`radiobutton`), простые поля для ввода строки текста (`entry`), текстовые редакторы (`text`), рамки для группирования других виджетов — простые (`frame`) и снабжённые заголовком (`labelframe`), списки для выбора одного или нескольких вариантов (`listbox`), меню (`menu`), кнопки для вызова меню (`menubutton`), текстовые сообщения, отформатированные с заданной шириной (`message`), *шкалы* для выбора числового (возможно, дробного) значения в заданных пределах (`scale`), многопанельные окна с перемещаемыми границами между панелями (`panedwindow`), скроллеры (`scrollbar`), окна для геометрических рисунков (`canvas`); более новые версии Tcl/Tk включают поддержку целого ряда дополнительных виджетов, доступны также сторонние библиотеки, содержащие виджеты.

К сожалению, объём книги не позволит нам рассмотреть даже все основные виджеты, не говоря о дополнительных; более того, нам придётся оставить за кадром часть возможностей тех виджетов, которые всё же будут рассмотрены. Восполнить эти пробелы читатель сможет самостоятельно, обратившись к документации; отметим, что обычно в системе вместе с Tcl/Tk устанавливаются также man-страницы, подробно описывающие все доступные виджеты. Соответствующая секция man обычно называется `3tk`, так что, например, чтобы прочитать про кнопки, нужно дать команду `man 3tk button`. Отказавшись от превращения нашей книги в подобие справочника, мы попытаемся указать основные принципы взаимодействия с виджетами — так, чтобы читатель хотя бы понимал, что конкретно следует искать в справочниках.

Основных способов воздействия на виджет, как мы уже видели, существует два: через опции, задаваемые при создании виджета, и через использование имени виджета в роли команды. Конкретный набор опций виджета, как и конкретное множество подкоманд для виджета как команды, определяется типом виджета; но и среди опций, и среди подкоманд есть некие подмножества, общие для виджетов всех поддерживаемых типов. Подкоманд таких, как мы уже знаем, всего две: `cget` и `configure`; общих (так называемых «стандартных») опций гораздо больше — настолько, что им посвящена отдельная man-страница, расположенная всё в той же секции `3tk`; она так и называется `options`, то есть увидеть её можно, дав команду `man 3tk options`. Строго говоря, опции, описанные в этой man-странице, не обязательно поддерживаются *всеми* виджетами до единого — для попадания в разряд стандартных достаточно, чтобы опция поддерживалась *несколькими* разными типами виджетов, причём работала для них всех одинаково.

Одну такую опцию мы уже видели — это опция `-text`, задающая строку текста, которую виджет должен отобразить. Эта опция применима к виджетам типа `label`, `button` и к некоторым другим — но, конечно, далеко не ко всем. Например, окна верхнего уровня (`toplevel`) и простые рамки (`frame`) этой опции не имеют, но подписанные рамки (`labelframe`) — имеют. Коль скоро речь зашла о тексте, который виджет должен отображать, отметим, что существует более изощрённый способ его задания: опция `-textvariable` задаёт *имя глобальной переменной*, значение которой виджет должен отображать. Эта опция поддерживается, как и `-text`, для всех типов виджетов, отображающих строки; изменение значения переменной (с помощью `set` или любым другим способом) влечёт немедленное изменение текста, отображаемого во всех виджетах, для которых эта переменная задана в качестве `-textvariable`. С другой стороны, если эту опцию применить к виджету `entry`, то привязанная переменная будет менять своё значение по мере того, как пользователь будет редактировать текст в виджете (собственно говоря, так с полями ввода обычно и работают). Этим можно воспользоваться для создания довольно забавного, хотя и бесполезного эффекта: если привязать одну и ту же переменную в качестве `-textvariable` к нескольким разным виджетам, один из которых — `entry`, то, когда пользователь начнёт редактировать текст в поле ввода `entry`, тот же текст появится в остальных виджетах, меняясь синхронно с действиями пользователя.

Отслеживание изменений значения той или иной переменной и выполнение заданных команд, когда такое изменение произошло, — это штатная возможность интерпретатора Tcl. Основным интерфейсом для неё выступает команда `trace`, позволяющая также отслеживать обращения к переменным (то есть доступ к их значениям) и любые действия с именами команд — выполнение, удаление и переименование команды с заданным именем. Обычно все эти возможности используются для отладки, но, как видим, при создании библиотеки Tk для `trace` нашлось другое применение.

В качестве примера опции, поддерживаемой почти всеми виджетами, можно назвать `-relief`, задающую внешний вид границ окошка виджета. Tk умеет отрисовывать вокруг виджета «трёхмерную» рамку так, чтобы виджет выглядел «приподнятым» (`raised`) или «утопленным» (`sunken`); ещё обрабатываются значения `solid` (простая рамка), `groove` (виджет выглядит расположенным на том же уровне, что и окружающее изображение, но окружён рамкой, «утопленной» на манер желобка), `ridge` (то же самое, только рамка, наоборот, «приподнята») и `flat` (рамка отсутствует). Учтите, что для корректной работы `groove` и `ridge` нужно, чтобы толщина рамки была задана не менее чем в два, а лучше в три пикселя. Для этого применяется опция `-borderwidth` или сокращённо `-bd`:

```
% .mylabel configure -bd 3 -relief ridge
```

Многие виджеты поддерживают упоминавшиеся выше опции `-padx` и `-pady`, задающие, сколько места соответственно по горизонтали и вертикали следует добавить вокруг основного содержимого. Некоторые виджеты, такие как `label` или `button`, используют эти опции только для текста, если же их заставить показывать пиктограмму или другое изображение, то про `padx` и `pady` они благополучно забудут. Виджеты, чья основная работа — содержать в себе другие виджеты (а таких мы знаем три: `toplevel`, `frame` и `labelframe`), значения `-padx` и `-pady` отрабатывают всегда. Существуют и виджеты, не знающие таких опций — например, `canvas`.

Большинство виджетов ***пассивны***, то есть с ними что-то происходит только по прямому указанию со стороны выполняющейся программы. Пользователь может сколько угодно тыкать мышкой в окошко, нарисованное, например, виджетом `label`, никакого эффекта от этого не будет[8]. Совсем другое дело — ***активные*** виджеты, такие как известные нам `button` или `entry`, а также `checkbutton`, `radiobutton`, `listbox` и другие: они специально предназначены, чтобы реагировать на действия пользователя, и для них предусмотрены специальные опции, позволяющие такую реакцию настроить.

Все активные виджеты, а также виджет `label` поддерживают опцию `-state`, которая предназначена для «выключения» элементов интерфейса: виджет может быть виден, но не реагировать на действия пользователя — например, если с помощью других активных виджетов пользователь выбрал что-то, делающее использование данного виджета бессмысленным. Обычно такие виджеты выделяются цветом — *становятся серыми* (хотя, конечно, конкретный цвет можно настроить). Интересно, что `-state` не относится к числу стандартных опций, несмотря на то, что поддерживается большим количеством виджетов. Дело тут в том, что для разных виджетов эта опция позволяет задавать разные режимы. Так, кнопка (`button`) может быть в состояниях `normal` (обычный режим), `active` («активная» кнопка — обычно при наведении на неё курсора мыши) и `disabled` (недоступная кнопка, та самая «серая»). В то же время для поля ввода текста (`entry`) набор состо-

[8] Вообще-то даже это можно изменить, задав явным образом реакцию на те или иные события; мы рассмотрим это в одном из следующих параграфов. Иной вопрос, что виджет `label` изначально *не предназначен* для реакции на действия пользователя.

яний другой: `normal`, `disabled` и `readonly`; в последнем случае виджет не позволяет менять своё содержимое. От состояния `disabled` это отличается тем, что виджет выглядит так же, как обычно (не «серым»), даже позволяет выделять текст мышкой (например, чтобы скопировать его), только не даёт его редактировать.

Попробуем научиться работать с основными активными виджетами — и начнём с хорошо знакомой нам кнопки. Пользоваться кнопкой мы уже умеем, поскольку именно она рассматривалась в примерах §12.4.1: надпись на кнопке задаётся опцией `-text`, действия при её нажатии — опцией `-command`, опцию `-relief` к кнопкам лучше не применять, хотя никто этого формально не запрещает; имя виджета-кнопки, используемое в роли команды, помимо общих подкоманд `configure` и `cget`, поддерживает также подкоманды `flash` и `invoke`; так, если ваша кнопка называется `.mybutton`, то команда `.mybutton flash` заставит кнопку на экране «мигнуть», а `.mybutton invoke` сделает вид, будто пользователь нажал на кнопку — то есть (в применении к кнопке) попросту принудительно выполнит команду, ранее заданную опцией `-command` (кстати, если кнопка в настоящее время находится в состоянии `disabled`, то `invoke` для неё не сделает ничего).

Чуть сложнее обстоят дела с галочкой (`checkbutton`). Опции `-text` и `-command` для неё работают точно так же, как и для обычной кнопки, и состояния (`-state`) она поддерживает те же самые (`normal`, `active`, `disabled`), но по своему смыслу она предназначена не для генерации «разовых событий», подобных нажатию на обычную кнопку, а для выбора значения некоторого булевого параметра (включено/выключено, да/нет, 1/0 и т. п.); если использовать имя виджета как команду, набор её подкоманд оказывается существенно шире: стандартные `cget` и `configure`, а также знакомые нам по обычной кнопке `invoke` и `flash` дополняются здесь подкомандами `select`, `deselect` и `toggle` (соответственно привести галочку в положение «отмечена», «не отмечена» и сменить положение на противоположное).

Как ни странно, среди всего перечисленного нет (вообще!) способа *узнать*, а в каком же положении сейчас пребывает наша галочка, то есть нет никакой команды, которая сообщила бы нам, например, `0`, если галочка сброшена, и `1` — если установлена. Доступ к значению галочки организован через *глобальную переменную*, имя которой по умолчанию совпадает с последним токеном имени виджета (например, если наш `checkbutton` имеет имя `.foo.bar`, то переменная будет по умолчанию называться `bar`), но чаще используется переменная, имя которой задано явным образом с помощью опции `-variable`; опять-таки, по умолчанию эта переменная получает значение `0`, когда пользователь снимает галочку, и `1` — когда пользователь галочку устанавливает, но и это можно изменить с помощью опций `-offvalue` и `-onvalue`, задающих значения, которые нужно присвоить переменной, когда галочка соответственно сброшена или установлена. Связь между виджетом и переменной действует в обе стороны: присваивание этой переменной значения, совпадающего с опцией `-onvalue`, принудительно установит галочку, а если присвоить любое другое значение, галочка будет сброшена.

При наличии связанной переменной (а без неё всё равно ничего не получится) использование опции `-command` становится (в отличие от работы с обычной кнопкой) необязательным; коль скоро скрипт всё же будет задан, интерпретатор при изменении состояния галочки *сначала* изменит значение привязанной переменной, и лишь затем исполнит скрипт, что позволяет из скрипта узнать, по какому поводу (при установке или при снятии галочки) его вызвали.

Отметим ещё одну интересную возможность: с помощью опции `-indicatoron` галочку можно превратить в залипающую кнопку — если эту опцию установить в `0`, никакой галочки в квадратике мы больше не увидим, как и самого квадратика, а виджет будет выглядеть как обычная кнопка — просто при нажатии на неё она будет оставаться в нажатом положении, а при повторном нажатии — «отлипать». Ни на чём, кроме внешнего вида виджета, эта опция не сказывается.

В отличие от кнопок и галочек виджеты типа `radiobutton` предназначены для выбора одного варианта из нескольких возможных. Многое из того, что у нас есть для `checkbutton`, работает и здесь: опции `-text`, `-state`, `-command`, даже `-indicatoron`, подкоманды (для имени виджета в роли команды) `cget`, `configure`, `select`, `deselect`, `flash`, `invoke`, разве что нет `toggle`, поскольку она по смыслу к выбору одного из многих не подходит. Остаётся понять, как устроено всё, что составляет специфику `radiobutton`: как объяснить нескольким виджетам, что они составляют единую группу, и если один из них оказывается в положении «выбран», то чтобы остальные автоматически переходили в положение противоположное? А если таких групп несколько, то как разделить однотипные виджеты по этим нескольким группам, чтобы они не мешали друг другу?

Все проблемы снимаются комбинацией двух опций: `-variable` и `-value`. Опция `-variable` нам уже знакома, она задаёт имя глобальной переменной, связанной с виджетом, значение которой синхронизируется с состоянием виджета; ну а `-value` задаёт то значение, которое должно соответствовать «выбранности» данного конкретного виджета. Когда пользователь выбирает данную конкретную радиокнопку, виджет в переменную заносит своё значение (заданное опцией `-value`); как водится, связь тут двусторонняя — если изменить переменную, то и все виджеты, связанные с данной переменной, приведут своё положение в соответствие значению, то есть «выберутся», если значение переменной соответствует их собственному, и «снимут выбор», если значение окажется любым другим. Дальнейшее оказывается совсем просто: виджетам одной группы следует назначить одну и ту же переменную, но разные значения. Друг о друге виджеты, входящие в одну группу, не имеют ни малейшего понятия, но им это и не нужно: они просто следят за переменной, и если пользователь выбрал один из виджетов группы, то этот виджет занесёт в переменную своё значение, а все остальные перейдут в состояние «не выбраны».

Ещё более развесистым оказывается интерфейс поля текстового ввода (виджет `entry`). Для начала отметим, что опция `-text` здесь есть, но зачем она нужна — совершенно непонятно; новички, пытающиеся с помощью этой опции задать начальное значение для редактирования или узнать, что там пользователь навводил, оказываются разочарованы.

И то и другое можно сделать с помощью подкоманд, которых тут в изобилии. Если, например, наш виджет называется `.input`, то результатом команды `.input get` станет текущее содержимое виджета — тот текст, который к настоящему моменту ввёл пользователь; его можно, например, занести в переменную:

```
set myvar [.input get]
```

или сделать с ним что-то ещё, что обычно делают со значениями. Принудительно изменить текст в виджете можно, сначала удалив весь текст с помощью подкоманды `delete`, а потом вставив новый текст командой `insert`:

```
.input delete 0 end
.input insert 0 "This is the new text"
```

Поясним, что подкоманда `delete` требует двух параметров — индекс первого символа, который нужно удалить, и индекс символа, *следующего за последним символом*, который нужно удалить. Нумерация идёт с нуля. Например, если в поле ввода сейчас строка `"abcdef"` и мы выполним команду `.input delete 0 3`, то первые три символа исчезнут. Слово `end` — это специальное значение индекса, означающее позицию сразу за последним *существующим* символом, так что первая из двух вышеприведённых команд удаляет из виджета весь имеющийся текст. Вторая команда вставляет заданный текст *перед* указанной позицией, так что нулевая позиция означает вставку в начало — впрочем, виджет у нас на момент выполнения этой команды всё равно был пуст. Кроме слова `end`, специальные значения для индексов включают ещё `insert` — позицию, где в настоящее время стоит курсор, и несколько других.

Подкоманды для виджета типа `entry` включают также `icursor`, с помощью которой можно изменить позицию курсора, `selection`, позволяющую установить, какой фрагмент вводимого текста будет выбран (например, для копирования) и ещё с десяток подкоманд, рассматривать которые мы не будем. Так или иначе, обычно работу с виджетом типа `entry` организуют через хорошо знакомые нам *привязанные переменные*, это намного проще, чем заморачиваться с подкомандами.

Здесь нас ждёт довольно странный сюрприз: привычной опции `-variable` у виджета почему-то нет, а привязка переменной выполняется с помощью опции `-textvariable` (что особенно неожиданно, если учесть, что опция `-text` тут хотя и есть, но совершенно никак не работает). Впрочем, это нужно просто запомнить, и больше проблем не будет: значение привязанной переменной меняется каждый раз, когда пользователь изменяет текст в поле ввода, а связь между переменной и виджетом, как водится, двусторонняя — при изме-

нении переменной изменится и текст в поле ввода, к которому эта переменная привязана.

Вообще виджеты типа `entry` предоставляют довольно много разнообразной функциональности, которую мы в основном оставим за рамками книги. Упомянем только один важный момент: с полем ввода можно связать так называемую *валидацию* — определённые процедуры проверки «допустимости» вводимого текста; иначе говоря, с помощью этого механизма можно сделать так, чтобы пользователь не мог ввести текст, не удовлетворяющий определённым условиям, чтобы при попытке это сделать либо выдавались какие-нибудь сообщения, либо текст «сам собой» как-то менялся и т. д. Подробно рассказывать про валидацию мы не будем, упомянем только, что с помощью опции `-validatecommand` (или `-vcmd`) задаётся скрипт, который должен проверять соответствие текста правилам, а опция `-validate` указывает, *когда* следует проводить валидацию — при любом редактировании, при получении или потере фокуса ввода либо никогда. За дальнейшими пояснениями следует обратиться к документации, включая man-страницу `entry` из секции `3tk`.

Остальные виджеты мы рассматривать не будем; можно надеяться, что читатель успел получить определённое виденье того, как устроены виджеты библиотеки Tk, и по мере надобности сможет самостоятельно освоить те из её возможностей, которые мы вынуждены оставить за кадром.

12.4.4. Положение и размеры

Если читатель пробовал создавать графические интерфейсы пользователя с помощью библиотеки FLTK, которую мы рассматривали ранее, или с помощью любых других библиотек виджетов, он мог заметить, что львиная доля времени при этом уходит на то, чтобы расположить виджеты друг относительно друга и заставить их корректно вести себя при изменении размеров главного окна. Эти два вопроса — взаимное расположение элементов интерфейса и их реакция на смену размера окна — требуют особого внимания, и можно сказать, что в библиотеке Tk с этим особым вниманием всё в порядке.

Для начала нам придётся перевести с английского понятие *geometry manager*, и мы, чтобы долго не думать, условимся применять словосочетание ***геометрический менеджер***, хотя, конечно, такой перевод оставляет желать много лучшего. В терминологии Tk так называется некая программно реализованная сущность, позволяющая задать то, как ведомые виджеты располагаются относительно своего ведущего и как будут меняться их взаимное расположение и/или размеры, если изменится размер ведущего виджета; более того, часто, хотя и не всегда, геометрический менеджер, владея информацией о размерах и расположении ведомых, *сам изменяет размер ведущего окна* — это явление в документации называется *geometry propagation*, что можно приблизительно перевести как «продвижение геометрии».

Геометрических менеджеров в Tk имеется три. Самый примитивный из них называется `place`, он позволяет для каждого подчинённого виджета задать его положение и размеры либо явно в виде фиксированных чисел, либо с неким коэффициентом относительно размеров ведущего. Каждый, кто хотя бы раз в жизни пытался скомпоновать графический пользовательский интерфейс, знает, что таким способом получить прилично выглядящее окно довольно сложно.

Второй геометрический менеджер называется `grid` и действует по принципу «решётки», имеющей строки и столбцы; каждый подчинённый виджет может занимать одну ячейку решётки или прямоугольную область, состоящую из таких ячеек. Геометрический менеджер сам вычисляет ширину каждого столбца и высоту каждой строки решётки, исходя из размеров расположенных там подчинённых виджетов — так, чтобы они все помещались в отведённые им ячейки или области; естественно, в этот процесс можно вмешаться.

Третий и, пожалуй, самый популярный из геометрических менеджеров Tk называется `pack`; он позволяет лишь в самых общих чертах указать, в какой части экрана следует разместить каждый из подчинённых виджетов. Во многих случаях это позволяет «слепить» сложное диалоговое окно очень быстро, и оно будет вполне пристойно смотреться, но вот если от вас кто-то потребует какого-то определённого расположения элементов интерфейса, убедить `pack` делать всё «чуть-чуть иначе» будет довольно сложно. Кроме того, при использовании `pack` виджеты часто приходится группировать с помощью *промежуточных* виджетов (`frame` и `labelframe`), которые с точки зрения соседних виджетов выступают как единое целое, но внутри себя реализуют собственное управление геометрией, причём часто не такое, как у их ведущего. Несмотря на всё сказанное, `pack` — это несомненный прорыв в области расположения интерфейсных элементов, поскольку в подавляющем большинстве случаев его применение позволяет сэкономить огромное количество времени в сравнении с другими подходами к геометрии.

Прежде чем обсуждать возможности этих трёх геометрических менеджеров в деталях, сделаем одно важное замечение. Везде, где в Tk подразумевается размер или расстояние — в том числе при задании положения или размеров окна, размера рамки, паддинга (т. е. при использовании известных нам опций `-padx` и `-pady`) — соответствующие величины можно задать не только в пикселях, как мы поступали до сей поры, но и в других единицах измерения. Например, если написать что-то вроде `-padx 10`, указав параметром опции обычное целое число, Tk воспримет это число как количество пикселей, но если написать «`-padx 2c`», это будет означать два сантиметра — как можно догадаться, Tk воспринимает в качестве обозначения сантиметров букву `c` (от слова *centimeters*). Аналогичным образом `i` означает дюймы (*inches*), `m` — миллиметры (*millimeters*), а `p` — типографские пункты (*points*), которые, как известно, соответствуют $\frac{1}{72}$ дюйма (именно в этих пунктах обычно измеряется высота шрифта — 12, 14 и т. п.).

Начнём с рассмотрения примитивного `place`. Здесь всё довольно просто: чтобы поместить виджет в его ведущее окно с помощью этого геометрического менеджера, нужно воспользоваться командой `place`, первым параметром указать имя виджета, а дальше с помощью опций задать как минимум координаты и плюс к тому, возможно, длину и/или ширину. Координаты можно задать явно с помощью опций `-x` и `-y`; второй способ их задания — *относительный*: используются опции `-relx` и `-rely`, параметрами им служат дробные числа, обычно от `0` до `1`, задающие коэффициент соответственно к ширине и высоте ведущего окна.

Ширину и высоту размещаемого виджета можно не указывать, тогда геометрический менеджер использует информацию, получаемую от самого виджета; но если вы её всё же решите задать, то это тоже можно сделать явным указанием размеров (опции `-width` и `-height`) либо указанием размеров в виде доли от соответствующего размера ведущего окна (опции `-relwidth` и `-relheight`).

Полезно помнить о существовании опции `-anchor`, которая указывает, *какую именно точку* виджета следует расположить в месте, заданном опциями `-x`/`-y` или `-relx`/`-rely`. Эта опция использует строковые значения, соответствующие *сторонам света*: `n` — север (верх), `s` — юг (низ), `w` — запад (лево), `e` — восток (право). По умолчанию используется левый верхний угол, что соответствует значению `nw` (*northwest*, т. е. северо-запад), но можно также указать любой другой угол (`ne`, `sw`, `se`), середину одной из четырёх сторон (`n`, `s`, `e`, `w`) или центр виджета (так и называется `center`).

Для примера попробуем создать текстовую метку, расположенную в самом верху главного окна, а по горизонтали — в середине. Делается это примерно так:

```
label .lb -text "This is a label" -relief raised
place .lb -y 0 -relx 0.5 -anchor n
```

Ещё одна интересная особенность менеджера `place` состоит в том, что явные и относительные величины можно задавать одновременно, и тогда они *суммируются*. Например, `-x 20 -relx 0.5` означает «на 20 пикселей правее середины по горизонтали», если же надо, наоборот, левее, то никто не мешает значение для `-x` указать отрицательное.

Команда `place forget` позволяет убрать виджет из текущей геометрической конфигурации; полезны также команды `place info`, которая выдаёт текущую конфигурацию для заданного виджета, и `place slaves`, значением которой будет *список* ведомых для заданного ведущего.

Геометрический менеджер `grid` сложнее адекватно описать и труднее освоить, но если ваш графический интерфейс должен состоять из большого количества виджетов, то придать ему пристойный вид с помощью `grid` получится намного быстрее.

«Решётка», «сетка» или как там ещё нам придёт в голову перевести английское слово *grid*, существует только в нашем воображении, причём её в некотором смысле можно считать бесконечной. В действительности правильнее предполагать, что в этой сетке столько колонок и столько строк, сколько мы явным образом успели задействовать: например, если какой-нибудь из виджетов мы расположили в пятой строке и седьмой колонке, а какой-то

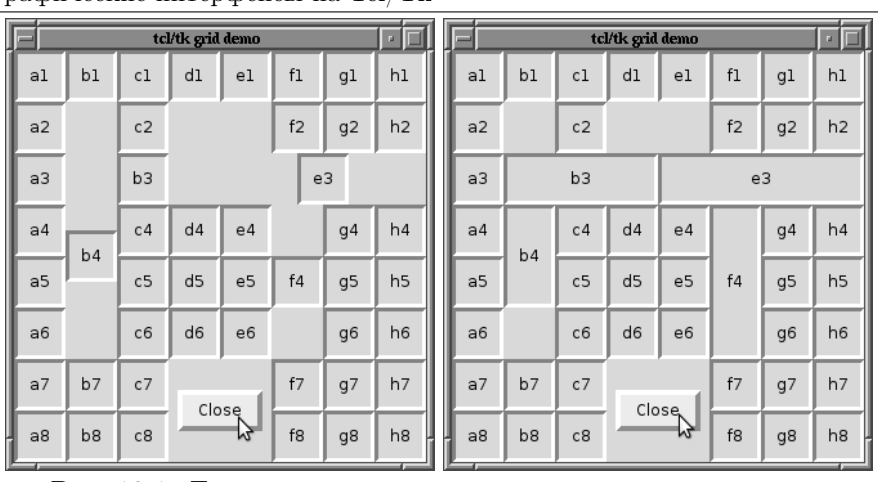

Рис. 12.1. Демонстрация геометрического менеджера `grid`

другой – в девятой строке и третьей колонке, то сетка у нас будет иметь размер 9x7, даже если только эти два виджета у нас и будут; но никто не мешает в любой момент загнать ещё один виджет в какую-нибудь ячейку с координатами (20, 20), тем самым увеличив сетку.

Виджеты можно вставлять в сетку по одному, явным образом указывая для каждого из них позицию: опция `-row` задаёт номер строки, опция `-column` — номер столбца (учтите, что строки и столбцы нумеруются с нуля), полезны также опции `-columnspan` и `-rowspan`, позволяющие указать, *сколько* столбцов и строк будет занимать вставляемый виджет (по умолчанию он занимает одну строку и один столбец). Например, команда

```
grid .lab1 -row 2 -column 2 -columnspan 3
```

вставит виджет `.lab1` в сетку, связанную с его ведущим окном (в данном случае это главное окно, «`.`»), начиная с ячейки $(2, 2)$, причём этот виджет займёт три смежные ячейки в той же строке — саму $(2, 2)$ и ещё $(3, 2)$ и $(4, 2)$. Но действовать таким образом — долго и скучно. Существенно лучше возможности `grid` раскрываются, если формировать наш набор виджетов сверху вниз, строка за строкой. По умолчанию команда `grid` каждое обращение к себе, предполагающее вставку виджетов, рассматривает как инструкцию по созданию строки целиком слева направо; естественно, для этого можно указать больше одного виджета. Параметры `-row` и `-column` при этом не используются вовсе. Больше того, вместо `-columnspan` и `-rowspan` можно воспользоваться *замещающими символами*, указываемыми вместо очередного виджета: `x` означает, что данную клетку в текущей строке следует оставить пустой, `-` (минус) растягивает предыдущий виджет ещё на одну клетку (например, три таких минуса имеют тот же эффект, который имела бы опция `-columnspan 4`), а символ `^` делает то же самое по вертикали — растягивает вниз (то есть на текущую строку) виджет, расположенный в данном столбце в предыдущей строке сетки.

Чтобы проиллюстрировать сказанное, мы для начала сделаем процедуру, которая позволит нам создать много однотипных окошек — это будут виджеты типа `frame` в «утопленной» рамке, в самом центре которых располагаются метки (`label`) с «идентификатором» данного окошка; в роли таких идентификаторов мы воспользуемся «шахматными» координатами вроде `a1`, `h7`, `e4` и т. п. На трюк с фреймами нам приходится пойти из-за того, что виджет типа `label`, содержащий текст, принимает только размеры, выраженные в строках и буквах, а не в пикселях, тогда как нам для эстетичного вида нужны правильные квадратики. Процедуру создания таких окошек мы назовём `m` (от слова *make*):

```
set dim 40
proc m { id } {
    global dim
    frame .fr$id -relief sunken -bd 3 -height $dim -width $dim
    label .fr$id.lb -text "$id"
    place .fr$id.lb -relx 0.5 -rely 0.5 -anchor center
    return .fr$id
}
```

Как видим, внутри каждого фрейма для расположения метки по центру мы воспользовались описанным выше геометрическим менеджером `place`; с такой простой задачей он справляется лучше всех.

Имея такую процедуру, мы можем из однотипных квадратиков построить некое подобие мозаики, заодно продемонстрировав в деле все три замещающих символа `x`, `-` и `^`:

```
grid [m a1] [m b1] [m c1] [m d1] [m e1] [m f1] [m g1] [m h1]
grid [m a2] x      [m c2] x      x      [m f2] [m g2] [m h2]
grid [m a3] [m b3] -      -      [m e3] -      -      -
grid [m a4] [m b4] [m c4] [m d4] [m e4] [m f4] [m g4] [m h4]
grid [m a5] ^      [m c5] [m d5] [m e5] ^      [m g5] [m h5]
grid [m a6] x      [m c6] [m d6] [m e6] ^      [m g6] [m h6]
grid [m a7] [m b7] [m c7] x      x      [m f7] [m g7] [m h7]
grid [m a8] [m b8] [m c8] x      x      [m f8] [m g8] [m h8]
```

Для завершения программы предусмотрим кнопку `Close`, которую расположим в пустом месте в середине двух нижних строк:

```
grid [button .b -text Close -command exit -bd 4] \
    -row 6 -column 3 -rowspan 2 -columnspan 2
```

Полученный результат будет выглядеть, как показано на рис. 12.1 (слева)[9]. Результат может нас несколько обескуражить: элементы `b3` и `f4` оказались явно не на своём месте, а `b4` и `e3` вообще куда-то съехали. Так происходит из-за того, что под эти четыре элемента выделено больше одной клетки, а по умолчанию в такой ситуации — точнее, в ситуации, когда виджет меньше, чем выделенное под него место — `grid` располагает его по центру отведённого пространства. Если посмотреть на рисунок внимательнее, можно понять, что произошло именно это. Впрочем, никто не заставляет удовлетворяться полученным эффектом, нужно просто объяснить интерпретатору, что мы хотим чего-то другого. Мы воспользуемся опцией `-sticky`, которая позволяет указать, к каким сторонам клетки вставляемый виджет должен «прилипнуть». Стороны указываются уже знакомыми нам буквами, обозначающими стороны света: `n` означает верх клетки (север, *north*), `w` — левый край (запад, *west*) и т. д. Если указать две смежные стороны, виджет будет располагаться в углу, если же указать две *противоположные* — то виджет между ними растянется. Мы, не мудрствуя лукаво, укажем все четыре буквы, в результате чего все наши виджеты (кроме кнопки `Close`, для которой мы этого не сделаем) будут занимать отведённое пространство целиком:

```
grid [m a1] [m b1] [m c1] [m d1] [m e1] [m f1] [m g1] [m h1] -sticky nswe
grid [m a2] x      [m c2] x      x      [m f2] [m g2] [m h2] -sticky nswe
grid [m a3] [m b3] -      -      [m e3] -      -      -      -sticky nswe
grid [m a4] [m b4] [m c4] [m d4] [m e4] [m f4] [m g4] [m h4] -sticky nswe
grid [m a5] ^      [m c5] [m d5] [m e5] ^      [m g5] [m h5] -sticky nswe
grid [m a6] x      [m c6] [m d6] [m e6] ^      [m g6] [m h6] -sticky nswe
grid [m a7] [m b7] [m c7] x      x      [m f7] [m g7] [m h7] -sticky nswe
grid [m a8] [m b8] [m c8] x      x      [m f8] [m g8] [m h8] -sticky nswe
```

Результат показан на рис. 12.1 справа; пожалуй, это более наглядная демонстрация.

У нашей программы можно будет обнаружить довольно неприятный недостаток: если средствами оконного менеджера (например, с помощью мышки) поменять размеры главного окна, виджеты останутся на месте — не поменяют ни своих размеров, ни тем более своего расположения. Исправить это можно, сообщив нашему геометрическому менеджеру, что колонки и строки имеющейся сетки могут менять размер — и как именно они это должны делать:

```
grid columnconfigure . all -weight 1 -uniform a
grid rowconfigure . all -weight 1 -uniform b
```

Первая из этих команд указывает, что все (слово `all`) имеющиеся к настоящему моменту колонки будут относиться к одной «группе однородности» (*uniform group*), которую мы назвали `a`, и получат одинаковый вес, равный `1`; это, попросту говоря, означает, что все колонки могут растягиваться и сжиматься, но должны это делать одинаково. Вторая команда делает то же самое для строк (группа однородности в этот раз называется `b`). Полностью текст нашего примера читатель найдёт в файле `griddemo.tcl`.

Возможности подкоманд `columnconfigure` и `rowconfigure` достаточно широки, как и возможности команды `grid` в целом. Заинтересованному читателю мы можем порекомендовать man-страницу по этой команде, здесь же отметим только ещё один момент: команда `grid` имеет, помимо прочего, подкоманды `forget`, `info` и `slaves`, работающие точно так же, как и соответ-

[9] См. сноску 31 на стр. 93.

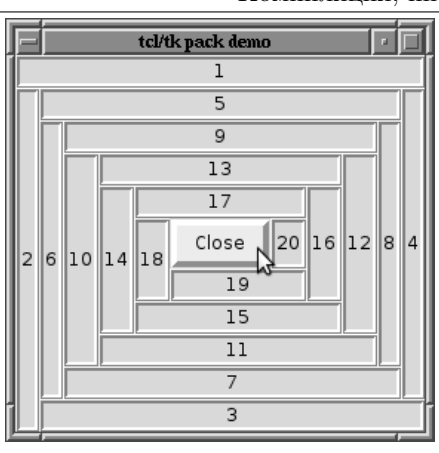

Рис. 12.2. Демонстрация геометрического менеджера `pack`

ствующие подкоманды для рассмотренного выше `place`; в дополнение стоит отметить подкоманду `remove`, которая отличается от `forget` тем, что виджет временно убирается из сетки, но все связанные с ним параметры `grid` продолжает помнить, так что его можно вставить обратно, указав только имя виджета.

Третий и последний из геометрических менеджеров, представленных в Tk — `pack` — действует по достаточно простому принципу: размещает очередной виджет вдоль одной из четырёх сторон оставшегося *пустого пространства* ведущего окна; в оригинальной английской документации это пустое пространство называют словом *cavity*. Первоначально пустое пространство совпадает со всем окном целиком; затем при добавлении очередного виджета `pack` выделяет этому виджету прямоугольную область вдоль одной из сторон — верхней (`top`), нижней (`bottom`), левой (`left`) или правой (`right`). Когда область выделяется сверху или снизу, её ширина будет равна ширине всего оставшегося пустого пространства, а высота выбирается равной высоте виджета; при выделении области слева или справа, наоборот, высота области определяется высотой пустого пространства, а её ширина — шириной виджета.

Основных опций, управляющих размещением виджетов, можно выделить три. Прежде всего это опция `-side`, параметром ей служат слова `top`, `bottom`, `left` и `right`; она, как несложно догадаться, указывает, с какой стороны пустого пространства разместить очередной виджет; по умолчанию используется значение `top`, то есть очередной виджет размещается сверху. Опция `-fill` принимает значения `none`, `x`, `y` и `both`; она указывает, следует или нет геометрическому менеджеру «растягивать» виджет, если выделенное под него пространство оказалось больше, чем он сам; `x` означает растягивание только по горизонтали, `y` — только по вертикали, `both` — в обоих направлениях, так что виджет займёт всё отведённое пространство. Опция `-expand`, принимающая логическое значение, разрешает или запрещает увеличивать пространство, выделяемое для данного виджета в случае, если после размещения всех виджетов в ведущем окне всё ещё остаётся свободное пространство; `pack` попытается распределить свободное место между всеми ведомыми виджетами, для которых параметр `-expand` установлен в истину. Стоит подчеркнуть разницу в *объектах приложения* для опций `-fill` и `-expand`: первая касается размеров *самого виджета*, тогда как вторая — размеров *выделяемой под него области в ведущем окне*, которую виджет совершенно не обязан занять целиком.

Для примера расположим в главном окне по спирали против часовой стрелки (сверху, слева, снизу, справа, сверху, ...) двадцать меток, содержащих числа от `1` до `20`, а в середину всего этого великолепия вставим кнопку `Close`. Скрипт будет выглядеть так:

```
#!/bin/sh
# packdemo.tcl
# The next line starts wish \
exec wish "$0" -name "pack_demo" "$@"

wm title . "tcl/tk pack demo"

set i 0
for { set n 0 } { $n < 5 } { incr n } {
    foreach sd { top left bottom right } {
        incr i
        label .l$i -text $i -relief ridge -bd 3
        pack .l$i -side $sd -fill both -expand 1
    }
}

button .b -text Close -command exit -bd 4
pack .b -side left -expand 1 -fill both
```

Результат работы скрипта показан на рис. 12.2. Чтобы понять на уровне интуиции, как именно опции `-expand` и `-fill` влияют на происходящее, обязательно возьмите этот скрипт из архива примеров и попробуйте позапускать его, устанавливая разные значения для этих опций; учтите, что результаты части экспериментов проявятся только при изменении размеров окна средствами оконного менеджера (например, с помощью мышки).

Как и два других геометрических менеджера, `pack` обрабатывает подкоманды `forget`, `info` и `slaves`.

По умолчанию геометрические менеджеры `grid` и `pack` сами вычисляют минимальные размеры ведущего окна (в наших примерах в его роли выступает главное окно программы) и именно такие размеры ведущему окну устанавливают. Это, как уже упоминалось, называется *продвижением геометрии* (*geometry propagation*). В применении к окну верхнего уровня это проявляется довольно неожиданным способом: пользователь с помощью мышки по-прежнему может менять размеры окна, но любые попытки изменить его размеры из самой программы попросту перестают работать. Например, команда вроде

```
. configure -height 400 -width 400
```

по идее должна установить главному окну размер 400 x 400 пикселей, но если в главном окне мы применили `pack` или `grid` и при этом не отключили *geometry propagation*, эта команда *вообще не даст никакого эффекта*. Никаких ошибок она тоже не выдаст — интерпретатор её просто тихо проигнорирует. Исправить ситуацию можно, запретив геометрическому менеджеру производить продвижение геометрии для главного окна:

```
pack propagate . 0
```

Например, если в наш скрипт `packdemo.tcl` перед циклом добавления виджетов вставить строки

```
. configure -height 400 -width 400
pack propagate . 0
```

— то начальный размер главного окна составит 400 x 400 пикселей, а все метки будут соответствующим образом растянуты, чтобы полностью занять пространство окна.

12.4.5. Управление шрифтами

Во многих случаях нас вполне устраивает то, как Tk отображает в виджетах текстовые строки, так что можно не думать о выборе шрифта; но в какой-то момент шрифт может показаться нам, к примеру, слишком мелким или мы захотим использовать шрифты разных размеров, выделить какой-нибудь текст ожирнением или курсивом, а то и поменять очертания символов — то, что в типографиях обычно называют гарнитурой. Любой из перечисленных вопросов потребует умения обращаться со шрифтами — точнее, с тем программным интерфейсом, который для этой цели предоставляет библиотека Tk.

Сразу же следует смириться с одним очень серьёзным ограничением, накладываемым библиотекой: **любой виджет может отображать свой текст только каким-то одним шрифтом**, так что, например, выделить курсивом или ожирнением отдельные слова в какой-нибудь надписи не получится[10]. Для настройки этого единственного шрифта все виджеты, подразумевающие отображение текста, обрабатывают стандартную опцию `-font`.

В документации описано довольно много вариантов для задания параметра опции `-font`, но далеко не все они переносимы, и бóльшая их часть требует громоздких объяснений. Мы ограничимся только двумя наиболее простыми и общеупотребительными вариантами. Первый из них предполагает явное задание имени гарнитуры, размера и (опционально) стилистических

[10] Точнее, этого можно достичь, если сформировать одну надпись из нескольких виджетов, например, типа `label`, но чрезвычайная трудоёмкость такого решения делает его бессмысленным.

особенностей шрифта; информация оформляется в виде *списка* (напомним, что списком в Tcl считается строка, разделённая пробелами на отдельные элементы). Первым элементом списка выступает имя гарнитуры, вторым — размер, далее можно добавить дополнительно слова `normal`, `bold` (жирный шрифт), `roman` («прямой» шрифт), `italic` (курсив), `underline` (подчёркнутый), `overstrike` (вычеркнутый). Tk гарантирует на всех платформах и во всех средах корректную обработку по меньшей мере трёх имён гарнитур: `Times` (классический шрифт с засечками), `Helvetica` (шрифт без засечек) и `Courier` (моноширинный шрифт). Например,

```
label .lb -text {This is a label} -font {Courier 20 italic}
```

создаст метку, текст которой будет отображаться наклонным моноширинным шрифтом размером в 20 пунктов.

Чуть более сложным для описания, но намного более правильным в плане управляемости оказывается другой способ, предполагающий использование команды `font` для связывания некоторого имени (идентификатора, выбираемого программистом) с определённой конфигурацией шрифта. Создание нового имени для шрифта производится с помощью `font create`, которая принимает в качестве опций `-family` (имя гарнитуры, например `Courier`), `-size` (размер), `-weight` (ожирнение; здесь только два варианта — `normal` и `bold`), `-slant` (наклонение; `roman` или `italic`), `-overstrike` и `-underline` (для двух последних параметром служит логическое значение). Новое имя шрифта затем можно использовать в качестве параметра для опции `-font` при создании и настройке виджетов. Например, двумя командами

```
font create my_font -family Courier -size 20 -slant italic
label .lb -text {This is a label} -font my_font
```

мы создадим такую же метку, как и в предыдущем примере, но при этом с настройками её шрифта будет связано имя `my_font`. Наиболее очевидное достоинство такого варианта — что это имя можно указать при создании других виджетов, которые по нашему замыслу должны использовать тот же шрифт, что и метка `.lb`; но есть и ещё один момент: шрифтовые настройки, связанные с именем, мы можем в любой момент изменить с помощью команды `font configure` (она принимает тот же набор опций, что и `font create`). Так, если выполнить команду

```
font configure my_font -size 15
```

— то *все* виджеты, которым была задана опция `-font my_font`, разом изменят размер шрифта с 20-го на 15-й.

12.4.6. Обработка событий

Как мы видели, некоторым виджетам через их опции можно указать, какой набор команд Tcl («скрипт») выполнить в случае определённых действий пользователя. Для кнопок это опция `-command`, и ею в более-менее простых случаях вполне можно ограничиться: наша программа дождётся, пока пользователь «нажмёт» на кнопку, и лишь после этого проанализирует строки, введённые в поля ввода, положения галочек и кнопок выбора, а также состояние других виджетов, и на основе результатов анализа что-то там полезное сделает. Теоретически ничто не мешает «повесить» активные действия на валидацию ввода в поле `entry`, воспользовавшись опцией `-vcmd`, хотя удобство такого подхода сомнительно: редактируя строку в поле ввода, пользователь может быть психологически не готов к тому, что программа вдруг что-то начнёт делать, среагировав на некую комбинацию введённых символов (которая к тому же могла возникнуть случайно, по ошибке и т. п.)

Обработка событий с помощью команды `bind` существенно расширяет возможности реагирования на действия пользователя. Общая идея тут довольно проста: вы сообщаете интерпретатору, что́ нужно сделать (т. е. какие команды выполнить) при наступлении того или иного *события*. Очевидными примерами событий служат:

- действия пользователя — нажатие и отпускание клавиш на клавиатуре и кнопок мыши, перемещение мыши, смена фокуса ввода, изменение положения или размеров окна средствами оконного менеджера;
- некоторые действия вашей программы в отношении виджетов — например, когда с помощью того или иного геометрического менеджера виджет показывается на экране или, наоборот, скрывается, или когда виджет уничтожается командой `destroy`;
- так называемые «виртуальные» события, которые вы вводите сами — чтобы такое событие произошло, нужно дать соответствующую команду.

Обработка событий, заданная с помощью `bind`, никак не зависит от типа виджета — например, никто не запрещает обрабатывать движение мыши и нажатия на её кнопки в окнах пассивных виджетов, таких как `label` или `frame`. Можно в первом приближении считать, что обработка событий производится независимо от реализации конкретных виджетов, то есть та часть невидимого нам главного цикла, которая отвечает за диспетчеризацию событий, сначала определяет, не задана ли для произошедшего события какая-нибудь обработка, и лишь затем передаёт событие в подпрограммы, отвечающие за поведение виджета.

Чтобы было понятнее, о чём идёт речь, приведём для начала несколько примеров установки обработчиков событий. Команда

```
bind . <KeyPress-Escape> exit
```

устанавливает обработку нажатия клавиши Escape для главного окна и всех находящихся в нём виджетов, причём обработка состоит в выполнении команды `exit`, завершающей работу вашей программы. Попросту говоря, теперь можно выйти из программы, нажав Escape. Более сложной выглядит команда

```
bind . <Control-Shift-KeyPress-F3> { puts "Very strange event" }
```

Эта команда предписывает реагировать на нажатие клавиши F3, но только при условии, что при этом одновременно нажаты клавиши Control и Shift. Кстати, слово `KeyPress` можно не писать, если задано имя клавиши, так что две предыдущие команды можно написать короче:

```
bind . <Escape> exit
bind . <Control-Shift-F3> { puts "Very strange event" }
```

Если вы хотите задать реакцию на ввод с клавиатуры простого печатного символа (за исключением пробела и знака `<`), можно сделать ещё проще — например, выход из программы по нажатию на `q` можно запрограммировать так:

```
bind . q exit
```

Учтите только, что подобные вещи совершенно не сочетаются с наличием в вашей программе полей для ввода строк: в самом деле, пользователь может захотеть ввести строку, содержащую букву `q` (например, слово *sequence*), и если программа при этом завершится, пользователя это может, мягко говоря, удивить.

Следующие две команды создают метку с надписью, объясняющей, что с ней надо сделать, и устанавливают реакцию на тройное (sic!) нажатие правой кнопки мыши; когда пользователь справится с задачей, текст метки изменится, констатируя сей факт:

```
label .lab1 -relief solid -text { Triple-right-click here! }
bind .lab1 <Triple-ButtonPress-3> {
    .lab1 configure -text { Well, you did it }
}
```

Кстати, тройное нажатие — не предел, Tk поддерживает ещё и слово `Quadruple` для обработки четырёхкратных нажатий (и, конечно, слово `Double` для привычного двойного клика). Стоит ли применять в вашем интерфейсе четырёхкратные нажатия, обдумайте сами.

Для обозначения виртуальных событий используются двойные угловые скобки. Например:

```
bind . <<TimeToGo>> { puts "So let's go!" }
```

Чтобы заставить эту команду `puts` выполниться, событие нужно *сгенерировать*:

```
event generate . <<TimeToGo>>
```

Теперь, когда мы более-менее поняли, как это всё происходит, можно описать команду `bind` более детально. Она принимает три аргумента. Первый аргумент определяет, *в каких окнах* должно произойти событие, чтобы на него отреагировать, второй параметр задаёт собственно событие, и третий — скрипт, который нужно выполнять при его наступлении.

В большинстве наших примеров в роли окна выступало главное окно программы, именуемое точкой «`.`», и только в одном из примеров использовался конкретный виджет `.lab1`. Вообще этот параметр может принимать одну из трёх форм. Если указать любое окно верхнего уровня (`toplevel`), включая, естественно, и главное окно приложения, то событие будет обрабатываться для самого этого окна и всех виджетов, размещённых в нём. Указание любого окна (виджета), не являющегося окном верхнего уровня, указывает, что события нужно обрабатывать, если они произойдут именно в области этого виджета. Последняя, третья форма первого параметра — так называемый *тег окна* (*window tag*). В роли такого тега может выступать *имя типа виджета* — например, `button` или `label`; обработка события при этом устанавливается для всех виджетов этого типа. Также можно использовать слово `all`, соответствующее всем окнам. Наконец, можно придумать какой-то свой тег и пометить им некий произвольный набор окон/виджетов с помощью команды `bindtags`.

С помощью этой команды можно также отменить для конкретного окна действие предопределённых тегов, в том числе тега `all`, и изменить их порядок; больше того, на самом деле на события, относящиеся к виджету, можно установить обработку с использованием его имени или имени его окна верхнего уровня лишь по той причине, что и его имя, и имя его окна верхнего уровня тоже входят в список его тегов, и это тоже можно изменить. В некотором смысле можно считать, что никаких «трёх форм» первого параметра `bind` в действительности нет, есть только одна форма — тег окна, просто некоторые теги, включая имя самого окна, предопределены. Подробнее эти механизмы описаны в man-странице, посвящённой команде `bindtags`.

Второй параметр, задающий событие, тоже может принимать разные формы — а точнее, ровно три: без угловых скобок, с одиночными и с двойными угловыми скобками. Все три варианта в наших примерах приводились; вариант без скобок соответствует одиночному печатному символу (напомним, это не может быть пробел и символ `<`), в двойных скобках — имени виртуального события, которое нужно сгенерировать командой `event generate` (кстати, она может сгенерировать произвольное событие, а не только такое). Наиболее сложный вариант — с одиночными угловыми скобками. Здесь описание события состоит в общем случае из нескольких частей, разделяемых, как мы видели, символом тире. В середине этого великолепия находится *тип события* — мы таких видели два, `KeyPress` (нажатие клавиши на клавиатуре) и `ButtonPress` (нажатие кнопки мыши). Всего поддерживаемых типов достаточно много, приведём только несколько примеров: `KeyRelease` и `ButtonRelease` — *отпускание* соответственно клавиши и кнопки, `Motion` — движение мыши, `MouseWheel` — движение скроллингового колёсика мыши, `FocusIn` и `FocusOut` — получение и потеря фокуса ввода, `Configure` — изменение размеров или положения окна (и ещё зачем-то изменение толщины рамки), `Destroy` — уничтожение виджета и т. д.

В зависимости от типа события может потребоваться уточнить конкретику — например, какая клавиша клавиатуры или кнопка мыши имеется в виду. В документации эта часть описания события называется *detail*; она записывается через тире *справа* от типа события. Клавиши на клавиатуре, соответствующие буквам и цифрам, так и обозначаются буквами и цифрами, знаки препинания имеют имена вроде `comma` или `minus`, служебные клавиши обозначаются названиями наподобие `Enter`, `Escape`, `F7`, `Tab`, `Shift_L` (левый Shift), `Up` (стрелка вверх) и т. д. Полный список распознаваемых имён приведён в man-странице `keysyms` (напомним, что нужно использовать секцию `3tk`). При наличии любого из этих обозначений можно, как мы уже отмечали, опустить название типа события, т. е. слово `KeyPress`.

Для событий `ButtonPress` и `ButtonRelease` в роли параметра detail выступает номер кнопки мыши от `1` до `5`, причём левой кнопке соответствует `1`, правой — `3`, а номер `2` соответствует скроллинговому колёсику, используемому в роли кнопки[11].

Слева от имени типа события, опять же через тире, могут быть записаны *модификаторы*: например, `Control`, `Shift` и `Alt` означают, что должна быть нажата соответствующая специальная клавиша, `Double`, `Triple` и `Quadruple` — двойное, тройное и четырёхкратное повторение события (между прочим, это можно использовать не только для кнопок мыши). Существуют и другие модификаторы, подробности можно найти в документации на команду `bind`.

Ещё одна интересная возможность — задать в качестве второго параметра `bind` не одно событие, а несколько; тогда обработчик будет вызван, когда (если) последовательно произойдут все заданные события.

В роли третьего параметра выступает скрипт, который нужно исполнить, когда заданное событие случится. Здесь тоже есть определённая особенность: из скрипта можно получить доступ к информации о произошедшем событии. Делается это через обозначения с символом `%`. Например, для событий `ButtonPress` и `ButtonRelease` комбинация `%b` превратится в *номер кнопки* (имеет смысл, если при задании обработчика мы не указали номер кнопки, так что он будет вызываться для нажатия или отпускания любой кнопки мыши), `%k` обозначает код символа/клавиши для `KeyPress` и `KeyRelease`, `%x` и `%y` означают координаты курсора мыши (это работает для всех событий, связанных с мышью и клавиатурой) и т. п. За подробностями, как обычно, предложим обратиться к man-странице.

Сделаем ещё одно важное замечание. Одно событие может повлечь запуск нескольких обработчиков (например, они могут быть установлены на разные теги, но все эти теги окажутся в списке тегов одного виджета). Между прочим, сами виджеты тоже пользуются обработкой событий — например, кнопка нажимается при щелчке мышью, потому что для виджетов этого типа настроен соответствующий обработчик события `<ButtonPress-1>`. Если повесить собственный обработчик того же события, например, на конкретный экземпляр виджета `button`, будет выполняться и этот обработчик, и стандартный обработчик для кнопки (то есть кнопка всё ещё будет нажиматься). В теле обработчика можно использовать команду `break`; если это сделать, оставшиеся обработчики (этого же события) вызваны уже не будут. Например, если командой `break` завершить обработчик события `<ButtonPress-1>` для виджета типа `button`, то нажать эту кнопку с помощью щелчка мышкой у пользователя уже не получится.

Команда `continue` тоже может быть использована в теле обработчика события, она досрочно завершает выполнение данного конкретного обработчика, но другие обработчики, если они есть, после этого будут выполнены.

12.4.7. «Ресурсы» и русификация

Коль скоро дело дошло до создания интерфейса пользователя, рано или поздно возникнет вопрос, как *корректно* заставить вашу программу общаться с пользователем по-русски. Как мы неоднократно подчёркивали ранее, в тексте программы русским буквам не место[12]; в ходе изучения языка Си мы рассмотрели возможности библиотеки `gettext`, которая специально предназначена для хранения строк, переведённых с английского, во внешних файлах и подстановки их вместо английского оригинала во время выполнения программы (см. т. 2, §4.8.5). Аналогичные возможности есть и для Tcl — нынешние версии включают пакет `msgcat` (*message catalog*), предназначенный для тех же целей, что и библиотека `gettext`, но мы рассмотрим другой подход, поскольку он, с одной стороны, несколько проще, а с другой — служит поводом для поверхностного знакомства с так называемыми *ресурсами X Window*, которые могут оказаться полезны не только для русификации.

В системе X Window поддерживается механизм так называемых ***X-ресурсов*** (*X resources*), призванный облегчить пользователю, а также администратору многопользовательской системы изменение настроек оконных программ (в основном их внешнего вида). Загадочное слово «ресурс» на самом деле обозначает довольно примитивную сущность: это некое хитро сформированное имя (параметра, опции, можно называть это как угодно) и заданное для него значение в виде строки. Значением может, собственно говоря, быть едва ли не любая строка, тогда как имя состоит из нескольких идентификаторов, разделённых точками или звёздочками; идентификаторы состоят из латинских букв, цифр и знаков подчёркивания, при этом X Window отличает идентификаторы, начинающиеся с заглавной буквы, от тех, что начинаются со строчной; первые считаются *классами*, вторые — *именами* (приложений, виджетов или опций). Самый первый идентификатор в имени ресурса задаёт имя или класс приложения, последний идентификатор соответствует опции, а идентификаторы между ними указывают на конкретное окно или класс окон (когда этих идентификаторов больше одного, речь

[11] Да, на него можно надавить сверху до щелчка, в X Window это используется для вставки выделенного текста. Вы не знали?

[12] См. т. 1, сноску 19 на стр. 90 и далее §2.12.1, замечание на стр. 176; т. 2, §4.8.1.

идёт об иерархии вложенных окон). Использование звёздочки вместо точки означает, что на этом месте могло бы быть ещё несколько идентификаторов.

В файлах, описывающих X-ресурсы, значение отделяется от имени двоеточием и пробелом. Например, строка

```
Fig.msg_form.balloon_toggle.topShadowContrast: -40
```

означает, что в приложении, имеющем класс `Fig`, есть некое окно (скорее всего, диалоговое) с именем `msg_form`, в нём, в свою очередь, имеется окно (по-видимому, дочернее) с именем `balloon_toggle`, для которого актуальна опция с именем `topShadowContrast`, и значение этой опции следует установить равным `-40`; для того же приложения класса `Fig` строка

```
Fig*canvas.background: gray97
```

предписывает значение `gray97` для опции `background` всех виджетов `canvas`, которые могут быть ещё куда-то вложены, а строка

```
Fig*foreground: black
```

устанавливает `black` значением опции `foreground` вообще для всех виджетов из приложения `Fig`, для которых другое значение не было установлено более специфическим ресурсом.

Более подробную информацию об X-ресурсах читатель может почерпнуть из Интернета (например, [28]); опустив множество технических деталей, перейдём непосредственно к проблеме русификации программ, написанных на Tcl/Tk.

Для приложений на Tcl/Tk *имя приложения* (в том смысле, в котором оно используется в ресурсах) задаётся параметром командной строки `-name` интерпретатора `wish` (см. стр. 193), а имя класса приложения используется такое же, только первая буква приводится к верхнему регистру. Так, если мы укажем в начале нашего скрипта в строчке, запускающей `wish`, параметр `-name myprog`, то именем приложения будет слово `myprog`, а его классом — слово `Myprog`. Интересующая нас опция — `text`; например, если мы решим создать простую метку так, как мы это уже не раз делали:

```
label .lab -relief ridge
```

— то текст для такой метки можно будет задать с помощью ресурсной строки

```
Myprog.lab.text: This is a text for the label
```

Если же у нас возникнет сложная иерархия виджетов, например, такая:

```
toplevel .dlg1
frame .dlg1.buttongrp
button .dlg1.buttongrp.b1 -command { dlg1_b1_pressed }
```

— то и имя ресурса придётся использовать соответствующее:

```
Myprog.dlg1.buttongrp.b1.text: Button One
```

Вполне может оказаться полезной и возможность использовать звёздочки в именах ресурсов. Например, если у нас в программе задействовано много всевозможных диалоговых окон, но во всех есть кнопка Close, и мы всегда называем её именно `close`, а не как-то иначе, то задать для неё текст можно так:

```
Myprog*Button*close.text: Close
```

Кстати, как можно догадаться, `text` — не единственная опция, которую можно задавать через ресурсы. Например, если мы хотим, чтобы *все* метки в программе имели выступающую рамку толщиной в три пикселя, можно предусмотреть вот такие ресурсные строки:

```
Myprog*Label*relief: ridge
Myprog*Label*borderWidth: 3
```

По правде говоря, идея выносить это в файл ресурсов несколько сомнительна, но ресурс можно задать прямо в программе с помощью команды `option add`:

```
option add "Myprog*Label*relief" ridge
option add "Myprog*Label*borderWidth" 3
```

Добавим к сказанному, что текстовый файл, содержащий ресурсные строки, можно прочитать командой `option readfile`, и читатель, скорее всего, догадается, куда мы клоним. Итак, открываем на редактирование файл, в котором у нас будут ресурсные строки; обычно имя такого файла снабжают суффиксом `.rc` (от слова *resource*), например `myprog.rc`. Убираем из текста нашего скрипта все вхождения опции `-text`, заменяя каждое из них на соответствующую ресурсную строку, помещаемую в файл `.rc`. Далее создаём копию ресурсного файла под другим именем, что-нибудь вроде `myprog_ru.rc`, и в нём заменяем все англоязычные значения русским переводом. Остаётся только в начале скрипта — *до* формирования виджетов — решить (например, исходя из параметров командной строки или переменных окружения), на каком языке мы собираемся работать в этот раз, и в зависимости от принятого решения подгрузить с помощью `option readfile` файл с английскими строками или с русскими.

Здесь, к сожалению, есть одна хитрость. С тех пор, как Tcl официально стал поддерживать интернационализацию, в качестве его внутренней кодировки принята utf8. Ресурсные файлы, считываемые с помощью `option readfile`, не предусматривают ни задания их кодировки в самом файле, ни возможности указать кодировку в программе при считывании, так что интерпретатор всегда неявно подразумевает, что ресурсный файл сформирован именно в utf8. Если там используются только символы ASCII, этот факт никак себя не проявляет, поскольку представление символов ASCII в utf8 совпадает с обычным ASCII, но русские буквы в ASCII, как мы знаем, не входят.

Если ваша системная кодировка — utf8 (в современных условиях практически для всех дистрибутивов Linux это по умолчанию именно так, а хорошо ли это — вопрос иной), то никаких проблем не возникнет; если же вы используете другую кодировку[13], то ресурсный файл с русскими строками придётся перекодировать в utf8, например, с помощью программы `iconv`.

12.5. Стратегии выполнения как парадигмы

12.5.1. И всё же — что такое скрипт

Теперь, когда мы знакомы с двумя характерными командно-скриптовыми языками, самое время задать себе вопрос, что же такое, собственно говоря, ***скриптинг***. Вопрос этот в действительности сложнее, чем выглядит на первый взгляд. Вспомним, например, что в §12.1 мы выделили три основные характерные черты командно-скриптовых языков: интерпретируемость, представление любой информации строками и доступность средств управления внешними командами. Признаем теперь, что один из самых популярных языков, предназначенных для скриптинга — JavaScript[14] — из этих трёх свойств проявляет разве что интерпретируемость. Модель данных в нём похожа скорее на знакомую нам по диалектам Лиспа и языку Пролог — типизированные данные и нетипизированные переменные, ну а работа с внешними программами вообще не предусмотрена. Известны применения для скриптинга разнообразных вариантов Лиспа и Scheme, которые тоже, как мы помним, не укладываются в наш образ командно-скриптового языка. С другой стороны, в современных условиях часто можно видеть применение командно-скриптовых языков для программ, которые никак нельзя назвать скриптами (и это происходит намного чаще, чем хотелось бы).

Коль скоро и скрипты пишут не только на скриптовых языках, и сами эти языки используют не только для скриптов, становится ясно, что вопрос о границах понятия «скрипт» — это вопрос самостоятельный. Мы попытаемся на него ответить, и для этого сначала припомним разные случаи, когда программу называют скриптом.

Впервые с этим словом мы столкнулись в применении к файлам, содержащим команды Bourne Shell. Характерными для скриптинга здесь можно назвать две особенности: во-первых, файлы, написанные на языке командного интерпретатора, в большинстве случаев (хотя и не всегда) пишутся, чтобы *автоматизировать рутинные последовательности действий*, которые в

[13] Например, автор книги работает с koi8r и не собирается её ни на что менять.

[14] Этот язык изначально предназначался и по-прежнему чаще всего используется для встраивания скриптов в веб-страницы; такие скрипты выполняются *в браузере*. С точки зрения безопасности эта практика не просто недопустима, здесь чудовищна сама идея выполнения *программы*, написанной на алгоритмически полном языке и загруженной из непонятного места Всемирной Паутины, не только без разрешения, но и без ведома пользователя. Если ваш браузер тормозит, подвисает, глючит или вовсе падает — можете быть уверены, что дело тут в очередном скрипте, написанном на JS; вообще client-side sctipting — одна из самых катастрофических идей, применяемых в современной IT-индустрии. Однако речь сейчас не об этом. Для нашего обсуждения важен тот факт, что «программы», написанные на JS и исполняющиеся в браузере, — это, несомненно, именно скрипты.

противном случае пришлось бы выполнять человеку — вручную; во-вторых, такие файлы *запускают другие программы*, или, иначе говоря, *управляют* программами.

Читатель наверняка сталкивался или хотя бы слышал о скриптах, исполнение которых происходит в программах для конечного пользователя — таких как офисные приложения; сами скрипты встраиваются в файлы документов, электронных таблиц, презентаций и т. п. (их официальное название — макросы, но сути оно не меняет). Может показаться, что и с пресловутым JavaScript в браузере наблюдается приблизительно аналогичная ситуация, но это не так — или, во всяком случае, не совсем так. Конечно, здесь есть много общего: *документы*, от которых логично было бы ждать *содержания*, внезапно обретают собственное *поведение* — или, точнее, диктуют поведение той программе, которая пытается их обрабатывать. Разница тут в том, с какой целью это делается. Скрипты-макросы в текстовых процессорах и электронных таблицах обычно создаются для уже знакомой нам *автоматизации* рутинных процедур: есть некие последовательности действий, которые автор документа вынужден часто повторять, и, чтобы сэкономить силы, он выполнение этих последовательностей возлагает на компьютер. Со скриптами на веб-страничках ситуация совершенно иная: «автор документа» (разработчик сайта) делает их уж точно не для себя, навязывание некоего поведения вашему браузеру становится самостоятельной целью, и во многих случаях это совсем не такое поведение, которым пользователь будет доволен.

Любители старых текстовых онлайн-игр, так называемых MUDов (*multi-user dungeon*), а равно и те, кто застал IRC образца середины девяностых, могут припомнить ещё одно значение слова «скрипт»: клиентские программы для того и другого содержали встроенные интерпретаторы, позволявшие автоматизировать реакцию на определённые происходящие события — например, в играх автоматически употреблять лечебные предметы при снижении уровня здоровья до определённой отметки, а в чатах — автоматически здороваться с одними знакомыми и автоматически выкидывать из канала других.

Встроенные интерпретаторы часто применяются для сложной настройки программ и целых программных систем в случаях, когда обычные конфигурационные файлы с проблемой уже не справляются; это явление широко распространено в областях, связанных с управлением предприятиями — от бухгалтерского учёта до программной поддержки бизнес-процедур. До определённой степени такое использование интерпретируемых языков тоже можно отнести к скриптингу, хотя в какой-то момент, совершенно не поддающийся формализации, происходит своего рода переход количества в качество: внезапно обнаруживается, что программный код, написанный на языке встроенного интерпретатора, значительно превзошёл по объёму и сложности ту программу, внутри которой он выполняется. В этом случае о *скриптинге* речи уже не идёт.

Отметим заодно, что встроенный интерпретатор часто реализует либо полностью свой собственный язык программирования (как, например, широко известные бухгалтерские программы), либо *диалект* какого-то существующего языка (примеры этому — неоднократно упоминавшиеся выше AutoLisp и ELisp); в обоих случаях правомерно говорить о ***предметно-ориентированном*** языке программирования, в противоположность ***языкам общего назначения***. Соответствующие английские термины — *domain-specific [programming] language (DSL)* и *general purpose language*.

В ходе рассуждения о предметно-ориентированных языках возникает соблазн заявить, что таким окажется вообще любой встроенный язык, в противном случае от него не будет никакой пользы — ведь чтобы управлять программой, в которую он встроен, язык должен включать специфические возможности, предоставляющие программисту доступ, собственно говоря, к «органам управления», которые, конечно же, будут свои для каждого приложения, включающего в себя встроенный интерпретатор. Проблема здесь, как можно заметить, терминологическая. В самом деле, если мы встроим в наше приложение существующий интерпретатор Tcl (который как раз для этого и предназначен, если на то пошло) и снабдим его проблемно-ориентированными командами, правомерно ли будет говорить о новом *языке*? Ответов на этот вопрос два, «да» и «нет», и в пользу каждого из них можно найти весьма убедительные аргументы.

К обсуждению предметно-ориентированных языков программирования мы вернёмся в §12.5.7, пока отметим только, что языки для скриптинга представляют собой их частный случай: скриптинг — это, несомненно, предметная область, или, точнее, это довольно большое семейство предметных областей, но есть ведь и другие предметные области.

Можно с достаточной степенью уверенности выделить одну общую черту всех случаев скриптинга: **скрипты — это программы, которые управляют другими программами**. Управление может осуществляться как изнутри — средствами интерпретатора, встроенного в программу, так и снаружи, обычно из интерпретатора, по совместительству работающего командной оболочкой — путём подачи команд, запускающих и останавливающих программы; но всегда, во всех случаях скриптовой язык выступает во вспомогательной роли, а написанная на нём программа (которая как раз и называется скриптом) не имела бы никакого смысла без тех программ, которыми она управляет. Попросту говоря, *скриптовые программы не самодостаточны*, они лишь дополнение к чему-то другому.

Косвенно это подтверждается и тем, что программы для встроенных интерпретаторов называют скриптами лишь до тех пор, пока они не сравняются по сложности со своей средой исполнения: после этого момента во «вспомогательном» положении оказывается, наоборот, интерпретатор с его окружением — он лишь реализует возможности, которыми пользуется основная программа. Таким же точно образом ядро операционной системы при всей его блистательной мощи — лишь своего рода обслуга для пользовательских задач. Конечно, пользовательские программы не смогли бы функционировать без ядра ОС, а интерпретируемая программа никак не может существовать без исполняющего её интерпретатора, но дело тут не в этом. Вопрос скорее в том, *кто тут для кого предназначен*. Скрипты всегда предназначены для программ, которыми они управляют; они могут сделать управляемые ими программы немного лучше, но основная ценность полученного инструмента заключена именно в основных программах, а не в написанных для них скриптах. С ядром ОС и пользовательскими программами ситуация очевидным образом противоположна: ядро предназначено для выполнения пользовательских программ, а не наоборот. С интерпретатором, встроенным в некую основную программу, дело может обстоять, как мы видим, и так, и эдак, но *скриптами* интерпретируемые программы называются лишь до тех пор, пока они сохраняют вспомогательную роль.

12.5.2. Дихотомия Оустерхаута и её противники

Итак, определяющее свойство скриптов состоит в том, что они не самодостаточны. Логично было бы ожидать, что не будут самодостаточными и языки программирования, предназначенные для скриптинга, но в этом плане явно что-то когда-то пошло не так: в современной реальности программные разработки, использующие тот же Tcl, или Lua, или Perl, или Python, или Ruby, или JavaScript — пусть и не командно-скриптовой в обычном смысле, но исходно предназначенный именно для скриптинга — в роли *основного* (и даже единственного) языка реализации почему-то никого не удивляют.

Прежде чем обсуждать это довольно странное явление и пытаться выяснить его причины, расскажем одну историю, основными участниками которой стали весьма известные люди — Ричард Столлман, лидер проекта GNU, автор лицензии GNU GPL и по совместительству автор Emacs'a, и Джон Оустерхаут — напомним, что это автор Tcl.

В сентябре 1994 года Ричард Столлман опубликовал в новостной группе `comp.lang.tcl` письмо, имевшее довольно бескомпромиссный заголовок: «Why you should not use Tcl» (*почему не следует использовать Tcl*) [29]. Ссылаясь на свой опыт создания Emacs, Столлман заявил, что язык для расширений не должен быть просто языком для расширений, он должен подходить для создания полноценных программ, так как люди хотят их создавать; между тем, Tcl для серьёзного программирования не годится, поскольку в нём нет массивов, отсутствуют средства для создания связных списков, а вместо чисел — подделка, которая работает, но неизбежно оказывается медленной. Далее в сообщении утверждалось, что «своеобразный» (в оригинале — *peculiar*) синтаксис Tcl может быть привлекателен для «хакеров»[15] из-за его простоты, но для большинства пользователей этот синтаксис «странен», и если Tcl станет «стандартным скриптовым языком», пользователи будут проклинать его долгие годы, так же, как Фортран, MS-DOS, синтаксис командной оболочки Unix и другие стандарты де-факто. В заключение Столлман призвал

[15]Напомним, что изначальный смысл слова «хакер» не имеет никакого отношения к компьютерным взломщикам и прочим «плохим людям»; Столлман в данном случае использовал это слово скорее в качестве положительной характеристики, ведь к хакерам он относит и себя самого.

тех, кто использует Tcl из-за наличия в нём Tk, перейти на язык Scheme и использовать интерпретатор STk.

Сообщение Столлмана содержало ещё некую декларацию о намерениях проекта GNU создать два языка программирования, похожих по семантике, но различающихся синтаксисом; один должен был быть лиспоподобным (для «хакеров»), а другой — более традиционным (для всех остальных). За прошедшие с тех пор четверть века ничего подобного так и не появилось, хотя попытки вроде бы были. Что касается STk, то последний релиз этого интерпретатора вышел в 1999 году.

Публикация такого воззвания в телеконференции, большинство участников которой — активные пользователи Tcl, естественным образом вызвала самый, мягко говоря, *живейший* отклик [30], так что Джон Оустерхаут, отвечая Столлману в той же конференции тремя днями позже, был вынужден начать своё послание [31] с призыва сохранять технический характер обсуждения и не переходить на личности.

> Я думаю, претензии Столлмана к Tcl, — писал далее Оустерхаут, — могут проистекать во многом из одного аспекта в дизайне Tcl, которого он либо не понимает, либо не согласен с ним. Этот [аспект] — утверждение, что в крупной программной системе следует использовать **два** языка: один, такой как Си или Си++, для обработки сложных внутренних структур данных, где ключевую роль играет производительность, и другой, такой как Tcl, для написания небольших скриптов, которые соединяют вместе куски [кода на] Си и используются для расширений. Для скриптов на Tcl лёгкость изучения, лёгкость программирования и лёгкость соединения[16] более важны, чем производительность или средства для сложных структур данных и алгоритмов. Мне кажется, что эти две области программирования столь различны, что для одного языка было бы трудно хорошо работать сразу в обеих. Например, не слишком много людей использует Си (или даже Лисп) в качестве командного языка, несмотря на то, что оба этих языка вполне подходят для более низкоуровневого программирования.
>
> Поэтому я построил Tcl так, чтобы было по-настоящему просто прыгнуть вниз[17] к Си или Си++, когда вы встречаете задачи, для которых более осмысленно [использовать] низкоуровневые языки. То есть Tcl'ю не нужно решать все проблемы в мире. Столлман, судя по всему, предпочитает подход, когда один язык используется для всего и вся, но я не знаю успешного случая [применения] этого подхода. Даже Emacs внутри использует существенные объёмы Си, разве нет?
>
> Я не создавал Tcl для построения огромных программ из десятков или сотен тысяч Tcl'ных строк, и для меня было изрядным сюрпризом, что люди использовали его для огромных программ. Что для меня ещё более удивительно — это что в некоторых случаях полученные приложения оказываются вменяемы[18]. Разумеется, это не то, для чего я предназначал язык, но результаты оказались не столь плохи, как я мог бы ожидать.

Оустерхаут далее пишет, что в Tcl вполне могут быть недостатки, и некоторые из них устранимы, другие нет; по-видимому, можно придумать язык лучше, чем Tcl, обладающий при этом всеми его достоинствами, и даже он сам, автор языка, сделал бы некоторые вещи иначе, если бы снова пришлось начать сначала. «Но является ли двухъязыковой подход правильным? — продолжает он. — Я считаю, что да, хотя [некоторые] разумные люди могут не согласиться». В заключение Оустерхаут отметил, что Scheme (и вообще любой диалект Лиспа), вероятнее всего, не является «самым правильным», ведь слишком много людей за последние 30 лет «проголосовали ногами», перейдя к использованию других языков, и призвал всех, кто не верит в Tcl, создавать свои «правильные» языки, с тем чтобы публика в итоге рассудила, какой из языков правильнее, попросту используя или не используя их.

Позже Оустерхаут опубликовал статью «Scripting: Higher Level Programming for the 21st Century» [32], в которой противопоставил «языки системного программирования» и «скриптовые языки»; это противопоставление получило название ***дихотомии Оустерхаута***. Довольно любопытны предложенные в статье величины для сравнения языков программирования: «уровень» языка (в смысле высокоуровневых или низкоуровневых языков) автор предложил измерять как среднее количество машинных команд на один оператор, а в дополнение к этому ввёл ещё «степень типизированности», согласно которой языки могут быть нетипизированными (один конец шкалы), строго типизированными (противоположный конец), а также типизированными слабее и сильнее. Отложив эти характеристики по двум осям координатной плоскости, Оустерхаут получил диаграмму (см. рис. 12.3), на которой показаны области языков системного программирования и скриптовых языков, и эти области отчётливо друг от друга отделены.

Конечно, с тех пор (а статья официально вышла в 1998 году, но была написана ещё раньше) мир несколько поменялся. Если в диаграмму из статьи Оустерхаута добавить, например, язык Haskell с его чрезвычайно высоким уровнем в сочетании не просто со строгой типизацией, но с задействованной математической теорией типов, он займёт пустующее место в правом верхнем углу и явно окажется за пределами обеих введённых Оустерхаутом областей. Означает ли это приговор дихотомии Оустерхаута? Совершенно не факт, особенно если учесть, насколько реже Haskell используется в сравнении с другими языками. Впрочем, Haskell — далеко не единственный повод критиковать Оустерхаута с его дихотомией.

Довольно забавным выглядит тот факт, что основную массу оппонентов дихотомии Оустерхаута составляют сторонники таких своеобразных языков программирования, как Python, Ruby, Perl, реже — Java или Lua. Основная претензия критиков состоит в том, что многие интерпретируемые языки (якобы) прекрасно подходят на роль языков общего назначения и совершенно не обязаны быть *скриптовыми*. Можно встретить аналогичные утверждения в несколько иной формулировке: на языках, исходно позиционировавшихся как скриптовые, тоже можно прекрасно программировать, и даже Tcl — детище самого Оустерхаута — тому пример; вспомните, ведь автор Tcl ещё в 1994 году, отвечая Столлману в `comp.lang.tcl`, высказывал по этому поводу своё удивление.

К *интерпретации* в смысле более широком, нежели скриптинг, мы вернёмся в следующем параграфе, а пока попробуем разобраться, кто и почему пытается использовать в роли языков общего назначения языки, исходно для этого не предназначенные. Такие случаи встречаются гораздо чаще, чем можно было бы ожидать. Язык Perl изначально предназначался для преобразования текстов, то есть был проблемно-ориентированным, но его превратили в язык для произвольного скриптинга, и практически сразу появилось заметное число разработок, написанных на этом языке и никакого отношения к скриптингу не имеющих. Про то, как автор Tcl удивлялся применению этого языка для обычного (нескриптового) программирования, мы уже упоминали. JavaScript, исходно предназначавшийся «в довесок» к HTML для выполнения в браузере, в какой-то момент начали активно использовать для создания программ на стороне сервера; встречаются программы на этом языке, вообще не имеющие никакого отношения к Web. Использование JavaScript в роли языка общего назначения имеет своих сторонников (см. напр. [33]). Можно встретить людей, пишущих обычные (нескриптовые) программы на Lua, на Ruby, даже на PHP. Отдельная история с языком Python — некоторые его сторонники утверждают, что он вообще не является и никогда не был скриптовым.

Понять, почему люди превращают скриптовые языки в языки общего назначения, судя по всему, не так уж сложно. Языки такого рода обычно просты в изучении и, как следствие, доступны непрофессионалам, барьер вхождения здесь намного ниже, чем для «серьёзных» языков вроде Си или даже Паскаля. Человек, не являющийся программистом и не собирающийся таковым становиться, может научиться писать простенькие скрипты из чистого любопытства, или же скриптинг может потребоваться ему для решения какой-то задачи, но, так или иначе, людей, пишущих свою первую в жизни программу на каком-то из скриптовых языков, намного больше, чем тех, кто начинает с настоящих языков общего назначения.

Далее срабатывает сразу два известных общезначимых принципа, один из которых — *закон золотого молотка* — мы уже упоминали (см. стр. 190): когда в руках молоток, все окружающие предметы кажутся гвоздями. Второй принцип, обычно называемый ***синдромом утёнка***, состоит в том, что первый встреченный объект того или иного рода (в данном случае — первый освоенный язык программирования) всегда кажется более правильным, чем любые другие, и чтобы это ощущение переломить, требуются значитель-

[16] В оригинале *ease of glue-ing*.

[17] В оригинале *drop down to*.

[18] В оригинале *manageable*, но в этом контексте прямой перевод (слово *управляемы*) не подходит по смыслу.

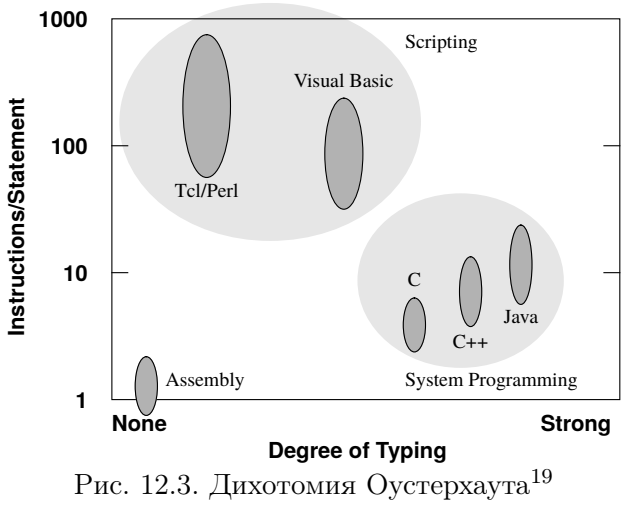

Рис. 12.3. Дихотомия Оустерхаута[19]

ные усилия. Закон золотого молотка побуждает новичка (хотя, к сожалению, далеко не любого) применять только что освоенный инструмент не только для задачи, для которой он осваивался, но и для других задач, к которым он кажется хотя бы отдалённо подходящим; в это же самое время синдром утёнка мешает заметить, что для всех этих задач существуют другие пути решения. В результате человек решает некую задачу на каком-нибудь PHP отнюдь не потому, что PHP для этой задачи хорошо подходит. Просто этому человеку, с одной стороны, PHP уже известен, а с другой — вставшую задачу на PHP решить *как-нибудь* возможно. При этом разница между *каким-нибудь* решением и решением *грамотным* может оказаться недоступна восприятию не только самого автора «какого-нибудь» решения, но и более опытных программистов.

12.5.3. И всё же что такое «интерпретация»?

В предисловии к этой части книги мы обещали рассмотреть интерпретацию и компиляцию под, возможно, неожиданным углом — а именно, как *парадигмы программирования*. Почему-то мало кто задумывается о влиянии выбора стратегии исполнения на мышление программиста, хотя такое влияние, несомненно, есть; но чтобы обсуждать его, нам придётся для начала определить, чтó же такое, собственно, интерпретация, а что такое компиляция.

Вопрос этот не так прост, как может показаться. Граница между интерпретируемым и компилируемым исполнением давно утратила чёткость; элементы интерпретации проникают в компилируемое окружение и наоборот. Многие популярные языки программирования, такие как C# и Java, традиционно компилируются, но результатом компиляции становится не машинный код, а некоторое промежуточное представление, предназначенное для выполнения (т. е. *интерпретации*) виртуальной машиной. С другой стороны, современные интерпретаторы в большинстве случаев сначала переводят исходный текст программы в некоторое внутреннее представление, которое затем исполняют. Отличия между таким компилятором и таким интерпретатором оказываются довольно эфемерными: в первом случае промежуточное представление сохраняется во внешнем файле, а виртуальная машина реализована отдельно от транслятора, во втором случае виртуальная машина интегрирована с транслятором, что делает файл промежуточного представления необязательным.

Припомним теперь некоторые известные нам сведения о Лиспе. С одной стороны, мы знаем (см. стр. 118), что во время исполнения программы в памяти должен находиться весь интерпретатор (или, если угодно, весь код реализации Лиспа). Больше того, на примере трёх рассмотренных нами реализаций Common Lisp мы убедились, что *откомпилировать* программу на Лиспе и получить самодостаточный исполняемый файл если и возможно, то смысла в этом никакого нет, поскольку этот файл будет (неизбежно) содержать всю реализацию и окажется громоздким до неприличия, в особенности когда речь идёт о SBCL (см. стр. 111).

При этом создатели SBCL утверждают, что их реализация — это компилятор, а на людей, говорящих, что Лисп — язык интерпретируемый, следует «не обращать внимания» (см. §11.1.7). Терминологическую неразбериху подобные заявления усугубляют ещё больше. Можно даже столкнуться с утверждением, что вообще любое исполнение является интерпретирующим, поскольку исполнение машинного кода центральным процессором есть не что иное, как интерпретация, причём, если вспомнить о существовании в процессорах микрокода, такое утверждение оказывается не столь далёким от истины. Услышав подобное, можно в большинстве случаев ожидать, что сейчас нас в очередной раз будут убеждать в *несущественности* выбора модели исполнения и, в частности, всенепременнейшим образом заявят, что программы на том или ином языке, предполагающем полную или частичную интерпретацию, выполняются «почти столь же эффективно, как на Си» (что уже вызывает недоумение — ведь очевидно, что *плохо написанная* программа на Си запросто может «перетормозить» кого угодно, но из этого ровным счётом ничего не следует), а если даже и нет, то «люди дороже компьютеров» — последнее следует понимать в том смысле, что скорость *написания* программ (экономия времени программиста) важнее, чем скорость их работы.

Любителям подобных рассуждений стоит сразу же возразить, что об *эффективности* их вообще-то никто не спрашивал; в конце концов, скорость исполнения кода как таковая во многих случаях в самом деле оказывается фактором не столь важным, а *видимое* время работы программы довольно часто состоит в основном из ожидания — и если это так, эффективность по времени исполнения (вот то есть именно *выполнения кода процессором*, когда он выполняется, а не стоит столбом в очередной блокировке) — очевидным образом дело десятое. Но, услышав, что эффективность нас не особенно беспокоит, наши оппоненты часто изображают наивно-удивлённый взгляд и вопрошают, а чтó же, в таком случае, нас может не устроить в интерпретации.

Для ответа на этот вопрос напомним неоднократно подчёркивавшееся ранее свойство некоторых языков, в том числе Лиспа, Scheme и большинства командно-скриптовых: для выполнения программы, написанной на любом из них, необходимо присутствие программы-транслятора, как бы мы этот транслятор ни называли; в частности, создатели SBCL сколько угодно могут именовать своё детище компилятором, но ни связки `eval`/`intern`/`read`, ни обязательности наличия самого SBCL *во время выполнения программы* (в противоположность времени её написания) это не отменяет. Подчеркнём, что речь здесь идёт об *интерпретаторе языка программирования*, то есть о программном инструменте, которым пользуется программист в своей работе, и который спроектирован и написан во многом с ориентацией на программиста, а не на конечного пользователя.

Это обстоятельство, как мы увидим чуть позже, оказывается ключевым при сравнении стратегий исполнения, так что мы возьмём на вооружение простейший критерий, позволяющий разделить интерпретацию и компиляцию. **Если для выполнения программы, написанной на каком-то языке программирования, нужно её *исходный текст* подать на вход другой программе, то речь идёт об интерпретируемом исполнении, а программа, исполняющая этот исходный текст, должна называться интерпретатором** вне всякой зависимости от того, как именно внутри неё реализовано исполнение программного кода, поступающего на вход. **Если же программа может быть оттранслирована в исполняемый файл, для выполнения которого присутствие транслятора не требуется, речь следует вести о компиляции**.

К этому же самому критерию можно подойти с другой стороны. Если программист не является единственным пользователем собственной программы, то, по-видимому, следует предполагать, что система, или, если угодно, *окружение*, в котором программа будет *выполняться*, в общем случае отличается от окружения, в котором её *писали*. Ключевым для разделения стратегий выполнения можно считать вопрос, *нужны ли инструменты программиста для выполнения написанной программы*, или даже *предполагает ли программа доступность инструментов программиста во время выполнения*. Положительный ответ соответствует интерпретируемому исполнению, отрицательный — компилируемому. В частности, упомянутое выше утверждение, что микрокод в процессоре делает интерпретируемым вообще любое исполнение, мы с помощью нашего критерия отсекли, ведь микрокод никоим образом не относится к инструментам программиста.

Сразу же придётся признать, что такой критерий тоже оказывается не вполне чётким, поскольку для выполнения программы практически никогда не требуются *все* инструменты, используемые программистом — уж во всяком случае вам вряд ли потребуется для выполнения программы именно тот редактор текстов, в котором программист её набрал.

В истории существовали и такие системы программирования, в которых программу можно было запустить только изнутри системы, а сама система либо включала

[19] Диаграмма из оригинальной статьи Оустерхаута [32].

в себя текстовый редактор для работы с кодом программы, либо, хуже того, вообще не была рассчитана на применение редакторов текстов, а вместо этого позволяла формировать и изменять текст программы каким-то другим способом; работу с таким интерпретатором Бейсика мы обсуждали на стр. 11. Хочется надеяться, что их время прошло.

Со средствами отладки ситуация, как мы могли убедиться, несколько хуже. Очевидно, что пользователю эти инструменты полезны примерно как рыбе зонтик; даже если пользователь внезапно попадётся умеющий программировать, лезть в потроха чужой программы ему, скорее всего, не захочется. Тем не менее от тех же приснопамятных реализаций Common Lisp отладчик никак не оторвать, и стоит большого труда даже просто не дать интерпретатору в этот отладчик вывалиться (в том числе при совершенно корректных действиях пользователя вроде нажатия `Ctrl-C`).

Можно, конечно, считать определяющим компонентом системы программирования именно транслятор, то есть ту программу, которая непосредственно анализирует исходный текст — но тогда получится, что языки, транслируемые в промежуточное представление, которое затем интерпретируется (ту же Джаву), следует отнести к компилируемым — несмотря на то, что для выполнения программы в этом случае требуется *интерпретатор промежуточного представления*, часто называемый ***виртуальной машиной***. Как мы вскоре сможем убедиться, такой способ выполнения программы обладает большинством важных особенностей интерпретации, причём как её достоинств, так и недостатков, и в контексте нашего обсуждения парадигм будет правильнее считать его интерпретацией — ну, скажем, ***частичной***.

Следует, с другой стороны, учитывать, что *хоть какие-то* компоненты системы программирования в каком-то виде присутствуют во время выполнения программы почти всегда: на сей раз путаницу в картину вносят *библиотеки*. Несомненно, библиотеки — это инструмент программиста, конечные пользователи в большинстве своём не представляют, что это такое. В современных условиях почти всегда (если не предпринять целенаправленных мер) исполняемый файл, полученный с помощью компилятора Си или Си++, будет требовать наличия ***динамических библиотек***[20], что, кстати говоря, часто порождает существенные проблемы — непереносимость исполняемых файлов, конфликты версий, избыточный расход оперативной памяти и т. д. — которые программистская публика почему-то предпочитает игнорировать. Даже если у нас есть статические версии всех нужных библиотек (увы, это далеко не всегда так) и мы воспользуемся при сборке ключом `-static`, библиотеки (правда, на сей раз — только те модули, которые требуются нашей программе) всё равно окажутся присутствующими во время исполнения, просто они будут включены в состав исполняемого файла. При всём при этом язык не повернётся назвать интерпретацией исполнение программ, написанных на Си и откомпилированных обычным для этого языка путём.

Путаницы добавляет ещё и тот факт, что на некоторых платформах — в частности, в системах линейки Windows, а также в большинстве систем для мобильных устройств — интерпретаторы тех или иных виртуальных машин присутствуют абсолютно всегда и воспринимаются как часть операционной системы[21]: для Windows это .NET, для Android — виртуальная машина, исполняющая байт-код языка Джава (в старых версиях — Dalvik, в более новых — ART, который знаменит тем, что «на ходу» транслирует байт-код в машинные инструкции процессора). Компонент, гарантированно доступный на избранной платформе, считать частью системы программирования невозможно. Довершает дело ещё и тот факт, что ядра систем семейства Windows всегда спрятаны за динамической библиотекой, их системные вызовы нигде не документированы, а библиотека, реализующая Windows API, *обязательна* для выполнения *любой* программы и никакой статической сборки, естественно, не подразумевает.

Попробуем предложить выход из терминологического лабиринта. Очевидно, что чёткой границы между компиляцией и интерпретацией мы обозначить не можем, из чего следует, что исполнение может быть «более интерпретируемым» и «менее интерпретируемым». Теперь уже всё становится совсем просто: **стратегию исполнения программы следует считать тем более интерпретируемой, чем больше компонентов системы программирования (инструментов программиста) требуется для выполнения программы**. В этом смысле примеры программ на языке ассемблера, которые мы рассматривали в третьей части книги (в первом томе), следует считать полностью компилируемыми, поскольку для их выполнения не требуется *никаких* элементов системы программирования, получаемые исполняемые файлы абсолютно самодостаточны, никаких библиотек не содержат, а зависят при этом только от возможностей центрального процессора и ядра операционной системы. Практически такого же уровня — полного отсутствия даже намёков на интерпретацию — мы достигли, написав на Си программу без использования стандартной библиотеки (см. т. 2, §4.12), но там нам несколько повезло — в программе не потребовалось ничего из тех возможностей языка Си (самого языка, не библиотеки!), которые компилятор `gcc` реализует в библиотеке `libgcc`.

12.5.4. Интерпретация как парадигма

Во время исполнения программы, если таковое является полностью интерпретируемым, доступны как транслятор (интерпретатор), так и исходный текст программы, и эту доступность было бы трудно проигнорировать; одна из особенностей инженерного мышления состоит в том, что относительно любой открывающейся возможности в голову сами собой приходят разнообразные способы её применения к решению практических задач. Одно такое применение нам уже встречалось — это примитив `EVAL`, позволяющий исполнить произвольный фрагмент кода, сформированного во время исполнения; но возможности, открывающиеся в силу доступности как интерпретатора, так и самой программы, этим отнюдь не исчерпываются.

Представляется очевидным, что интерпретируемая программа *управляет* интерпретатором: в самом деле, ведь интерпретатор действует в соответствии с тем, что написано в программе, в буквальном смысле *выполняя* указания, из которых программа, собственно говоря, *состоит*. Тем не менее возможности интерпретатора, доступные выполняющейся программе, могут быть несколько ограниченными — просто потому, что в интерпретаторе что-то не реализовано, либо (что в контексте нашего обсуждения намного интереснее) создателям интерпретатора тот или иной аспект его возможностей просто *не пришло в голову* сделать программно управляемым — или даже наоборот, авторы интерпретатора могли намеренно сделать те или иные его аспекты недоступными выполняемой программе.

Прекрасным примером этого может послужить интерпретатор Hopeless, который мы изучали в §11.5; как мы помним, этот интерпретатор обрабатывает ряд директив, позволяющих объявлять и описывать новые объекты, в том числе функции и типы, просматривать существующие объекты и даже редактировать их код, и, в довершение всего, поддерживает директиву `write`, с помощью которой можно выдать в поток вывода некий потенциально бесконечный список, причём выдавать его по мере вычисления его элементов, не дожидаясь, пока он будет сформирован целиком (чего может вообще никогда не произойти). Но при этом ни одна из директив, доступных на верхнем уровне (будь то в сеансе интерактивной работы или в файле скрипта), не может встретиться в теле функции — и, как следствие, *работающая программа* не может сделать ничего такого, для чего требуются все эти директивы; их применение — прерогатива программиста.

Недоступность программе директив верхнего уровня в случае Hopeless — скорее следствие авторского замысла, имевшего целью сохранить функциональную чистоту языка Хоуп; но в большинстве случаев ограничения такого рода, не имея чёткого и понятного (желательно — понятного даже идиотам) объяснения, держатся недолго либо вообще не закладываются в реализацию интерпретатора с самого начала. В результате программа во время исполнения получает доступ ко всему, что изначально предназначено в помощь программисту и вроде бы должно использоваться не во время исполнения, а во время создания программы. Получив полную власть над своим собственным исполнителем, программа может анализировать сама себя, узнавать и обрабатывать (в виде простых строк) имена переменных, функций и других объектов, изменять все эти объекты (то есть свои собственные фрагменты, попросту — саму себя), может перенастраивать интерпретатор — вплоть до того, что он перестаёт понимать исходный текст той программы, которую сейчас выполняет, или начинает понимать его как-то не так.

Все эти возможности известны под общим названием «***рефлексия***». Само понятие рефлексии было, судя по всему, введено Брайаном Смитом в работе [34]. Смит описывает вычисление программы в виде связей между тремя *доменами*: синтаксическим доменом (исходный код), доменом внутреннего представления программы и доменом «реального мира». В терминах, введённых Смитом, язык программирования поддерживает *структурную рефлексию* (*structural reflection*), если он позволяет анализировать или изменять объекты внутреннего домена. Если же язык позволяет вмешаться в процесс вычисления (*нормализации* по Смиту), т. е. предоставляет прямой

[20] Динамические библиотеки неоднократно упоминались в предыдущих томах, см. т. 1, §3.7.5, т. 2, гл. 4.12, а в §12.3.2 мы их даже использовали.

[21] В расширенном смысле этого термина, то есть не как часть ядра, а как обязательный компонент обвески.

доступ к вычислительному контексту программы, речь идёт о *поведенческой рефлексии* (*behavioral reflection*).

Для обеспечения поддержки рефлексии исходный текст программы и/или её внутреннее представление, а также разнообразные аспекты интерпретатора и его работы должны быть отображены некими объектами, доступными выполняющейся программе, с возможностью выполнять над этими объектами некоторый набор операций. Это часто обозначают термином «***реификация***», пришедшим из философии.

Чтобы понять, о чём идёт речь, можно припомнить кое-что из наших сведений о Лиспе. Как мы помним, в этом языке *объект функции* — такой объект, который мы можем применить к списку аргументов — всегда представляет собой ***замыкание***, включающее как тело функции, так и значения свободных переменных, использующихся в нём. Естественно, это свойство функциональных объектов очень легко обнаружить, мы рассматривали целый ряд примеров, показывающих, что лисп-вычислитель работает именно так; но при этом замыкание как таковое предстаёт в виде «чёрного ящика», печатное представление которого не допускает (в отличие от большинства других типов S-выражений) обратного прочтения функцией `read`. «Влезть внутрь» объекта замыкания нам не позволяют, мы не можем ни узнать, ни тем более модифицировать хранящееся в нём тело функции, нам не доступен ни в каком виде список законсервированных свободных переменных и их значений; обнаружить свойства замыкания можно только опосредованно, демонстрируя, как эти замыкания работают. Собственно говоря, именно так — по косвенным признакам — и исследуются чёрные ящики, если говорить об исконном смысле этого термина из области эпистемологии.

Теперь представим себе, что мы изменили тот или иной интерпретатор Лиспа (или вообще написали новый с нуля) так, что внутреннее устройство замыкания стало открыто для работы с ним — например, в нашем интерпретаторе появились встроенные функции, позволяющие извлечь и модифицировать тело замыкания (уже существующего), получить список свободных переменных, узнать и изменить значение, связанное в данном замыкании с каждой из них. Ничего невозможного в создании такого интерпретатора, вообще говоря, нет, хотя жизнь реализаторов станет несколько труднее[22], и тогда можно будет сказать, что внутренности замыканий подверглись *реификации* или *были реифицированы*.

В этом плане проще всего обстоят дела с исходным текстом программы, если он доступен и выполняется именно в виде текста (как в случае Tcl) — для его реификации не нужно изобретать ничего нового, работать с текстовыми строками практически в любом языке программирования можно изначально, надо только предусмотреть какие-то примитивы для доступа к тексту программы и для замены фрагментов программы новыми версиями; обычно такой замене подвергаются подпрограммы целиком, но здесь возможны самые разнообразные варианты. Если же интерпретатор переводит программу в некое внутреннее представление и выполняет уже его (а большинство интерпретаторов именно так и поступают), реификация текста программы потребует введения специальных функций, работающих с этим внутренним представлением, и, скорее всего, специальных объектов, соответствующих самому представлению.

В интерпретаторах Лиспа, которые мы рассматривали, внутреннее устройство замыканий недоступно (то есть не реифицировано), а вот, скажем, имена символов реифицированы в любой существующей версии Лиспа: мы можем получить имя заданного символа в виде строки и найти (или создать) символ по его строковому имени. Напомним, что в §11.1.12 мы обсуждали эту возможность, указали, откуда она взялась и почему в современных условиях она кажется нелепой. На этом примере видно, что **реификация — это отнюдь не всегда хорошо**.

Пока речь идёт о структурной рефлексии, всё выглядит довольно просто, её проявления встречаются не только в интерпретируемых, но и в частично компилируемых языках, таких как Джава. Иногда зачатки рефлексии можно найти даже в программах на Си — например, при использовании динамически связываемых библиотек приходится оперировать *именами* функций.

Всё становится намного интереснее, если рассматривать рефлексию поведенческую. До определённой степени проявлением поведенческой рефлексии можно считать ***континуации***, знакомые нам по Scheme (см. §11.2.4) — они, как можно догадаться, *реифицируют* состояние вычислителя; но сторонники рефлексии на этом останавливаться не собираются и обсуждают возможность реифицировать непосредственно структуры данных, с помощью которых интерпретатор выполняет программу. Брайан Смит в своей работе отмечает, что в таком случае изменения, вносимые в структуры данных интерпретатора *программой*, могут войти в конфликт с изменениями, которые вносит в них же сам интерпретатор в ходе своей обычной работы; такой конфликт он называет «интроспективным наложением». Бороться с интроспективными наложениями предлагается путём построения «башни интерпретаторов» (*рефлективной башни*, в оригинале *reflective tower*), в которой нижний интерпретатор выполняет пользовательскую программу, следующий интерпретатор выполняет предыдущий интерпретатор и так далее, причём *потенциально* — до бесконечности. Авторы статьи [35] на полном серьёзе анализируют проблему интроспективного наложения, описанную Смитом, и объясняют, как с этим можно справиться *без* построения потенциально бесконечной башни интерпретаторов. Чего никто из них не делает — так это не задаётся вопросом, зачем всё это нужно.

Программы, меняющие во время своей работы не только сами себя, но и язык, на котором они написаны, а также всевозможные одиозные построения вроде рефлективных башен — это, к счастью, скорее всё же вотчина абстрактных теоретиков, и можно надеяться, что практического применения столь страшные конструкции никогда не найдут; впрочем, если вернуться в область практического программирования и возможностей рефлексии, *реально применявшихся и применяющихся*, мы обнаружим целый ряд «методик», при близком рассмотрении вызывающих едва ли не такой же священный ужас.

Начнём с явления, в своё время получившего английское название *guerilla patching* — в буквальном переводе «партизанское пришивание заплаток» (если угодно, *партизанская латка*), но ключевым тут будет слово «партизанский», а *patching* с английского обычно не переводят, так и оставляя откровенно сленговую словоформу «патчинг». Состоит сие явление примерно вот в чём: программа *во время исполнения* по-тихому («по-партизански») модифицирует сама себя, так что часть подпрограмм начинает работать не так, как написано в исходном тексте. После этого программа радостно продолжает выполняться.

Разобраться в функционировании программы после таких изменений может оказаться весьма затруднительно: мы можем «точно знать», что делает та или иная подпрограмма или подсистема, и, обладая этим «точным знанием», долго искать, почему вся программа работает не так, как мы от неё ожидаем; при этом нам так и не придёт в голову посмотреть, что же всё-таки происходит при вызове некой хорошо нам знакомой функции — ведь мы же *знаем*, как она работает; между тем на самом деле её давным-давно подменили, просто не в исходном тексте, а в памяти уже работающей программы, и ошибка, которую мы безуспешно ищем, кроется как раз там.

Со временем слово *guerilla* из-за созвучия со словом *gorilla* превратилось в *monkey*, а сам термин, ныне звучащий как *monkey patching*, стал обозначать явление более узкое: когда программа использует некую (внешнюю) библиотеку, для чего подгружает её в свою память, но прежде чем начать к ней обращаться, вносит в неё коррективы, чтобы приспособить библиотеку под свои нужды. Впрочем, разобраться, что происходит в такой программе, будет ничуть не проще, ведь функционирование библиотеки теперь отличается от описанного в документации. А самое интересное начинается, когда программу на языке, допускающем monkey patching, пишут несколько человек, и одна подсистема модифицирует некую библиотеку, а другая подсистема, тоже использующая ту же самую библиотеку, ни о каких модификациях не знает и ожидает от библиотечных функций их исходного поведения.

Как ни трудно в это поверить, очень многие люди уверены, что monkey patching — это мощный и полезный инструмент, который можно и нужно использовать в повседневной практике; существуют даже книги, в которых подробно, с иллюстрациями и примерами, разбираются разнообразные случаи практического применения этого подхода. Держитесь от этих людей подальше.

Отметим, что ***частичная компиляция*** — перевод в промежуточное представление, которое затем выполняется специальным интерпретатором (именно так работают Джава и C#) — никоим образом не исключает рефлексию (и в Джаве, и в C# рефлексия так или иначе присутствует), но по крайней мере сдерживает бесконтрольное применение метапрограммирова-

[22] Например, *имён* свободных переменных в замыкании обычно просто нет, а тело функции представлено неким списком, в котором можно не обнаружить символов, соответствующих свободным переменным и формальным параметрам функции; так что сделать всё это доступным через функции будет не так просто — но возможно.

ния: виртуальная машина, выполняющая байт-код или другое промежуточное представление, всё-таки не включает в себя транслятор, так что состряпать во время исполнения кусок исходного текста и тут же внедрить его в работающую программу не получится.

Вообще валить всё в одну кучу, не отличая аспекты программы, видимые пользователю, от всего того, с чем имеет дело программист — привычка довольно вредная; последствия такого подхода мы наблюдали при изучении Common Lisp, где ошибки, очевидно относящиеся к «косякам» программиста, в том числе обнаруживаемые при трансляции, обрабатываются точно так же, как ошибки ввода-вывода, которых программист избежать заведомо не может.

12.5.5. Зависимости и самодостаточность

Ранее мы уже обсуждали тот странный факт, что сторонники интерпретируемых языков склонны рассматривать в качестве недостатка интерпретации эффективность (как правило, по времени выполнения) и более ничего. Конечно, практика показывает, что эффективность интерпретируемого исполнения, несмотря на все усилия его сторонников, в подавляющем большинстве случаев оставляет желать много лучшего, причём даже когда внутри интерпретатора применяется перевод в машинный код (как в SBCL), ведь сам по себе перевод в машинный код тоже происходит не мгновенно. Для программ, имеющих существенный объём исходного кода, интерпретирующее исполнение может оказаться непригодным из-за *длительного времени запуска программы*, ведь фактически трансляцию программы приходится при каждом запуске производить заново. Естественно, современные интерпретаторы стараются эту проблему так или иначе решить, переводя программу во внутреннее представление по частям по мере надобности либо сохраняя сгенерированное внутреннее представление (полностью или частично) для использования во время последующих запусков; существования проблемы всё это не отменяет. Но дело, как ни странно, не в этом. **Возможное снижение эффективности по времени представляет собой существенный, но отнюдь не единственный и даже не главный недостаток интерпретации.**

Чтобы понять, в чём в действительности состоит загвоздка, вспомним весьма распространённое в современной IT-индустрии явление — ***зависимости*** (англ. *dependencies*). В общем случае, когда в области техники идёт речь о зависимости одного от другого, это обычно означает, что в отсутствие этого «другого» первое не может функционировать и/или быть полезно; к примеру, кварцевые наручные часы зависят от элементов питания, а в отсутствие таковых (например, если вдруг элементы питания нужного типа перестанут производить) становятся бесполезны, тогда как механические часы не зависят вообще ни от чего — можно сказать, что они ***самодостаточны***.

Соответствующий английский термин — *self-contained*. Часто можно встретить другой перевод — «автономные»; проблемы с этим вариантом начинаются при попытке перевести его обратно на английский, поскольку английское *autonomous* — это совершенно не то же самое, что *self-contained*. Отметим, что *self-contained* — это прямой перевод слова «самодостаточный».

В применении к компьютерным программам *формально* самодостаточность не достигается *никогда*, ведь любой программе для работы нужен хотя бы компьютер, а подавляющему большинству программ — ещё и операционная система (как минимум, её ядро). Поэтому обычно неявно предполагают, что зависимость программы от компьютера и ядра операционной системы можно не проговаривать как нечто само собой разумеющееся; самодостаточной считают программу, которая не имеет *иных* зависимостей.

Для компилируемых программ (и вообще таких, которые передаются на компьютер конечного пользователя в форме, отличной от исходного текста) обычно выделяют ***зависимости времени сборки***, к которым относят компилятор, библиотеки, утилиту автоматизированной сборки (`make` или её аналог, если, конечно, для данной программы таковая применяется) и вообще всё, что необходимо для успешного получения из исходных текстов исполняемого файла (в более общем случае — для перевода программы в то представление, в котором она будет выполняться на машине пользователя). При этом всё то, что нужно для *работы* уже собранной (переведённой в исполняемое представление) программы, называют зависимостями ***времени исполнения***. В их число могут входить динамические библиотеки, содержащие часть используемых программой библиотечных функций, а также, например, файлы с данными, нужными для работы программы, поставляющиеся вместе с ней, но по какой-то причине не включённые в саму программу (например, это могут быть файлы с иконками и другими изображениями, которые программа использует в своём графическом интерфейсе, всевозможные табличные данные и т. п.). Наконец, программа может в своей работе опираться на возможности других программ — например, если некоторые подзадачи она решает, запуская внешние программы; в этом случае в список зависимостей одной программы могут входить другие программы.

Зависимости могут образовывать длинные цепочки, которые, налагаясь друг на друга, превращаются в совершенно хаотический ориентированный граф, всё время норовящий выйти из-под контроля. Если совсем честно, контролировать граф зависимостей становится практически невозможно почти сразу, даже когда в нём жалкий десяток элементов, во всяком случае, если речь идёт о ручном отслеживании зависимостей. Эту проблему, что вполне естественно, пытаются решить путём автоматизации. В большинстве дистрибутивов того же Linux используется тот или иной *пакетный менеджер*, а каждая устанавливаемая в системе программа оформляется в виде пакета, содержащего, помимо всех нужных программе файлов, ещё информацию о том, от каких других пакетов она зависит.

Итог здесь немного предсказуем, но почему-то не всем понятна его принципиальная неотвратимость. Попробуем привлечь аналогию. Представьте себе некую судоверфь, инженеры которой решили, что обеспечивать полную герметичность швов корпуса, расположенных ниже ватерлинии, сложно, дорого и неэффективно, правильнее будет установить в трюме мощные помпы, которые прекрасно справятся со сравнительно небольшим количеством воды, проникающей через швы. Тем более что помпы, откачивающие воду из трюма, в любом случае нужны — небольшие повреждения корпуса судна полностью исключить невозможно. Поначалу помпы действительно справляются, автоматически включаясь по мере необходимости, так что трюм остаётся практически сухим. Поскольку проблема негерметичных швов теперь решена, на их герметизацию вообще перестают обращать внимание. Вскоре на всех новых судах, сходящих с верфи, швы текут так, что помпы справляться перестают, но эта небольшая досадная сложность даже не заслуживает серьёзного обсуждения: просто поставим помпы помощнее, и дело сделано. Дальше, разумеется, найдётся очередной технический гений, который заявит, что швы вообще не нужно проваривать, намного дешевле и быстрее применить болтовые соединения, причём без всяких уплотнителей — зачем они нужны? Вода, разумеется, через такие швы будет хлестать со страшной силой, но все же прекрасно знают, что помпы справятся и с этим. Ну а когда построенные по таким принципам суда начинают тонуть из-за отказавших помп, инженеры нашей верфи презрительно заявляют, что перед выходом в плаванье нужно проверять оборудование, особенно помпы, так что те, кто утонул, виноваты сами; на любые намёки, что помпы вообще-то расходуют энергию, следует ответ, что энергетическая установка судна в любом случае должна быть достаточно мощной, ведь на судне есть ещё и двигатель, и он помощнее этих несчастных помп; помпы, естественно, приходится держать включёнными, когда судно стоит в порту или на рейде, но и тут всё понятно: минимизируйте простои вашего судна, всё равно ведь долгие стоянки — это невыгодно; очевидная недолговечность корпуса, построенного по принципу дырявого таза, тоже не должна никого смущать, корпус ведь в любом случае нужно как можно чаще профилактически осматривать и проводить регламентные работы. Ну и самое главное: на любого, кто осмелится предложить всё-таки вернуться к герметичным сварным швам, немедленно обрушивается поток обвинений в ретроградстве и профессиональной несостоятельности, при этом рефреном звучит утверждение вроде «эти ваши герметичные швы устарели лет на пятьдесят».

Заявление, что корпус корабля можно изначально делать дырявым, коль скоро помпы в трюме имеют достаточную мощность — конечно же, совершенно нелепо. Осмелимся теперь предложить читателю провести небольшой эксперимент. Дайте прочитать предыдущий абзац случайно выбранному человеку, уточните, согласен ли он, что подобные действия инженеров абсолютно нелепы и невозможно даже представить себе ничего подобного в реальности, после чего попросите объяснить, *почему* ваш подопытный так считает. В подавляющем большинстве случаев вы получите нечленораздельное мычание и фразы вроде «да ну очевидно же». Если вам попадётся человек, который сможет дать вразумительный ответ, цените такого человека — это очень большая редкость.

Ну а сам правильный ответ тут прост как валенок и заключён в следующем принципе: **если мы знаем, как решить проблему — это не повод**

её создавать. Решение проблемы никогда не бывает бесплатным, хотя очень часто таким кажется; и если проблему, *имеющую решение*, начинают рассматривать как *несуществующую*, ситуация рано или поздно, но совершенно неизбежно выйдет из-под контроля.

Чтобы понять, что́ тут имеется в виду в применении к зависимостям между пакетами, приведём пример из реальной жизни. Автор этих строк за несколько дней до написания в буквальном смысле *этих строк* попытался на одном из своих компьютеров установить программу `atril`, предназначенную для просмотра PDF-файлов. Автоматически вычислив все косвенные зависимости, пакетный менеджер предложил скачать и поставить 166 (!) пакетов общим «весом» почти полгигабайта, в число которых вошли, например, программа `parted` (для редактирования разделов жёсткого диска), пачка программных библиотек для обработки звука и видео и ещё много чего, имеющего столь же прямое отношение к просмотру файлов в формате PDF. Самое интересное, что без *некоторых* из пакетов, которые перечислил пакетный менеджер, программа `atril` действительно попросту не запустится — её исполняемый файл требует присутствия нескольких десятков динамических библиотек; но бóльшая часть предложенного к установке — лишь следствие чьего-то нежелания применять головной мозг при упаковке пакетов.

Конечно, здесь прежде всего следует говорить о безответственности людей, создающих пакеты с программами и прописывающих в них зависимости; возможно, при применении должной осмотрительности с подобными странными эффектами до определённой степени удастся справиться. Но даже если все, кто занимаются пакетами в популярных дистрибутивах Linux, вдруг станут безупречно аккуратны, от многих прелестей графа зависимостей это не спасёт. Представьте себе несколько программ, зависящих от одной разделяемой библиотеки. В какой-то момент вам требуется ещё одна программа, зависящая от той же библиотеки, или более свежая версия одной из уже установленных программ; вы запускаете менеджер пакетов, и он вам заявляет, что нужная программа (или новая версия) действительно доступна, но требует более новой версии библиотеки. Чтобы обновить библиотеку, нужно обязательно обновить и все программы, которые от неё зависят. Некоторые из этих программ тоже не стояли на месте, так что могут потребовать новых версий *других* библиотек и тем самым потянуть за собой обновления ещё большего количества установленных программ, и, что самое интересное, совершенно не факт, что все они стали лучше; увы, *регрессии* — явление в области IT достаточно частое, к тому же разработчики зачастую откровенно плюют на пользователей, решив, что та или иная возможность «никому не нужна» и её можно выкинуть. Да и вообще, почему вы должны обновлять половину системы (и хорошо ещё, если не всю), когда вам потребовалась свежая версия одной несчастной программы?!

Больше того, создатели пакетов — отнюдь не боги и могут не всегда успевать за меняющимися версиями библиотек; может случиться и так, что одни программы, нужные вам, требуют некую библиотеку версии не ниже указанной, а другие программы, вам тоже совершенно необходимые, упакованы так, что с новыми версиями этой библиотеки работать не могут. И смеем вас заверить — всё это только начало истории, которую благодарные пользователи лет двадцать назад прозвали *dependency hell*[23]. Подчеркнём, что принцип «увяз коготок — всей птичке пропасть» работает здесь как нельзя лучше: как только где-то появляется граф зависимостей между компонентами системы и средства для автоматического их отслеживания, этот самый dependency hell становится неизбежен.

Ругать тех, кто пакует пакеты — так называемых *мейнтейнеров* (англ. *maintainters*) — в действительности изначально не вполне правильно, им самим неплохо достаётся, на сей раз от разработчиков программ. Зачастую, чтобы собрать какую-нибудь более-менее развесистую программу из архива исходников, приходится обеспечить наличие в системе десятка сторонних библиотек, каждая из которых тоже может от чего-нибудь зависеть. В итоге на *подготовку* к сборке какого-нибудь программного, с позволения сказать, *изделия* — на то, чтобы найти в Интернете, скачать и откомпилировать все требуемые библиотеки — может уйти времени раз в двести больше, чем на саму сборку; когда же речь идёт не просто о сборке, а о создании инсталляционного пакета для того или иного дистрибутива Linux, все требуемые библиотеки тоже приходится «опакечивать», при этом на каждый новый пакет тратятся силы и время.

Что бы ни говорили апологеты систем с автоматическим отслеживанием зависимостей, с точки зрения конечного пользователя *идеальной* будет программа, представленная одним исполняемым файлом, который достаточно скопировать на машину, после этого можно запустить — и он будет работать, а для деинсталляции потребуется только удалить его. Точно так же идеальная программа с точки зрения мейнтейнера — это программа, архив исходных текстов которой содержит всё, что нужно для её компиляции и сборки: распаковал, дал команду `make` — и всё собралось.

Если программа написана на одном из компилируемых языков, оба идеала — и пользовательский, и мейнтейнерский — вполне достижимы: в дерево исходников можно включить все используемые библиотеки (естественно, тоже в форме исходных текстов), а итоговый исполняемый файл скомпоновать в статическом режиме, так что он не будет при старте загружать никакие динамические библиотеки. Собственно говоря, при этом и архив исходных текстов, и исполняемый файл окажутся *самодостаточными*. Очевидно, что **в случае интерпретируемого исполнения самодостаточность принципиально недостижима**, ведь, согласно нашему пониманию интерпретации, для такого выполнения программы нужен как минимум интерпретатор.

Если на минутку представить себе «идеальный» интерпретатор — этакую компактную, статически собираемую и не зависящую ни от чего внешнего программу — можно даже предположить, что необходимость его наличия была бы не бог весть какой проблемой: ну пошлём мы пользователю не один исполняемый файл (нашей программы), а два — интерпретатор и исходник. Вот только в реальной жизни так, разумеется, не бывает, а корень проблемы кроется, как ни странно, в наших любимых *парадигмах*, то есть в том, как воспринимают действительность авторы интерпретаторов.

Для начала заметим, что трансляторы, в том числе, естественно, и интерпретаторы, *воспринимаются* прежде всего как инструмент программиста; добавим к этому, что, на чём бы ни был написан интерпретатор, пусть даже на чистом Си, те, кто его пишет, заведомо не являются противниками интерпретируемой модели исполнения — иначе они не стали бы придумывать очередной интерпретируемый язык и писать для него интерпретатор. Иначе говоря, эти люди не видят ничего плохого в том, что весь этот инструментарий должен будет оказаться на машине пользователя, чтобы программа, написанная на их прекрасном языке, могла выполниться, ну а где сам интерпретатор, там и (а что такого, в самом деле) всевозможные вспомогательные программы — например инсталляторы, которые сами умеют загружать из Интернета библиотеки, написанные на этом же интерпретируемом языке, когда очередной запускаемой (или устанавливаемой — это зависит от глубины интреграции инсталлятора в окружение) программе что-то такое потребовалось (welcome to dependency hell[24]). В итоге вокруг очередного интерпретатора вырастает целая, как это сейчас принято говорить, *экосистема*, позволяющая программистам поменьше думать. Сама по себе возможность лишний раз не задумываться о технических мелочах — прекрасна, ведь программисту и без того есть обо что сломать мозги, вот только в случае интерпретаторных экосистем расплачивается за это *пользователь*, вынужденный не только терпеть на своей машине всю эту экосистему целиком, но и зачастую *уметь с ней обращаться*.

На этом месте можно услышать возмущённые возгласы, смысл которых сводится к тому, что пользователя же никто не заставляет самого заниматься установкой и настройкой экосистемы — достаточно дать соответствующую команду пакетному менеджеру, и он всё как надо поставит. Так вот, во-первых, так обстоят дела только в системах на основе Linux, притом не во всех; во-вторых, всех проблем это не решает, и те, кто утверждают противоположное, попросту врут.

Начнём с того, что пакеты для всех мыслимых и немыслимых интерпретаторов имеются в готовом виде для десятка, от силы для двух десятков наиболее популярных дистрибутивов Linux; стоит сделать шаг в сторону от мейнстрима, и, вероятнее всего, нужный интерпретатор пользователю придётся собирать из исходников самостоятельно. Конечно, обычные («простые») лопоухие юзеры среди пользователей экзотических дистрибуций обычно не встречаются, как правило их предпочитают профессиональные сисадмины или программисты, квалификации которых вроде бы должно хватать на такую сборку; но ведь квалификация пользователя никоим образом не отменяет потраченного времени, усилий и нервов.

Автор книги здесь готов сослаться на свой собственный опыт. На серверных машинах он предпочитает использовать Openwall GNU/*/Linux — дистрибутив весьма и весьма малоизвестный. Так вот, в своё время вашему покорному слуге пришлось сде-

[23] Буквальный перевод — *ад зависимостей*.

[24] «добро пожаловать в ад зависимостей» (англ.).

лать выбор в пользу jabberd2, отказавшись от идеи использования ejabberd, поскольку последний написан на Эрланге, а собрать Эрланг под Openwall не удалось. С тех пор автор сих строк стал весьма разборчив в выборе программного обеспечения для серверных нужд. Из популярных интерпретаторов в дистрибуцию Openwall входит разве что Perl, для которого при этом доступны далеко не все популярные библиотеки, так что и с этим языком приходится соблюдать осторожность; ну а программы, написанные на Python, Ruby и других подобных языках приходится приравнивать к несуществующим.

Отметим далее, что Linux — не единственная система в мире, иногда приходится (увы) обеспечивать функционирование тех или иных программ на машинах с FreeBSD, где пакетного менеджера с автоматическим отслеживанием зависимостей просто нет, или в системах линейки Windows, где вообще нет ничего похожего на пакетный менеджер. Если некая проблема якобы решена в мире Linux, то это не значит, что она решена абсолютно везде.

Ну а настоящие приключения начинаются, когда в одной и той же экосистеме должны ужиться две (или ещё больше) масштабные программы, каждая из которых требует изрядного количества всяческих библиотек. Как показывает практика, возникновение конфликта версий здесь не более чем вопрос времени.

Отметим один довольно забавный факт, прекрасно иллюстрирующий действительное положение вещей с зависимостями. Толпы программистов, работающих на самых разных языках, десятилетиями хором убеждали публику и самих себя, что зависимости — это вообще не проблема, а если какие трудности и возникают, то они всегда легко решаются, и никому никогда зависимости в действительности ничем не мешают, а кому мешают — тот то ли должен повышать квалификацию, то ли просто дурак. В это же самое время системщики серьёзного уровня придумали решение, основанное на виртуализации операционной системы и позволяющее при желании загнать каждую программу со всеми её зависимостями в отдельное окружение («контейнер»), из которого не видно других контейнеров, работающих на той же самой машине, так что программистские изделия, работающие в разных контейнерах, никак не могут друг другу помешать. Наиболее популярное решение этого класса известно под названием Docker.

Конечно, распространять каждую программу в виде образа Docker-контейнера — это вряд ли идеальное решение, поскольку такой контейнер должен заключать в себе копию всего набора системного программного обеспечения, кроме разве что ядра ОС (оно обслуживает все контейнеры сразу). Объём каждого образа неизбежно оказывается достаточно значительным, чтобы его не получалось игнорировать. Но ведь мы же помним старую мантру — «люди дороже компьютеров»; в частности, разумеется, если из-за того, что кому-то *одному* лень немного подумать, *тысячи* других людей вынуждены покупать жёсткие диски бóльшей ёмкости — то это так и должно быть, а как же.

Между прочим, те самые люди, которые громче всех кричали, что, мол, зависимости — это вовсе никакая не проблема, с появлением Docker ухватились за него первыми и теперь предпочитают кричать, что Docker — это классно и правильно, поскольку с ним легко решаются те самые проблемы, которых якобы нет.

12.5.6. Миф о переносимости

Довольно часто можно встретить утверждение, что языки высокого уровня меньше зависят (или вообще не зависят) от особенностей машины и операционной системы, и, как следствие, чем выше уровень абстракций используемого языка, тем проще переносить программы на таком языке с одной платформы на другую. В теории всё вроде бы сходится, но практика, как это часто бывает, вносит свои поправки.

Проблема тут в том, что различные системы предоставляют в распоряжение пользовательской программы *разные* наборы базовых абстракций. Например, в системах семейства Unix процессы — это рабочая лошадка, которую можно и нужно гонять в хвост и в гриву, вызов `fork` в использовании прост, как отвёртка, и сравнительно дёшев; попытки сделать то же самое в системах линейки Windows сразу же отбивают всякое желание порождать дополнительные процессы. Вызов `select` под Windows есть, но работает только с сокетами. Ещё хуже обстоят дела с интерфейсом к графике: пожалуй, правильно будет сказать, что X Window по принципам построения интерфейса пользовательских программ изначально не имеет ничего общего с графическими подсистемами других платформ, не только Windows, но и основанных на Unix — таких как Mac OS X и Android; эти, впрочем, между собой тоже не слишком похожи. Различаются не только интерфейсы к возможностям платформ, но и сами возможности; так, в unix-системах можно ожидать, что у любой программы имеются стандартные потоки ввода-вывода, а в Windows ничего подобного исходно нет, хотя и возможно придумать некую эмуляцию.

Чтобы привести всё это великолепие «к общему знаменателю», создателям высокоуровневых инструментов приходится выдумывать свой собственный абстрактный мир, в котором будут существовать написанные с помощью их инструментов программы; эта искусственная реальность зачастую оказывается одинаково далека от всех существующих систем, так что программы в итоге на всех платформах работают одинаково плохо. Характерный пример такой искусственной реальности мы наблюдали, пытаясь справиться с интерпретаторами Common Lisp; добавим, что средства построения графических интерфейсов в спецификацию Common Lisp не входят, но и доступа к низкому уровню там тоже не предусмотрено, так что программы, написанные на Common Lisp, могут выполняться под Windows только в специально для этого построенной среде, включающей что-то вроде эмулятора терминала.

Между прочим, Common Lisp во всей своей кошмарной неуклюжести иллюстрирует ещё один момент: *всё предусмотреть невозможно*, и про какую-то из особенностей платформы создатели высокоуровневого инструмента могут просто не подумать. Так, Common Lisp использует свою (довольно своеобразную) модель ввода-вывода, в которую «не вписались» сокеты — и, значит, написать на нём какой-нибудь TCP-клиент или тем более сервер не получится. Чтобы преодолеть неадекватность выдуманных абстракций и всё-таки позволить программистам создавать нечто сколько-нибудь приемлемое, приходится так или иначе позволять программам добираться до низкоуровневых возможностей платформы, но такой доступ плохо ложится на высокоуровневые абстракции используемого инструмента, не говоря уже о том, что переносимость такие вещи убивают сразу же и наповал.

Наконец, сами высокоуровневые инструменты (например, интерпретаторы) — это довольно сложные программы, а при переносе между разнородными платформами приходится переписывать значительную часть их кода. Заставить такого монстра всегда (для любого пользовательского кода) одинаково вести себя на всех поддерживаемых платформах — задача не вполне реальная, то есть простенькие примеры обычно переносятся без проблем, но чем сложнее становится пользовательская программа, тем больше вероятность, что что-нибудь сломается при переносе её на другую платформу, хоть и с той же самой (вроде бы) системой программирования.

Как ни странно, опыт показывает, что добиться переносимости программы — если эту переносимость запланировать заранее, прежде чем начать писать код — проще всего на чистом Си, а если программа предполагает использование графического интерфейса пользователя — то на Си++: например, описанная ранее библиотека FLTK прекрасно работает под Windows, и для библиотек виджетов это вполне обычная ситуация. Дело в том, что при работе на этих языках у программиста есть доступ ко всем возможностям используемой платформы, а «привести к общему знаменателю» только те возможности, которые нужны в данной конкретной программе, многократно, в десятки раз проще, чем «все возможности всех платформ», как это вынуждены пытаться сделать создатели «переносимых» высокоуровневых интерпретаторов.

Автор книги на основании собственного опыта может однозначно заявить, что создать программу с графическим пользовательским интерфейсом, переносимую между Linux и Windows, намного проще на Си++ с использованием какой-нибудь библиотеки виджетов, нежели на супер-высокоуровневом Tcl/Tk, который вообще довольно сложно запустить под Windows. Если вам всё-таки придёт в голову это попробовать, засеките время, которое уйдёт у вас на инсталляцию Tcl под Windows, и осознайте, что если вы не передумаете программировать на Tcl/Tk с прицелом на переносимость, всем вашим пользователям, работающим под Windows, тоже придётся *это* проделать.

12.5.7. Когда интерпретация всё же допустима

Все языки программирования, которые мы рассматривали в этом томе, кроме Си++, в той или иной степени относятся к интерпретируемым; но если интерпретируемое исполнение (как вполне можно заключить, прочитав предыдущие параграфы) недопустимо, то зачем мы изучали все эти языки?

Попробуем ответить на этот вопрос. Во-первых, как говорилось в самом начале, альтернативные парадигмы программирования стоят того, чтобы их освоить, даже если они никогда потом не будут применяться на практике: практические навыки ценны, но эластичность мозга — бесценна. Во-вторых,

в ряде случаев интерпретируемая сущность того или иного языка не столь неизбежна, как кажется. Например, если выкинуть из той же Scheme примитивы `eval`, `read` и всевозможные проявления рефлексии вроде преобразования между объектом символа и его именем, то полученный язык будет вполне пригоден для компиляции и даже для статической сборки. Скорее всего, для достижения осмысленных результатов придётся перелопатить реализацию библиотеки функций, считающихся встроенными — так, чтобы они как можно меньше «зацепляли» одна другую и в исполняемый файл в итоге попадал код только тех функций, которые используются в программе; но и в этом ничего невозможного нет. Common Lisp в том виде, в котором он существует, сколько-нибудь приличной компиляции не допускает, но ведь это не единственный существующий (и тем более не единственный *возможный*) диалект Лиспа; с Прологом тоже наверняка можно что-нибудь придумать.

Наконец, последний аргумент в пользу изучения интерпретируемых языков состоит в том, что *интерпретация — это не всегда столь однозначно плохо*. Ранее мы в качестве основного недостатка интерпретации обсуждали необходимость наличия интерпретатора (и всей его обвески, которая может быть похлеще самого интерпретатора) на машине пользователя; но ведь это очевидным образом не имеет никакого значения, если *программист и пользователь — это одно и то же лицо*, а такие ситуации не столь редки. Бывает, что программа пишется ради трёх-четырёх запусков, а то и вовсе *одного*; программисту проще проделать эти считанные запуски самому, чем переносить программу на машину пользователя. Бывает и так, что автор программы с её помощью решает какую-то свою специфическую задачу, решает её так, как лично ему удобно, и полученная программа совершенно точно не предназначена для передачи кому бы то ни было — возможно, к примеру, что *научить* кого-то другого пользоваться этой программой настолько сложно, что сводит на нет потенциальные выгоды от её передачи. И что, спрашивается, в этой ситуации плохого в интерпретируемом исполнении? Очевидно, ничего.

Случай, когда пользователь и программист едины в одном лице, без проблем обобщается на ситуацию, когда рассматриваемое «лицо» — не физическое, а юридическое, то есть когда программа пишется для внутреннего использования в организации, где работает её автор, и совершенно точно не будет передаваться за её пределы. Применяя интерпретацию, программисты могут перекладывать свои проблемы на пользователей и мейнтейнеров, но в этом нет ничего плохого, коль скоро и программистам, и пользователям, и мейнтейнерам отдельно взятой программы платит зарплату один и тот же работодатель и этого работодателя всё устраивает; например, у программиста зарплата может быть в десять раз выше, чем у мейнтейнеров, так что час, сэкономленный программистом, может обойтись в целый рабочий день мейнтейнера, и работодатель всё ещё останется в плюсе.

Ещё один очевидный случай — когда конечному пользователю поставляется некий закрытый программно-аппаратный комплекс, то есть это либо некий специализированный компьютер, либо даже компьютер общего назначения, но на который всё программное обеспечение установила организация-разработчик, и вмешательство конечного пользователя в устройство комплекса (в том числе установка и настройка программного обеспечения) изначально не предусмотрено. Здесь важно, что мейнтейнеры или кто там вместо них — те, кто устанавливают программу на компьютеры, на которых она будет работать — получают зарплату в той же организации, что и программисты, написавшие программу, а пользователю оказывается совершенно всё равно, он с внутренней кухней поставленного ему закрытого решения никогда и никак не сталкивается.

Ещё один важнейший случай, когда интерпретация не только допустима, но и, пожалуй, *обязательна* — это упоминавшиеся ранее ***предметно-ориентированные языки программирования***, создаваемые специально для решения задач некоторого узкого класса. Важным подмножеством таких классов задач оказывается всё тот же ***скриптинг***, небольшие программы, управляющие работой других программ — либо снаружи, как это делают командные оболочки вроде `bash`, либо изнутри, когда скрипт выполняется встроенным в основную программу интерпретатором. В §12.5.1 мы отметили, что программа, работающая во встроенном интерпретаторе, может превосходить по своей сложности ту, *в которую* этот интерпретатор встроен, или даже не превосходить, но иметь сравнимую сложность. В этом случае речи о скриптинге уже не идёт — но по-прежнему можно говорить о проблемно-ориентированном языке. Кроме того, можно заметить, что скрипты играют вспомогательную роль в работе управляемых ими программ, тогда как в случаях проблемно-ориентированных языков, не относящихся к области скриптинга, во вспомогательной роли оказывается скорее сам интерпретатор и та программа, в которую он встроен.

Так или иначе, во всех этих случаях мы имеем два, если можно так выразиться, *слоя* программ. Один слой написан программистами и не предполагает модификации силами конечного пользователя. Возможно, такая модификация и не исключается — если программа распространяется в исходных текстах; но, по крайней мере, штатная эксплуатация программы не предполагает, что её нужно менять. На втором слое располагаются программы, которые, напротив, *предназначены* к написанию и модификации конечным пользователем, либо, возможно, программистами, которых этот пользователь наймёт, если не сможет справиться сам. Программы этого слоя могут поставляться вместе с основными программами в предположении, что пользователь адаптирует их под свои нужды, либо пользователю может быть только предоставлена возможность написать их при возникновении соответствующей потребности; это в данном случае не так важно.

В §12.5.2 мы познакомились с ***дихотомией Оустерхаута***, которая сводится к тому, что в крупной системе требуется два языка — основной язык и язык для скриптинга. В работах Оустерхаута речь шла только о скриптинге, но более сложный случай проблемно-ориентированных языков отличается разве что объёмом и сложностью программ «второго слоя», что может в некоторых случаях потребовать бóльшего внимания к эффективности встроенного языка. Главный принцип здесь состоит в том, что применять один и тот же язык для программ, составляющих оба слоя, было бы, по-видимому, неправильно: пользователь и программист — это (в общем случае) *разные* люди, обладающие разной квалификацией, перед которыми стоят разные задачи и у которых разные потребности.

Встроенный язык, в отличие от основного, можно сделать намного проще для изучения и работы. Кроме того, интерпретатор встроенного языка (как программа) может быть многократно проще компилятора основного языка. Представьте себе основную программу, написанную на Си++, к которой пользователю нужно будет, в силу особенностей предметной области, дописывать некие управляющие процедуры. Теоретически можно было бы, наверное, предоставить пользователю штатную возможность включать в программу его собственные модули, написанные на том же Си++, путём пересборки всей программы или даже без таковой — пользовательские модули можно оформить как разделяемые библиотеки и загружать в основную программу прямо во время исполнения. Но если в такую программу встроить компактный интерпретатор (например, тот же Tcl), это избавит пользователя не только от перспективы изучать Си++ (согласитесь, Tcl в десятки, если не сотни раз проще), но и от необходимости иметь на машине компилятор Си++ со всеми его библиотеками и прочей инфраструктурой.

Опять же, встроенный интерпретатор — на то и встроенный, чтобы находиться внутри исполняемого файла основной программы. Рассуждений о самодостаточном идеале (см. стр. 210) никто не отменял, так что интерпретатор, помещённый внутрь основной программы, хорош уже тем, что программа, содержащая его, может быть самодостаточна; если бы пользователю нужно было писать управляющие процедуры на том же языке, на котором написана программа, то эта программа неизбежно зависела бы от системы программирования на этом языке.

В большинстве случаев встроенные интерпретаторы в достаточной степени примитивны, чтобы вокруг них не образовывалась пресловутая экосистема, но даже если она всё-таки возникнет, есть по крайней мере надежда, что она будет ориентирована на пользователя, а не на программиста — разработчика программы; это позволит соблюдать принцип разделения программистского и пользовательского окружения.

Такое разделение программы или программной системы на два слоя, первый из которых создан программистами на компилируемом языке и до какой-то степени может рассматриваться как «сама программа», а второй интерпретируется и предназначен для написания и/или модификации пользователем, в принципе, тоже представляет собой парадигму. От дихотомии Оустерхаута в чистом виде эту парадигму отличает чёткое противопоставление основного кода программы и кода, предназначенного для пользовательского вмешательства; первый пишется на основном языке проекта, второй — на языке, который поддерживается встроенным интерпретатором. Кроме того, Оустерхаут мог подразумевать, но в явном виде не упоминал обязательность компиляции для основного языка.

При разделении на «основной» код и «пользовательский» становится очевидным следующий принцип: **программы, выполняемые встроенным интерпретатором, не должны становиться частью основной программы**. Иначе говоря, возможности основной программы, написанной на компилируемом языке, не должны зависеть от наличия или отсутствия тех или иных частей интерпретируемого кода. Интерпретируемые программы, выполняемые встроенным интерпретатором (будь то скрипты или более сложные программы) могут использовать (и, разумеется, будут использовать) функции основной программы, но обратная ситуация жёстко недопустима. Если подобное всё же произошло, то функциональность интерпретируемого кода, на которую полагаются какие-то части вашей основной программы, должна быть переписана на основном (компилируемом) языке проекта, даже если это увеличит трудозатраты.

Это требование уходит корнями в особенности восприятия программы человеком (в данном случае, как ни странно, в основном пользователем, хотя и программистом тоже), то есть его причины сугубо парадигматические. В такой двухслойной архитектуре любые фрагменты интерпретируемого кода будут неизбежно восприниматься как предназначенные к модификации пользователем, и с таким восприятием вы ничего не сделаете, даже если напишете кучу грозных комментариев и предупреждений — на них никто не обратит внимания, поскольку они противоречат подсознательно ожидаемому. Если правильная работа кода основной программы зависит от некоего фрагмента интерпретируемого кода, написанного только так и никак иначе (т. е. его изменение требует модификации кода на основном языке для сохранения общей целостности), а этот фрагмент кто-то изменил и из-за этого всё сломалось — вас в такой ситуации, как говорится, не поймут.

Вполне возможно, что наше перечисление случаев, когда интерпретация допустима и даже, как в случае встроенных интерпретаторов, желательна, не является полным; скорее всего, существуют ситуации допустимой интерпретации, которые мы не рассмотрели. Одно можно сказать со всей определённостью. Если вы пишете программу, предполагающую распространение и использование открытым (неизвестным заранее) кругом людей, то применение для этой цели интерпретируемого языка в качестве основного (или единственного) — это демонстративное неуважение к вашим пользователям, граничащее с хамством; выбрав интерпретируемый язык, вы, возможно, решили какие-то свои проблемы, но ведь цена этого «решения» — создание проблем для мейнтейнеров и конечных пользователей. С таким же успехом вы можете написать в документации на первой странице что-нибудь вроде «мне наплевать на всех, кто будет это читать, и втройне наплевать на тех, кто мою программу станет использовать».

Даже если вы не планируете вашу программу никому передавать, но *не уверены*, что ваши планы не изменятся — правильнее будет писать её на компилируемом языке, допускающем статическую сборку.

12.5.8. (*) Размышления о чистой компиляции

«Расширению возможностей» при полной или частичной интерпретации посвящено огромное количество литературных источников, ту же рефлексию постоянно обсуждают и в учебниках, и во вроде бы серьёзных научных статьях. Противоположный подход, основанный на *устранении* элементов интерпретации, литература, как это ни странно, обходит молчанием. Авторы статьи [36], в число которых входит ваш покорный слуга, были изрядно удивлены тем, что им не удалось найти буквально ничего, ни одной научной работы в этом направлении. Возможно, такие работы существуют, но тщательный поиск силами трёх человек никаких результатов не дал.

Продолжая использовать наш базовый подход к компиляции как к разделению программистского и пользовательского, предположим, что максимально возможная («чистая») компиляция может считаться достигнутой, если обеспечена как минимум *возможность* того, чтобы выбираемые программистом способы и методы достижения целей никак не влияли на всё, что видит конечный пользователь программы.

Наиболее очевидное базовое требование к компиляции мы уже неоднократно формулировали: для выполнения программы не должно требоваться присутствие транслятора. Это требование можно усилить: во время исполнения программы не должно происходить никаких элементов её трансляции, то есть должно быть возможно установить программу на компьютер, где она будет исполняться, в полностью готовом к выполнению виде, не требующем никакой дополнительной подготовки.

Ранее мы наблюдали подход к синтезу исполняемого файла путём соединения значительной части компонентов интерпретатора с исходным текстом программы или тем или иным его внутренним представлением; такой подход формально удовлетворяет базовому требованию, но при этом ясно, что это в действительности не компиляция; требование отсутствия не просто самого транслятора, а любых элементов трансляции позволяет это обосновать.

Заметив, что библиотеки очевидным образом представляют собой часть системы программирования, причём многие из них (не только «стандартные») прилагаются к транслятору, можно довести два предыдущих уровня требований до определённого логического завершения, потребовав для начала, чтобы генерируемый исполняемый файл не зависел (или по крайней мере *мог* не зависеть) от наличия каких бы то ни было компонентов системы программирования, в том числе динамически подгружаемых библиотек; переходя к следующему уровню, придётся потребовать, чтобы и в сам исполняемый файл не включались никакие библиотеки кроме тех, которые в явном виде затребовал программист, а минимально допустимое множество таких библиотек было пустым. Удовлетворить такому требованию может лишь язык программирования, из которого исключены (вытеснены в библиотеку) любые возможности, требующие сколько-нибудь нетривиальной реализации на уровне машинного кода.

Сделаем теперь шаг в сторону. Очевидно, что «программистское» не ограничивается только программными компонентами, входящими в систему программирования. Имеется другой аспект, возможно, даже более важный: *путь решения задачи*, избираемый программистом. Очевидно, что одна и та же задача может быть решена бесконечным[25] количеством способов. Большая часть возможных способов окажется неэффективна, но всегда, даже если речь идёт о программе «`Hello, world`», можно без труда найти несколько решений, из которых невозможно выбрать какое-то одно «наилучшее». Продолжая настаивать на разделении «программистского» и «пользовательского», логично будет потребовать, чтобы то, что видит пользователь, никак не зависело от принятых программистом частных решений, как следует реализовать тот или иной аспект функциональности программы. Наиболее очевидное требование из этой области — чтобы видимое пользователю поведение программы никак не зависело от избранных программистом имён (идентификаторов), то есть чтобы поведение программы не могло измениться, если все (или только некоторые) из использованных в ней идентификаторов переименовать, не нарушая при этом уникальности имён. Формально такое свойство программы можно назвать ***устойчивостью к альфа-преобразованию***.

Отметим, что всем перечисленным требованиям удовлетворяют в наше время разве что языки ассемблеров. В частности, компиляторы языка Си обычно порождают код, зависящий от пусть и небольшого, но далеко не пустого множества библиотечных функций; так, машинный код, созданный компилятором `gcc`, может потребовать сборку с библиотекой `libgcc`, даже если мы в программе никаких библиотечных функций не использовали; например, перемножение 64-битных целых чисел на 32-битных процессорах этот компилятор реализует через вызов библиотечной функции. Кроме того, в документации к компилятору перечислены несколько функций, на наличие которых он полагается даже при явном указании режима создания самодостаточного кода (в число этих функций входит, например, `memcpy`).

Более того, язык Си, строго говоря, не обладает устойчивостью к альфа-преобразованиям ввиду наличия в макропроцессоре возможности превратить идентификатор в его имя (строковую константу). Например, в программе можно объявить такой макрос:

```
#define GETIDNAME(x) (#x)
```

и затем использовать следующий вызов функции `printf`:

```
int myvar;
   /* ... */
printf("%s\n", GETIDNAME(myvar));
```

Такой вызов выведет строку «`myvar`»; если переменную `myvar` в программе переименовать, то, очевидно, изменится и выдаваемая программой строка.

[25] Бесконечным даже с формальной точки зрения: если задача алгоритмически разрешима, то всегда существует счётно-бесконечное множество алгоритмов, попарно эквивалентных между собой, решающих эту задачу.

Впрочем, даже достижение уровня требований, которым удовлетворяют только языки ассемблеров, не обязывает нас остановиться. Следующее требование оказывается очевидным как с точки зрения эффективности исполнения, так и с точки зрения самодисциплины программиста, но, тем не менее, соблюдать его в современных условиях практически невозможно. Итак: *любые вычисления и преобразования, для выполнения которых достаточно данных во время компиляции, должно быть возможно выполнить во время компиляции.* Это можно сформулировать иначе: *программистский инструментарий не должен вынуждать программиста переносить на время исполнения такие вычисления, которые можно проделать во время компиляции.* Нарушением этого принципа будет, в частности, интерпретация форматных строк в функциях `printf`/`scanf`. Впрочем, чтобы *соблюсти* это требование, избавиться от форматных строк недостаточно; чтобы научить компилятор проводить во время компиляции действительно *любые* преобразования, для которых достаточно информации, нужны, например, функции специального вида, возможно, помеченные каким-то ключевым словом; при вызове такой функции от параметров, значения которых известны во время компиляции, компилятор должен пытаться функцию вычислить и заменить её вызов полученным значением. В Си такого нет, но такими возможностями обладает, например, язык D («Ди»).

В терминах содержания исполняемого файла можно сформулировать ещё два требования к компиляции. Во-первых, система программирования должна позволять создать такой исполняемый файл, чтобы никакими средствами анализа нельзя было определить, какие конкретно инструменты были задействованы для его создания. Это требование позволяет надеяться, что в исполняемом файле *действительно не останется ничего лишнего*, обусловленного только выбранными инструментами и методами решения задачи и не имеющего отношения к решённой задаче как таковой. Во-вторых, можно потребовать, чтобы система программирования позволяла сформировать *любой* исполняемый файл, корректный с точки зрения целевой платформы. Среди существующих инструментов такими свойствами обладают только ассемблеры. Исполняемый файл, созданный компилятором языка Си, позволяет определить, какой был использован компилятор, причём в большинстве случаев — с точностью до версии.

Модель трансляции программ, удовлетворяющую всем перечисленным требованиям, можно назвать *чистой компиляцией* или *чисто компилируемым исполнением.* Очевидно, что для практического применения этой модели потребуется новый язык программирования, поскольку существующие языки высокого уровня (даже язык Си) введённым требованиям заведомо не соответствуют, а предложение о более активном использовании ассемблеров можно всерьёз не рассматривать.

В этом плане очень интересен рассмотренный нами Си++, пример которого ясно показывает принципиальную возможность существования языка низкого уровня, в котором возможно построение сколь угодно высокоуровневых абстракций. Конечно, сам по себе Си++ в том виде, в котором он существует сейчас, после стандартизаторских вакханалий, и даже в том, в котором он был изначально предложен, на роль чисто компилируемого (в только что введённом смысле этого слова) не годится. Дело в том, что в него с самого начала была включена достаточно нетривиальная система виртуальных функций, которую усложняет реализация множественного наследования в общем виде, и к этому приходится добавить подсистему обработки исключений, которая тоже появилась раньше первого стандарта (хотя и не с самого начала). Кроме того, в Си++ — в сам язык — включены средства работы с динамической памятью, которые, естественно, зависят от библиотеки. Когда же в Си++ добавили ещё и идентификацию типов во время исполнения (RTTI), этот язык оказался безнадёжно далёк от соответствия принципам чистой компиляции.

Все эти особенности языка при должном приложении усилий, скорее всего, удастся вытеснить в библиотеки, но для этого должна быть осознана необходимость такого вытеснения. В целом, если учесть как негативный, так и положительный опыт языка Си++, можно создать новый язык, допускающий чистую компиляцию в терминах, введённых выше, но при этом позволяющий ввести сколь угодно сложные абстракции. Такой язык мог бы одинаково хорошо (при условии адекватного выбора используемых библиотек, различного в каждом конкретном случае) подходить для решения как системных, так и прикладных задач, то есть быть в полном смысле *универсальным* — конечно, если говорить о языках общего назначения; для скриптинга и проблемно-ориентированных языков всё же лучше подходит интерпретация.

Главное тут, пожалуй, вот что. Потребность в новом языке программирования, который заменил бы откровенно морально устаревший Си с его трудоёмкостью кодинга и изувеченный стандартизаторами Си++, назрела уже давно, и попытки удовлетворить эту потребность за последние годы предпринимались не раз и не два, но результаты получаются довольно сомнительными. Сформулированные принципы чистой компиляции вполне могут оказаться как раз тем, чего не хватает.

Приложение 5. Сравнение файлов и наложение изменений

При работе с текстами, особенно если это исходные тексты программ, часто возникает потребность быстро понять, чем друг от друга отличаются две версии одного текстового файла. Выявить такие различия позволяет программа `diff` — получив две версии одного файла и предполагая, что первая из них — «исходная», а вторая — «изменённая», эта программа покажет, какие строки изменились, а какие были вставлены или удалены. Рассмотрим для примера хорошо знакомую нам программу «Hello, world»:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Запишем этот текст в файл `hello.c`, создадим его копию под именем `hello2.c` и внесём в новый файл небольшое изменение — например, уберём восклицательный знак в печатаемой фразе; после этого попробуем применить к этим двум файлам программу `diff`:

```
avst@host:~/work$ diff hello.c hello2.c
5c5
<     printf("Hello, world!\n");
---
>     printf("Hello, world\n");
avst@host:~/work$
```

Странное «`5c5`» в начале выдачи означает, что строка №5 превратилась (*changed*) в строку №5 — в самом деле, мы ведь не изменили количество строк, только отредактировали одну из них, так что её номер не поменялся. Далее `diff` печатает фрагмент оригинального текста, предваряя каждую его строку символом «`<`», затем выдаёт разделитель «`---`» и печатает фрагмент в его изменённой версии, в этот раз используя символ «`>`». Если бы мы не редактировали существующую строку, а вставили новые, `diff` отразил бы этот факт буквой «`a`» (от слова *append*); удалению строк соответствует буква «`d`» (*delete*). Для примера вернём на место восклицательный знак, а после строки, выдающей «Hello, world», вставим ещё две, которые будут печатать фразы «Glad to meet you, world» и «Good bye, world». Выдача `diff` при этом будет выглядеть так:

```
avst@host:~/work$ diff hello.c hello2.c
5a6,7
>     printf("Glad to meet you, world!\n");
>     printf("Good bye, world!\n");
avst@host:~/work$
```

Загадочное «`5a6,7`» означает, что после пятой строки исходного файла были добавлены строки, которые в результирующем файле стали шестой и седьмой. Если поменять местами исходный файл и результирующий, мы создадим у программы `diff` впечатление, что строки, наоборот, были удалены; об этом она сообщит так:

```
avst@host:~/work$ diff hello2.c hello.c
6,7d5
<     printf("Glad to meet you, world!\n");
<     printf("Good bye, world!\n");
avst@host:~/work$
```

«`6,7d5`» следует читать как «строки с шестой по седьмую были удалены, а если бы их не удалили, они располагались бы в результирующем файле после его пятой строки». Такой формат выдачи программы `diff` называется «нормальным»; надо сказать, что это лишь один из многих форматов, поддерживаемых программой `diff`.

Другой любопытный формат программа использует при указании флага `-y`; при этом она организует вывод в две колонки, слева — текст оригинального файла, справа — текст изменённого; колонки отделяются друг от друга столбцом символов, обозначающих, что произошло с данной строкой: если строка не изменилась, программа печатает пробел, если строка была удалена (иначе говоря, она есть только в оригинальном файле) — печатается символ «`<`», для добавленной строки (имеющейся только в изменённом файле) печатается «`>`», для изменившейся строки — «`|`». По умолчанию выдача `diff` в этом режиме имеет ширину 130 позиций, что соответствует длине строки на так называемых «широкоформатных» принтерах, широко применявшихся вплоть до конца 1990-х годов. Вы можете специально для работы с `diff -y` «распахнуть» окошко терминала на нужную ширину, либо, если ширина текста вашей программы это позволяет, указать программе `diff`, какой ширины выдачу вы хотели бы видеть, с помощью параметра `-W`; например, «`-W 80`» «загонит» `diff` в рамки стандартной ширины экрана.

Для примера снова возьмём программу «Hello, world», но на этот раз внесём в неё сразу три изменения: уберём пустую строку между `#include` и заголовком функции `main`, уберём восклицательный знак из печатаемой фразы, а после первого вызова `printf` добавим второй, печатающий «Good bye, world». Сравнение этих файлов в исполнении `diff -y` будет выглядеть так:

```
avst@host:~/work$ diff -y -W 80 hello.c hello2.c
#include <stdio.h>                      #include <stdio.h>
                                      <
int main()                              int main()
{                                       {
    printf("Hello, world!\n");        |     printf("Hello, world\n");
                                      >     printf("Good bye, world!\n");
    return 0;                               return 0;
}                                       }
avst@host:~/work$
```

Пожалуй, именно этот формат наиболее удобен для человеческого восприятия. Как ни странно, от программы `diff` чаще требуется прямо противоположное — представить изменения в виде, предназначенном для другой программы. Эта другая программа называется `patch` и предназначена для того, чтобы к копии исходного файла «применить» изменения, ранее обнаруженные с помощью `diff`. Используя связку `diff`/`patch`, программисты могут обмениваться изменениями, не пересылая друг другу весь исходный текст программы. Форматы описания изменений, специально предназначенные для такого применения, сделаны так, что в большинстве случаев программа `patch` может в полностью автоматическом режиме внести требуемые изменения даже в копию исходного файла, «не совсем» тождественную той, которая использовалась при запуске `diff`: скажем, если два программиста редактировали один и тот же файл, но в разных его местах (например, один в начале файла, другой — в конце), и один отправил свои изменения другому в виде выдачи `diff`, то в большинстве случаев второй программист сможет получить в своей копии исходных текстов изменения, внесённые первым программистом, несмотря на то, что в его копии файла нумерация строк уже совершенно другая.

Достигается это включением в выдачу `diff` так называемого *контекста* — нескольких (чаще всего трёх) неизменившихся строк, предшествующих модифицированному фрагменту, и стольких же — непосредственно следующих за ним. Программа `patch` использует контекст, чтобы привязаться к изменившейся нумерации строк исходного файла. Основных форматов выдачи `diff`, подразумевающих контекст, существует два: собственно «контекстный» (флаг `-c`) и так называемый «унифицированный» (`-u`). Чтобы составить общее впечатление об этих форматах, приведём выдачу `diff` с соответствующими ключами, полученную на тех же версиях файлов, что и в предыдущем примере:

```
avst@host:~/work$ diff -c hello.c hello2.c
*** hello.c     2016-05-26 18:48:44.000000000 +0300
--- hello2.c    2016-05-26 21:56:28.000000000 +0300
***************
*** 1,7 ****
  #include <stdio.h>
-
  int main()
  {
!     printf("Hello, world!\n");
      return 0;
  }
--- 1,7 ----
  #include <stdio.h>
```

```
  int main()
  {
!     printf("Hello, world\n");
!     printf("Good bye, world!\n");
      return 0;
  }
avst@host:~/work$ diff -u hello.c hello2.c
--- hello.c     2016-05-26 18:48:44.000000000 +0300
+++ hello2.c    2016-05-26 21:56:28.000000000 +0300
@@ -1,7 +1,7 @@
 #include <stdio.h>
-
 int main()
 {
-    printf("Hello, world!\n");
+    printf("Hello, world\n");
+    printf("Good bye, world!\n");
     return 0;
 }
```

Как можно заметить, унифицированный формат изрядно компактнее; у него имеются и другие достоинства, так что в основном в современных условиях используется именно он. Многие авторы утверждают, что этот формат годится только для версий `diff` и `patch`, созданных Фондом свободного программного обеспечения (GNU diff и GNU patch), но сейчас это уже не так — формат в действительности поддерживается практически всеми версиями этих программ, которые можно встретить.

Поскольку обычно исходные тексты программ представляют собой набор файлов, `diff` позволяет, что вполне естественно, сравнивать не отдельные файлы, а директории целиком, со всем их содержимым. Здесь обычно используют два флага — `-r` (от слова *recursive*), который включает режим сравнения директорий, и `-N` (от слова *new*), который предписывает `diff`'у рассматривать отсутствующие файлы как пустые. Для примера создадим две директории, `old/` и `new/`; в первую положим оригинальный вариант файла `hello.c`, во вторую — версию без восклицательного знака. Кроме того, в первой директории создадим файл `first.txt`, содержащий единственную строку текста «`Early bird gets the worm`», а во второй — файл `second.txt` с фразой «`But the second mouse gets the cheese`». Если применить к этим директориям команду `diff -Nur`, получится следующее:

```
avst@host:~/work$ diff -Nur old new
diff -Nur old/first.txt new/first.txt
--- old/first.txt       2016-05-27 18:07:57.000000000 +0300
+++ new/first.txt       1970-01-01 03:00:00.000000000 +0300
@@ -1 +0,0 @@
-Early bird gets the worm
diff -Nur old/hello.c new/hello.c
--- old/hello.c 2016-05-26 18:48:44.000000000 +0300
+++ new/hello.c 2016-05-27 18:08:47.000000000 +0300
@@ -2,6 +2,6 @@

 int main()
 {
-    printf("Hello, world!\n");
+    printf("Hello, world\n");
     return 0;
 }
diff -Nur old/second.txt new/second.txt
--- old/second.txt      1970-01-01 03:00:00.000000000 +0300
+++ new/second.txt      2016-05-27 18:08:16.000000000 +0300
@@ -0,0 +1 @@
+But the second mouse gets the cheese
avst@host:~/work$
```

Странная дата 1970-01-01 на самом деле означает *отсутствие* даты; действительно, ведь соответствующего файла просто нет. Дата и время в ОС Unix традиционно представляется как число секунд, прошедшее с полуночи по Гринвичу 1 января 1970 года, так что появление такой даты в выдаче программы соответствует арифметическому нулю; ну а загадочные три часа ночи на самом деле обусловлены часовым поясом Москвы, где проводился этот эксперимент.

Перенаправив вывод программы `diff` в файл, мы получим так называемый «патч» (англ. *patch*, см. также т. 1, сноску 15 на стр. 179) — список изменений, который можно применить автоматически. Опытные программисты часто, особенно при работе на языке Си, рекомендуют к флагам `-u`, `-r` и `-N` добавить ещё `-p`, который заставит `diff` указывать для каждого изменения, к какой функции языка Си это изменение относится. Если заголовок функции попал в выдаваемый контекст, то `diff` на этом успокаивается, если же заголовок в контекст не попал, программа добавляет его к заголовку изменения (после символов `@@`). Для некоторых других языков программирования `-p` тоже срабатывает. Итоговый набор флагов для получения патча проще всего запомнить в другом порядке: `-Nurp`. Некоторые программисты применяют эту комбинацию, давно забыв, что конкретно значит каждый из четырёх флажков. Итак, получаем патч (назовём его, например, `old-new.diff`):

```
diff -Nurp old new > old-new.diff
```

Зайдём теперь в директорию `old` и попробуем на её содержимое наложить изменения, описанные в нашем патче. Для этого мы воспользуемся программой, которая так и называется «`patch`». Файл с изменениями (собственно патч) подаётся ей в поток стандартного ввода, например, простым перенаправлением из файла. Кроме того, в большинстве случаев (в том числе и в нашем) потребуется указать программе `patch` параметр `-p1`; параметр `-p` в общем случае предписывает игнорировать заданное количество элементов имён файлов, или, если говорить совсем точно, программа `patch`, получив флаг `-p` с неким числом N, удаляет из имён файлов N первых символов «`/`» вместе со всеми буквами между ними. В нашем случае нужно убрать из имён файлов в патче имена директорий `old` и `new`. **В подавляющем большинстве случаев программу `patch` запускают именно так — с параметром `-p1`**, что и понятно: как правило, патч-файл получают путём сравнения двух директорий точно так же, как это сделали мы, а имена этих директорий при наложении патча следует, очевидно, проигнорировать. Итак, действуем:

```
avst@host:~/work$ cd old
avst@host:~/work/old$ ls
first.txt  hello.c
avst@host:~/work/old$ patch -p1 < ../old-new.diff
patching file first.txt
patching file hello.c
patching file second.txt
avst@host:~/work/old$ ls
hello.c  second.txt
avst@host:~/work/old$
```

Несложно убедиться (например, с помощью команды `cat`), что содержимое `hello.c` при этом тоже изменилось (исчез восклицательный знак), а `second.txt` содержит именно тот текст, который соответствующий файл содержал в директории `new`. При желании мы можем вернуть директорию в исходное состояние, применив тот же патч «в обратную сторону»; для этого укажем параметр `-R` (от слова *reverse*):

```
avst@host:~/work/old$ patch -p1 -R < ../old-new.diff
patching file first.txt
patching file hello.c
patching file second.txt
avst@host:~/work/old$ ls
first.txt  hello.c
avst@host:~/work/old$
```

Теперь, если вас попросят «прислать патч», вы будете точно знать, что делать: записать в файл выдачу команды `diff -Nurp` и полученный файл отправить по электронной почте или любым другим способом. Просьба «наложить патч» или «накатить патч» тоже не вызовет больших проблем: берёте пресловутый патч, заходите в директорию, где находятся тексты, которые нужно исправить, и запускаете команду «`patch -p1`», подав ей на стандартный ввод содержимое патча.

Наши эксперименты с программами `diff` и `patch` подсказывают причины возникновения одного очень важного правила, применяемого при оформлении программного кода: **в конце строки в тексте нельзя оставлять незначащие (невидимые) пробелы**. Поставьте себя на место человека, который, анализируя выдачу `diff`, видит, что некая строка заменена *на абсолютно такую же* строку (ведь пробелов-то не видно). О том, что причина на самом деле заключается в незначащих пробелах, можно догадаться не сразу; при этом немало времени уйдёт на внимательное побуквенное сличение двух строк. Между тем случайно оставить невидимый пробел в конце строки очень просто. К примеру, разбивая строку вроде

```
if(str) printf("%s\n", str);
```

на две, мы можем машинально поставить курсор на букву `p` в слове `printf` и нажать на Enter. Пробел, стоявший между скобкой и словом `printf`, так и останется на своём месте, но теперь он станет последним символом в своей строке, то есть превратится именно что в незначащий пробел. Будьте внимательны!

Приложение 6. Системы контроля версий

В ходе работы над любой мало-мальски сложной программой рано или поздно возникает вполне понятная боязнь испортить что-нибудь из уже реализованного. В принципе, любое редактирование исходного текста несёт в себе определённый риск, ведь людям свойственно ошибаться, и программисты знают это лучше, чем кто-либо ещё.

Вполне естественное «решение», применяемое начинающими программистами во избежание потерь при редактировании исходных текстов, состоит в том, что все исходные тексты периодически архивируются или просто копируются в отдельную директорию; имена таких архивов и директорий обычно включают в том или ином виде текущую дату. По мере набора негативного опыта программист начинает создавать такие копии всё чаще и чаще; сначала исходники копируются после завершения реализации очередной подсистемы, потом — после каждой новой дописанной функции, затем, возможно, после каждого сеанса редактирования. Иногда за один день появляется два, три архива, а то и больше.

Проблема тут, впрочем, не в количестве архивов: объёмы дисков нынче достаточно большие, исходные тексты программ обычно имеют совсем скромный размер, а при должной аккуратности в именовании можно даже не запутаться, где что лежит. Хуже другое: обнаружив, что в какой-то момент — возможно, достаточно давно — вы испортили какой-то фрагмент своего исходника, вы вряд ли сможете вспомнить дату, когда было внесено «ухудшающее» изменение (программисты называют неприятности подобного рода английским словом *regression*; общепринятого русскоязычного термина пока нет, так что часто так и говорят — ***регрессия***). Поиск нужного «среза» среди нескольких десятков, а то и сотен архивов может отнять много времени и сил; при попытке соединить найденный фрагмент старого исходника с теми полезными изменениями, которые вы внесли позже, часто получается совершенно нежизнеспособный гибрид, поскольку вы можете забыть, для каких целей вносились те или иные правки, упустить какую-то часть связанных между собой изменений и т. п. Несколько десятков чуть-чуть разных версий одного файла — это уже прекрасная возможность заблудиться, когда же проект сам по себе состоит из десятков файлов (модулей), в которые вносятся изменения, как связанные между собой, так и самостоятельные, и всё это существует в десятках копий от разных дат, положение постепенно становится отчаянным, а там и вовсе безнадёжным.

Ситуация резко обостряется, когда над программой работает не один человек, а хотя бы двое или трое, не говоря уже о коллективах из десяти-пятнадцати разработчиков. Уследить за своими собственными действиями ещё как-то можно; но представьте себе поток модификаций, внесённых в нетривиальный текст программы на протяжении хотя бы года несколькими программистами, *как-то* обменивающимися между собой очередными версиями, и нагромождения архивов исходных текстов, которые каждый из них сделал для себя в соответствии со своей собственной логикой — и вы поймёте, что задача поиска нужного варианта файла (или, хуже того, набора файлов) в таком месиве неразрешима принципиально.

Системы контроля версий позволяют не то чтобы полностью решить все эти проблемы, но хотя бы снизить их остроту. Такие системы на протяжении всего процесса разработки программы позволяют зарегистрировать каждое изменение исходных текстов, снабдить его комментарием, узнать, кто, когда и какие изменения внёс, а при необходимости — получить комплект исходных текстов или какую-то его часть в том состоянии, в котором файлы исходников существовали на определённую дату. Значимые моменты разработки, например, официальный выпуск очередной версии программы или просто «заработавшая» частная возможность, могут быть помечены специальными метками для удобства доступа к ним.

Больше того, все системы контроля версий позволяют поддерживать *одновременно* несколько вариантов истории изменений для одного и того же проекта. Например, часть разработчиков может заниматься одним направлением развития, часть — другим, а некоторые из разработчиков могут помогать то одной, то другой группе своих коллег, и до поры до времени эти группы могут действовать совершенно независимо друг от друга, обращая внимание только на свои изменения; состыковать результаты работы они смогут, когда придёт время — например, когда обе группы достигнут положительных результатов. Такие независимые фрагменты истории изменений называются ***ветками*** (англ. *branches*); они дают возможность проводить эксперименты, не боясь «сломать» основную версию программы. В некоторых случаях изменения, внесённые в отдельной ветке, так никогда и не включаются в основную версию программы; так происходит, если эксперимент признан неудавшимся, а также, например, когда приходится исправлять ошибки в старых версиях программы, причём с выпуском новой версии такие исправления по тем или иным причинам оказываются неактуальны.

При объединении изменений, внесённых разными людьми (в разных ветках или просто при одновременной работе), могут возникать ***конфликты***; это случается, когда участники разработки изменяют один и тот же фрагмент программы, внося *разные* изменения. Системы контроля версий изрядно облегчают процесс разрешения таких конфликтов. Так, при «ручном» объединении изменений участникам разработки достаточно было бы отредактировать один и тот же файл, чтобы столкнуться со сложностями; системы контроля версий обычно достаточно «интеллектуальны», чтобы понять, что изменения внесены в *разные места* одного файла, и в этом случае никакого конфликта не возникнет — система сама создаст новую версию файла, содержащую и те, и другие изменения. Конечно, автоматически подобные ситуации могут быть разрешены далеко не всегда, ведь изменения, внесённые разными людьми, могут коснуться одного и того же фрагмента текста; но и в этом случае система контроля версий оказывает существенную помощь, поместив конфликтующие варианты текста в один файл и выделив их специальными знаками, так что программистам остаётся только решить, как должен выглядеть итоговый текст.

База данных, в которой хранится вся история зарегистрированных изменений исходных текстов, называется ***репозиторием***. Как правило, репозиторий представляет собой обыкновенное дерево директорий с файлами, так что его можно, например, скопировать или перенести в другое место, запаковать в архив, создать резервную копию и т. п. Один из основных принципов работы с репозиторием состоит в том, что **единожды внесённая в репозиторий информация остаётся в нём навсегда**. Вы можете, например, удалить файл из проекта, и система контроля версий этот факт зарегистрирует, но в репозитории при этом останется и сам этот файл, и вся история его изменений, и дата, когда он был удалён. Если бы из репозитория можно было что-то удалять «по-настоящему», то после мы не смогли бы восстановить исходные тексты в состоянии, предшествующем удалению; между тем возможность восстановить *любую* версию наших исходников, существовавшую когда-либо на протяжении всей разработки — это одна из ключевых функций системы контроля версий.

Необратимость внесения информации в репозиторий имеет одно важное следствие: **перед регистрацией того или иного файла в системе контроля версий следует хорошо подумать**. Для текстовых файлов это обычно не создаёт проблем, но в некоторых (достаточно редких) случаях в репозиторий приходится вносить двоичные файлы, например, изображения, которые программа должна выводить на экран. Размеры двоичных файлов могут оказаться существенными, так что наш репозиторий «распухнет» и его станет трудно обслуживать, например, производить резервное копирование. Сказанное не означает, что двоичные файлы вообще нельзя регистрировать в системе; напротив, если вашу программу невозможно собрать без того или иного файла, то этот файл должен быть помещён в репозиторий в обязательном порядке. Просто следует сначала думать, а потом делать; этот принцип, впрочем, можно считать общезначимым.

Системы контроля версий делятся на ***централизованные*** и ***распределённые***. Централизованная система контроля версий подразумевает использование одного репозитория; все участники проекта перед началом сеанса работы получают из репозитория версию исходных текстов, содержащую свежие изменения, внесённые другими участниками, а в конце сеанса отправляют в репозиторий свои собственные изменения.

Распределённая система контроля версий работает по другому принципу. Для *каждой рабочей копии* исходных текстов создаётся собственный репо-

зиторий. Регистрация новых изменений, восстановление старых версий, переключение между ветками и другие штатные операции производятся с использованием локального репозитория; система позволяет синхронизировать содержимое разных репозиториев между собой.

Основной недостаток централизованных систем состоит в том, что для работы с ними вам необходим доступ к центральному репозиторию. Пока вы со всеми вашими коллегами, участвующими в том же проекте, сидите в одном офисе, проблем не возникает. Если программисты физически находятся в разных местах, доступ к репозиторию приходится предоставлять через Интернет, а это уже требует определённых мер по обеспечению информационной безопасности; кроме того, всем участникам проекта для работы требуется постоянный доступ в Интернет, что тоже не всегда возможно. Если использовать распределённую систему контроля версий, то ваш репозиторий будет сопровождать вас везде, где вы захотите, и вы, даже не имея доступа в Интернет, сможете воспользоваться всеми возможностями системы, за исключением получения свежих изменений от других участников и отправки им своих результатов. С другой стороны, распределённые системы устроены сложнее, и их намного сложнее освоить.

Ранние системы контроля версий, такие как RCS или SCCS, в наше время встречаются крайне редко. В течение сравнительно долгого периода — приблизительно с начала 1990-х годов до середины 2000-х — среди разработчиков программного обеспечения была популярна система CVS (*Concurrent Versioning System*), предполагающая использование центрального репозитория. Последний официальный релиз CVS вышел в 2008 году. Новые проекты, предпочитающие централизованный репозиторий, сейчас обычно используют систему Subversion (SVN), которая появилась в 2000 году и активно поддерживается до сих пор. Среди распределённых систем контроля версий наибольшей популярностью пользуется созданная Линусом Торвальдсом в 2005 году программа `git`. К распределённым относятся также Darcs (2003), Mercurial (2005), Bazaar (2005) и многие другие; в целом, судя по всему, распределённая модель организации репозиториев постепенно вытесняет централизованную.

Так или иначе, с наибольшей вероятностью вам могут встретиться именно `git` и, как ни странно, CVS, несмотря на то, что эту систему сегодня следует считать морально устаревшей; эпоха её господства тянулась так долго, что запас проектов, начатых в тот период и пока не сменивших CVS на что-то другое, исчерпается ещё не скоро. Эти две системы мы и рассмотрим, причём начнём с CVS как с более простой.

Система CVS

Все действия, предусмотренные в системе CVS, можно выполнить с помощью одной программы, которая так и называется `cvs`. Для начала вам потребуется создать репозиторий, в качестве которого CVS использует директорию с файлами; эту директорию обычно называют «CVS Root» (*корневая для CVS*). Если вы работаете в одиночку, проще всего создать репозиторий прямо в вашей домашней директории — например, если ваше имя пользователя «`vasya`», а домашняя директория — `/home/vasya`, то в роли репозитория вам вполне подойдёт `/home/vasya/cvsroot`. При работе в группе такое решение неудобно, но об этом позже.

CVS узнаёт о местонахождении репозитория из переменной окружения `CVSROOT`, так что вам придётся эту переменную задать. Если бы нам нужно было дать всего одну команду, мы могли бы задать переменную прямо в ней, например:

```
vasya@host:~$ CVSROOT=/home/vasya/cvsroot eval 'echo $CVSROOT'
/home/vasya/cvsroot
```

Напомним, что вместо `$CVSROOT` наш командный интерпретатор подставляет значение переменной `CVSROOT`, команда `echo` печатает свои аргументы, а команда `eval` *выполняет* свой аргумент, как если бы мы дали такую команду; без `eval` здесь ничего не получится, поскольку подстановки значений переменных командная оболочка успевает выполнить раньше, чем с переменными связываются заданные в команде временные значения; во время выполнения команды `eval` эти подстановки уже произведены, и это именно она подставляет временное значение вместо `$CVSROOT`, а `echo` печатает уже результат подстановки. На месте команды `eval` может выступать любая другая команда, в том числе команда `cvs`; в её окружении временные значения переменных будут установлены как надо.

В нашем случае одной командой дело не ограничится, так что удобнее будет по меньшей мере обеспечить существование нужной переменной на весь сеанс работы:

```
vasya@host:~$ export CVSROOT=/home/vasya/cvsroot
```

Ещё лучше будет сконфигурировать вашу систему так, чтобы переменная получала своё значение автоматически в каждом вашем сеансе работы. Интерпретатор командной строки `bash`, который мы используем[1], при запуске в качестве «лидера сеанса» пытается найти в вашей домашней директории файл с именем `.bash_profile`, а при любом другом запуске (например, в качестве интерпретатора командно-скриптового файла) — файл `.bashrc`, и если соответствующий файл найден, выполняет все записанные в таком файле команды. Для верности лучше вставить строку

```
export CVSROOT=/home/vasya/cvsroot
```

в оба этих файла. Чтобы проверить, что всё в порядке, правильнее всего будет завершить свой сеанс работы, зайти в систему снова, дать команду «`echo $CVSROOT`» и убедиться, что она печатает то, что нужно.

Создав директорию и установив переменную `CVSROOT`, сообщим системе, что мы хотим начать использование нового репозитория. Это делается командой «`cvs init`», которая создаст нужные для работы служебные файлы.

В одном репозитории система CVS позволяет размещать произвольное количество проектов (в терминах CVS — «модулей»). Традиционно каждый проект рассматривается как директория с файлами (и, возможно, поддиректориями), и в репозитории система тоже создаёт отдельную директорию для каждого проекта; это полезно знать, если будет нужно перенести проект со всей его историей изменений из одного репозитория в другой.

Для создания в репозитории нового проекта используется команда «`cvs import`». Будьте внимательны с этой командой! Она предполагает, что все файлы, содержащиеся в текущей директории (то есть в той, где вы находитесь, давая эту команду), как раз составляют импортируемый проект, и прилежно вносит их все (со всеми поддиректориями) в репозиторий. Если вы ошибётесь и дадите эту команду, например, в вашей домашней директории, CVS попытается скопировать в репозиторий вообще все ваши файлы, что вряд ли соответствует вашим желаниям. Впрочем, ничего фатального при этом не случится, можно будет просто зайти в ваш репозиторий и удалить оттуда ошибочно созданный проект с помощью команды «`rm -r`». Так или иначе, если вы начинаете проект с нуля, для его внесения в репозиторий вам понадобится пустая директория.

В непустых директориях команду `cvs import` дают, когда нужно начать регистрировать историю изменения исходных текстов, которые уже существуют — например, их написал другой программист, а вы хотите продолжить работу над его проектом. Очевидно, создатели CVS именно эту ситуацию и имели в виду, но в реальности так случается очень редко; чаще в репозитории создают именно пустой проект, в который затем вносят файлы.

Команда `cvs import` требует указания трёх параметров, первый из которых — это имя вашего проекта, под которым он будет известен системе, а два других — так называемые «метка производителя» и «метка выпуска» (*vendor tag* и *release tag*). Скорее всего, по замыслу создателей CVS эти два параметра должны были как-то идентифицировать происхождение исходных текстов, импортируемых в новый проект, но в реальности они *никак и ни для чего не используются*, несмотря на то, что система требует их обязательного указания и даже тщательно проверяет их соответствие неким условиям. Можно написать в этих параметрах любые слова, состоящие из латинских букв, обязательно разные, хотя бы и представляющие собой произвольную белиберду; но правильнее будет, например, в качестве «метки производителя» указать своё имя, а в качестве «метки выпуска» — `start` или что-нибудь подобное. Итак, создаём пустую директорию, заходим в неё и там даём команду `cvs import`:

```
vasya@host:~$ mkdir empty
vasya@host:~$ cd empty
vasya@host:~/empty$ cvs import myprogram vasya start
```

[1] Если вы используете другой интерпретатор командной строки, обратитесь к его документации.

Программа `cvs` на каждое действие с контролируемыми файлами требует указания комментария, для чего запускает редактор текстов (скорее всего, Vim, но если в вашей системе по умолчанию настроен другой редактор текстов, то его), давая вам возможность прокомментировать свои действия. Так она поступит и в этот раз. Запуска текстового редактора можно избежать, указав комментарий прямо в командной строке с помощью флажка `-m` (от слова *message*), примерно так:

```
cvs import -m "Starting the project" myprogram vasya start
```

Наконец, приняв команду, система ответит фразой «`No conflicts created by this import`»; теперь в вашем репозитории есть проект, который называется `myprogram`. Интересно, что в репозитории-то он есть, а вот рабочей директории, в которой можно редактировать файлы и отправлять изменения в систему, пока что нет — её нужно создать, получив из репозитория проект. Это делается командой `cvs get`, в данном случае — «`cvs get myprogram`». Учтите, что при этом в текущей директории будет создана директория `myprogram`, которая как раз и станет рабочей директорией для вашего проекта. Имеет смысл давать эту команду в том месте, где вы хотите работать, а не там, где вы делали `cvs import`, например:

```
vasya@host:~/empty$ cd
vasya@host:~$ cd work
vasya@host:~/work$ cvs get myprogram
cvs checkout: Updating myprogram
vasya@host:~/work$ cd myprogram
vasya@host:~/work/myprogram$ ls -F
CVS/
vasya@host:~/work/myprogram$
```

Как видим, в рабочей директории система создала поддиректорию `CVS` для хранения служебной информации. Там же хранится информация о расположении репозитория, так что **когда вы даёте команды `cvs`, находясь в рабочей директории, содержимое переменной `CVSROOT` игнорируется**; система использует тот репозиторий, из которого была изначально получена данная рабочая копия. Учтите это, если будете переносить репозиторий в другое место.

Чтобы получить представление о том, как протекает работа с CVS, создадим в нашей директории файл — например, `hello.c`, содержащий всё ту же программу «Hello, world». О появлении в проекте нового файла нужно сообщить системе с помощью команды `cvs add`:

```
vasya@host:~/work/myprogram$ ls -F
CVS/  hello.c
vasya@host:~/work/myprogram$ cvs add hello.c
cvs add: scheduling file ‘hello.c’ for addition
cvs add: use ‘cvs commit’ to add this file permanently
vasya@host:~/work/myprogram$
```

Система следит за изменениями только тех файлов, о появлении которых вы ей сообщили. Это позволяет избежать случайного попадания в репозиторий всевозможных исполняемых, объектных, временных и прочих файлов, которые не относятся к числу ваших исходных текстов; напомним, что передача любой информации в репозиторий необратима, так что этот момент очень важен.

Отправим наш файл в репозиторий; как можно догадаться из предыдущих сообщений системы, это делается командой `cvs commit`:

```
vasya@host:~/work/myprogram$ cvs commit -m "Wrote hello"
cvs commit: Examining .
/home/vasya/cvsroot/myprogram/hello.c,v  <--  hello.c
initial revision: 1.1
vasya@host:~/work/myprogram$
```

Если не указать параметр `-m`, программа запустит текстовый редактор для ввода комментария; кстати, в некоторых случаях это даже удобнее, поскольку во временном файле, который будет открыт в редакторе, вы увидите заботливо собранную системой информацию о том, какие файлы были добавлены, изменены или удалены; во многих случаях это помогает вспомнить, что́ реально было сделано, и написать более осмысленный комментарий. Сразу дадим довольно очевидный совет: **пишите комментарии по-английски**. CVS не содержит на этот счёт никаких ограничений, но в будущем вам, возможно, захочется показать историю своих изменений кому-то, кто не говорит по-русски.

Типичный сеанс работы с контролируемыми исходными текстами начинается с команды `cvs update`, которая извлекает из репозитория изменения, внесённые из других рабочих копий, и отражает их в ваших рабочих файлах. Полезно помнить, что по умолчанию `cvs update` не создаёт директории, так что если в другой рабочей копии была создана новая директория, вы её у себя не получите; исправить это можно указанием флага `-d`, причём некоторые программисты предпочитают этот флаг указывать при каждом вызове `update`. После приведения своей рабочей копии в соответствие репозиторию вы можете приступить к редактированию файлов и, возможно, созданию новых; каждый создаваемый файл, если он относится к числу ваших исходных текстов, следует зарегистрировать в системе с помощью `cvs add`.

Удаление файлов из проекта производится командой `cvs remove`, причём она отказывается удалять файл, если он всё ещё находится в вашей рабочей директории, так что сначала файл нужно убрать обычной командой `rm`:

```
vasya@host:~/work/myprogram$ rm file12.c
vasya@host:~/work/myprogram$ cvs remove file12.c
cvs remove: scheduling ‘file12.c’ for removal
cvs remove: use ‘cvs commit’ to remove this file permanently
vasya@host:~/work/myprogram$
```

Здесь вы можете заметить, что действовать так не совсем удобно, поскольку на момент выполнения второй команды файла уже нет и автодополнение его имени не работает, так что приходится его имя набирать целиком. Можно, конечно, воспользоваться менее известными возможностями командного интерпретатора и набрать что-то вроде «`cvs remove !:1`», но помнить такие тонкости не обязательно: эффект двух команд можно совместить, указав флаг `-f`:

```
vasya@host:~/work/myprogram$ cvs remove -f file12.c
cvs remove: scheduling ‘file12.c’ for removal
cvs remove: use ‘cvs commit’ to remove this file permanently
vasya@host:~/work/myprogram$
```

До команды `commit` вы можете передумать и отменить удаление файла, добавив его обратно командой `cvs add`; в этом случае файл будет получен обратно из репозитория, и всё будет выглядеть так, будто никто этот файл не удалял. Впрочем, даже выполнение `commit`, как мы уже знаем, на самом деле лишь пометит в репозитории ваш файл как удалённый, но физически файл удалён не будет; вы в любой момент можете получить его назад и снова внести в проект.

Отметим, что CVS не поддерживает переименование файлов; единственный способ переименовать файл в проекте — это удалить его и добавить обратно под другим именем. CVS при этом не будет знать, что это тот же самый файл, так что история изменений для него начнётся с нуля. Отсутствие средств переименования называют едва ли не самым значительным недостатком CVS; старайтесь продумывать принципы именования файлов так, чтобы их не приходилось переименовывать.

Когда вы достигнете той или иной «логической точки» или просто решите, что пришло время зафиксировать внесённые изменения, дайте команду `cvs commit`. Даже если вы намерены продолжить работу, полезно считать, что отдельно взятый сеанс на этом завершён и начать новый сеанс с самого начала, то есть с команды `cvs update`. Если вы работаете в группе, то за время вашего предыдущего сеанса кто-то из ваших коллег мог отправить в репозиторий свои изменения; чем раньше вы их получите, тем меньше вероятность возникновения конфликтов, при которых система не может совместить имеющиеся изменения автоматически и требует ручного вмешательства.

Команда `cvs update` позволяет получить из репозитория не только актуальную (самую последнюю) версию контролируемых файлов, но и *любую* их версию. Так,

```
cvs update -D "Apr 12, 2015 12:30"
```

приведёт рабочую копию в соответствие тому состоянию исходных текстов, которое было зарегистрировано в репозитории 12 апреля 2015 года в 12:30; команда

```
cvs update -D "15 days ago"
```

восстановит версию пятнадцатидневной давности. Кроме того, той или иной версии исходников можно дать *имя* (*tag* — если угодно, метку или тег) с помощью команды `cvs tag`, например:

```
cvs tag as_released_for_mr_petrov
```

Тегами обычно помечают версии, по той или иной причине выделяющиеся из общего ряда — например, когда начинает работать та или иная возможность или когда делается официальный релиз программы. Версию исходников, помеченную таким именем-тегом, можно в любой момент восстановить, используя имя тега:

```
cvs update -r as_released_for_mr_petrov
```

Полезно помнить, что команда «`cvs update -A`» снова приведёт вашу рабочую копию в соответствие с самыми поздними изменениями, зафиксированными в репозитории.

Для изучения истории изменений и текущего положения вещей очень полезны команды `cvs log` и `cvs status`. Первую обычно запускают с указанием конкретного имени файла, и она выдаёт список всех версий заданного файла вместе с комментариями, которыми сопровождалось каждое зарегистрированное в репозитории изменение этого файла. Вторая команда позволяет узнать, как соотносится с репозиторием та версия файла, которая находится в рабочей директории. Подробно описывать эти команды — задача неблагодарная, будет лучше, если вы сами поэкспериментируете с ними; практически сразу вам всё станет понятно.

Как уже говорилось, системы контроля версий умеют поддерживать одновременно несколько независимых *веток* («бранчей») истории изменений. Если вы собираетесь вносить в проект изменения, в полезности или успешности которых сомневаетесь, либо просто по той или иной причине не хотите, чтобы ваши изменения затрагивали основной ход истории разработки, вы можете потребовать от системы создать отдельную ветку для регистрации ваших изменений; в будущем вы при необходимости сможете объединить ваши изменения, сделанные в отдельной ветке, с основным ходом изменений или с другой веткой. Для создания ветки используется `cvs tag` с флажком `-b` и указанием имени для новой ветки, примерно так:

```
cvs tag -b doubtful_feature
```

Интересно, что такая команда создаст ветку `doubtful_feature`, но не переведёт вашу рабочую копию в режим работы с этой веткой. Последнее достигается командой `update`:

```
cvs update -r doubtful_feature
```

После этого все изменения, которые вы из вашей рабочей директории отправите в репозиторий с помощью `cvs commit`, будут отражаться именно в ветке `doubtful_feature`. Вернуться в основную ветку можно с помощью уже знакомой нам команды «`cvs update -A`».

Если изменения, зарегистрированные в отдельной ветке, будут признаны достойными включения в основную версию (или в другую ветку, так тоже бывает), вы можете воспользоваться командой `update` с флагом `-j` (от слова *join*) для присоединения изменений из другой ветки к текущей версии. Например, последовательность команд

```
cvs update -A
cvs update -j doubtful_feature
cvs commit
```

зарегистрирует в основной ветке все изменения, ранее внесённые в ветку `doubtful_feature`.

Репозиторий не обязан находиться на той же машине, где расположена ваша рабочая директория. Чаще всего репозиторий располагают на компьютере, к которому можно подключиться удалённо с помощью программы `ssh` (*secure shell*). При этом потребуется задействовать две переменные окружения — уже знакомую нам `CVSROOT`, в которой указать доменное имя машины с репозиторием, имя вашей учётной записи и расположение репозитория, и `CVS_RSH`, в которой задаётся имя программы, используемой для удалённого доступа (`ssh`). Например, если репозиторий расположен на машине `cvsserv.example.com` в директории `/home/cvs/CVSRoot`, а ваша учётная запись на этой машине называется `vasya`, то переменные окружения нужно установить следующим образом:

```
export CVSROOT=vasya@cvsserv.example.com:/home/cvs/CVSRoot
export CVS_RSH=ssh
```

Если ваша локальная учётная запись называется так же, как и запись на машине с репозиторием, то указывать её не обязательно (в нашем примере можно убрать из `CVSROOT` слово `vasya` вместе с символом «`@`»). Порядок работы с CVS при использовании удалённого доступа практически не меняется, за исключением того, что при каждом обращении к серверу система будет требовать ввести пароль. Этого можно избежать, если организовать удалённый доступ с использованием криптографических ключей, но рассказ об этом мы оставим за рамками нашей книги; в Интернете имеется достаточное количество руководств и описаний на эту тему, в том числе на русском.

Возможность доступа к удалённому (то есть находящемуся на другом компьютере) репозиторию открывает путь к организации групповой работы; собственно, для этого системы контроля версий изначально и предназначены. Организовать доступ группы программистов к одному репозиторию технически довольно просто. Для этого на машине, где располагается репозиторий, необходимо завести учётные записи для каждого участника проекта, объединить их в группу (обычно эту группу так и называют — `cvs`), а сам репозиторий сделать доступным для группы на чтение и запись, например, вот так:

```
chown :cvs -R /home/cvs/CVSRoot
chmod ug+rwX -R /home/cvs/CVSRoot
```

Когда над одним проектом работает несколько человек, иногда возникает ситуация *конфликта изменений* — попросту говоря, когда один и тот же фрагмент контролируемых файлов разные участники проекта изменили по-разному. Надо сказать, что CVS в этом плане ведёт себя достаточно стойко; простым внесением изменений в один и тот же файл систему не смутить, для возникновения конфликта нужно, чтобы изменённые фрагменты перекрывались или хотя бы располагались друг от друга на расстоянии менее чем в три строки. Тем не менее рано или поздно вы с конфликтной ситуацией столкнётесь. Между прочим, конфликт не всегда возникает именно при групповой работе: например, при слиянии двух веток (бранчей) система может заявить о конфликте даже когда вы работаете в гордом одиночестве, хотя, конечно, в этом случае организационно ситуация выглядит гораздо проще: конфликт у вас возник *с самим собой*, ну а с собой договориться обычно проще, чем с другими.

Так или иначе, если конфликт уже возник, пугаться не надо: система делает всё возможное, чтобы облегчить выход из создавшегося положения. Обычно конфликт обнаруживается при попытке отправить свои изменения в репозиторий; выглядит это примерно так:

```
cvs commit: Up-to-date check failed for ‘file1.c’
cvs [commit aborted]: correct above errors first!
```

Появление такого сообщения само по себе не означает, что возник конфликт; просто в репозитории в настоящий момент имеется версия файла `file1.c`, которая *свежее* той, в которую вы вносили изменения в своей рабочей копии. Иначе говоря, вы в какой-то момент получили из репозитория версию этого файла и начали её редактировать, а в это время кто-то из ваших коллег этот файл тоже успел отредактировать и отправить результат в репозиторий. Сообщение «`Up-to-date check failed`» означает, что вам необходимо запустить `cvs update`; при этом система постарается совместить в вашем локальном файле `file1.c` как те изменения, которые внесли в него вы, так и те, которые зарегистрированы в репозитории. В большинстве случаев системе это удастся, так что вам останется только повторить команду `cvs commit`. И лишь в ситуации, когда в автоматическом режиме система ничего сделать не может, в ответ на команду `update` вы увидите что-то вроде следующего:

```
Merging differences between 1.1 and 1.2 into file1.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in file1.c
C file1.c
```

Впрочем, даже в этой ситуации всё на самом деле не так уж и плохо, просто система требует ручного объединения изменений. Всё необходимое для этого уже помещено в ваш локальный файл, так что вам остаётся открыть его в редакторе текстов, найти все конфликтные фрагменты (которые отмечены весьма заметными знаками) и решить, какое итоговое содержание оставить. В файле размеченные конфликтующие версии выглядят примерно так:

```
<<<<<<< file1.c
    printf("Hello, world\n");
=======
    printf("Hello, wonderful world!\n");
>>>>>>> 1.2
```

Пометки <<<<<<<, ======== и >>>>>>>> вставлены системой CVS и обозначают соответственно начало конфликтного фрагмента, разделитель (отделяющий вашу версию от той, которая взята из репозитория) и конец фрагмента. В данном случае можно предположить, что изначально строковая константа содержала фразу «`Hello, world!`», но вы решили из неё убрать восклицательный знак, а ваш коллега в это время добавил в ту же константу слово «`wonderful`». Технически самый простой способ ликвидации конфликта — попросту выкинуть версию, написанную коллегой, стереть служебные пометки, вставленные системой, и в таком виде сохранить файл и отправить его в репозиторий с помощью `cvs commit`; но если вы часто будете так делать, рано или поздно коллеги устроят вам «тёмную» в пыльном углу и будут совершенно правы. Правильнее будет при обнаружении конфликта связаться с коллегой, который внёс слово `wonderful`, и спросить, очень ли это слово ему дорого. Вполне возможно, именно вариант, написанный другим программистом, сильнее приближает проект к успеху, а вас — к премии; упираться на манер барана далеко не всегда осмысленно.

Система `git`

Обсуждая CVS, мы рассказали практически обо всех основных возможностях этой программы, хотя и очень кратко. К сожалению, с `git` такой номер не пройдёт: даже самое краткое описание всех её основных возможностей и приёмов использования займёт добрую сотню страниц. В принципе, то же самое в большей или меньшей степени относится ко всем системам контроля версий, использующим распределённую модель. Не пытаясь объять необъятное, мы покажем, как начать работу с `git` и как в этой системе делать всё то, что мы уже умеем делать с помощью CVS, плюс ещё немного; с остальными возможностями вы либо разберётесь сами, когда они вам понадобятся, либо они вам вообще не понадобятся — так тоже часто бывает, когда речь идёт о достаточно сложных инструментах.

Все основные действия, предусмотренные системой, можно выполнить с помощью одной программы, которая называется `git` и, как и `cvs`, получает команду через свой (обычно первый) параметр командной строки.

Как уже говорилось, распределённые системы контроля версий предполагают поддержку своего репозитория при каждой рабочей директории. Этим обусловлено одно довольно важное концептуальное упрощение в сравнении с централизованными системами: отдельно взятый репозиторий здесь обслуживает только один проект. Теоретически это можно обойти, используя уже знакомые нам *ветки*, но гораздо проще и естественнее будет для каждого проекта завести собственный репозиторий. Заодно отметим, что репозиторий в таких условиях не нуждается в имени и идентифицируется только своим местоположением.

Превратить произвольную директорию в рабочую, имеющую свой репозиторий, можно одной командой «`git init`» без дополнительных параметров. Можно не беспокоиться о наличии в этой директории файлов, в том числе таких, которые мы не собираемся помещать в репозиторий: `git init` в любом случае создаст репозиторий *пустым*. На первый взгляд выполнение команды `init` ничего не меняет в окружающей действительности, но так кажется лишь потому, что в ОС Unix, как мы знаем, файлы, имена которых начинаются с точки, считаются *невидимыми*. Дав сразу после `git init` команду `ls -a`, мы обнаружим, что в текущей директории появилась поддиректория с именем «`.git`»; это и есть созданный репозиторий.

После создания репозитория желательно указать системе, как вас (как автора вносимых изменений) следует идентифицировать. Создатели `git` предлагают использовать для этого имя и адрес электронной почты, например, так:

```
git config user.name "Vasya Ivanov"
git config user.email "vasya.ivanov@example.com"
```

но с таким указанием адреса могут возникнуть определённые проблемы. Программы, предназначенные к свободному распространению, часто публикуют *в виде git-репозитория*, в котором, естественно, все внесённые вами изменения будут помечены с использованием ваших имени и адреса. Адрес электронной почты, попавший в Интернет, рано или поздно оказывается в базах данных у спамеров со всеми вытекающими последствиями в виде медленно, но верно растущего потока непрошеных писем. С другой стороны, совсем не указывать свой адрес хотя и можно (технически это не создаст больших проблем, разве что в том же проекте обнаружится ваш полный тёзка), но не совсем правильно, ведь кому-то может потребоваться связаться с вами как с автором того или иного фрагмента программы, чтобы обсудить какие-то технические детали. Поэтому правильнее будет, наверное, указать адрес в таком виде, в котором распознать его может только человек, например:

```
git config user.email "vasya.ivanov at example dot com"
```

Если вы работаете с большим количеством репозиториев и не хотите каждый раз настраивать свою идентификацию, можно сэкономить время, сделав параметры «глобальными» с помощью флага `--global`:

```
git config --global user.name "Vasya Ivanov"
git config --global user.email "vasya.ivanov at example dot com"
```

В этом случае `git` запишет значения параметров в файл `.gitconfig` в вашей домашней директории и будет использовать их, если для данного конкретного репозитория не установлены свои собственные значения. Отметим, что идентифицирующую информацию лучше не менять и использовать (во всяком случае, в рамках одного проекта) каждый раз одну и ту же; если вы используете несколько репозиториев в работе над одним проектом — например, на разных машинах — позаботьтесь о том, чтобы имя и адрес были установлены одинаково с точностью до буквы. Нередки ситуации, когда нужно, например, найти все изменения, внесённые в проект конкретным человеком, и в таких случаях возникают определённые сложности, если в разное время этот человек использовал разные (хотя, быть может, и похожие) имена.

Поместить файлы под контроль `git` можно с помощью команды `git add`, параметрами которой служат имена файлов. Отметим, что сходство с аналогичной командой системы CVS тут кажущееся: на самом деле `git add` добавляет не файлы, а *их изменения*, и добавляет их не «в систему вообще», а конкретно в тот набор изменений, который будет зарегистрирован в репозитории ближайшей командой `commit`. Сам этот набор изменений в документации называется «индексом». Иначе говоря, если мы отредактируем файл, сделаем на него `git add`, потом снова отредактируем его и после этого сделаем `git commit`, то *в репозиторий попадёт версия, имевшая место на момент последнего `git add`*, а все изменения, внесённые в файл между командами `add` и `commit`, система проигнорирует, поскольку они не внесены в индекс. Некоторые программисты считают такой подход удобным; другие предпочитают снабжать команду `commit` флагом `-a`, который сначала добавляет в индекс все изменения, *внесённые в известные системе файлы* (включая, кстати, удаления файлов, которые в норме следует выполнять командой `git rm`). Если с каждым `commit`'ом использовать `-a`, роль `git add` сводится к добавлению в систему новых файлов подобно тому, как мы делали это в CVS.

Пока вы не наберёте достаточно опыта, такая трёхступенчатая схема, состоящая из рабочей директории, индекса и репозитория, может казаться запутанной. «Распутать» её вам поможет, во-первых, команда `git status`, показывающая, какие файлы внесены в индекс (то есть уйдут в репозиторий, если сделать `commit` прямо сейчас), какие изменились, но не внесены в индекс, а какие файлы присутствуют в вашей рабочей директории, но не находятся под контролем `git`; во-вторых, команда `git diff`, показывающая разницу между текущим содержимым отслеживаемых файлов и той их версией, которая помещена в индекс.

Удаление файла производится командой `git rm`; файл не должен содержать изменений ни в индексе, ни в рабочей директории, иначе команда выдаст ошибку (впрочем, это ограничение можно обойти указанием флажка `-f`, но лучше этого не делать). Команда удаляет файл из рабочей директории и помечает его как удалённый в индексе, так что следующий `commit` зарегистрирует удаление в репозитории. Пока вы не сделали `commit`, у вас остаётся возможность передумать; вернуть свой файл обратно вы можете в два шага:

```
git reset HEAD -- file21.c
git checkout -- file21.c
```

Команда `reset` приведёт индекс в соответствие с последней версией, отражённой в репозитории (эта версия обозначается словом `HEAD`); команда `checkout` предназначена для приведения *рабочей директории* в соответствие с той или иной версией из репозитория, если же версию не указать — то из индекса,

что и имеет место в данном случае. Эту команду вы можете использовать не только при ошибочном удалении, но и при ошибочном изменении файла.

Система `git` позволяет переименовать файл; это делается командой `git mv` с двумя параметрами — старым и новым именем файла. В рабочей директории файл при этом переименовывается, в индексе помечается как переименованный, а ближайший `commit`, как обычно, фиксирует факт переименования в репозитории; вся история изменений файла при этом сохраняется. Автору доводилось видеть программистов, называвших возможность переименования файлов главной причиной их перехода с CVS на `git`.

Отдельного обсуждения заслуживают уже знакомые нам по системе CVS *ветки* (бранчи). Пользователи централизованных систем контроля версий обычно рассматривают создание ветки как некое серьёзное действие, которое нуждается в обдумывании, обсуждении с коллегами, возможно, даже в получении разрешения у руководителя разработки. Это можно понять, ведь репозиторий там один на всех, так что участники работы опасаются засорять его лишней информацией. Пользователи распределённых систем воспринимают ветки совершенно иначе, поскольку **ветки по умолчанию локальны в репозитории**, никто не заставляет передавать их в другие репозитории и вообще кому-то показывать. Иными словами, в распределённой системе контроля версий ветки представляют собой ваш личный локальный рабочий инструмент, использование которого остаётся целиком на ваше усмотрение и никак не касается других участников работы, если только вы сами не захотите им предложить использовать ваши ветки. При передаче изменений между репозиториями вы не обязаны передавать в другой репозиторий свои ветки, вы даже можете не сообщать никому об их существовании — но, с другой стороны, при желании вы можете и передать информацию о своих ветках, и создать в удалённых репозиториях ветки под теми же (или другими) именами.

В системе `git` ветка — это просто некое подмножество из всех изменений, которые зарегистрированы в вашем репозитории. Две разные ветки могут существовать абсолютно независимо, но могут и пересекаться, накладываться, сливаться — всё зависит от ваших действий и ваших желаний. Если веток стало слишком много, любую из них можно в любой момент удалить (ещё одна возможность, практически недоступная пользователям того же CVS, в котором каждое изменение привязано к своей ветке), только желательно, чтобы входящие в удаляемую ветку изменения к этому моменту были отражены в какой-то другой ветке; в противном случае, хотя изменения и не исчезнут, доставать их из репозитория придётся по одному, что довольно противно.

Репозиторий с несколькими ветками можно рассматривать как несколько репозиториев в одном; в самом деле, ведь никто не мешает вам создать несколько рабочих директорий, каждую со своим репозиторием, и при необходимости пересылать изменения между ними. Существуют и такие распределённые системы контроля версий, где вообще не поддерживаются ветки, а вместо них предлагается создавать отдельные репозитории (один из самых известных примеров такой системы — Darcs); но `git` ветки поддерживает, и делает это настолько эффективно, что время, затрачиваемое на «смену декораций» при переходе от одной ветки к другой, совершенно неощутимо. Пользователи, перешедшие на `git` с централизованных систем, обычно довольно быстро входят во вкус активной работы с ветками.

При создании нового (пустого) репозитория в нём автоматически заводится ветка с именем `master`; узнать, какие ветки у вас есть и какая из них сейчас активна, можно с помощью команды `git branch`:

```
vasya@host:~/work/prog1$ git branch
* master
vasya@host:~/work/prog1$
```

Символ «`*`» отмечает активную ветку — ту, с которой вы сейчас работаете. Создать новую ветку можно с помощью той же команды `branch`:

```
vasya@host:~/work/prog1$ git branch doubtful_changes
vasya@host:~/work/prog1$ git branch
  doubtful_changes
* master
vasya@host:~/work/prog1$
```

Как видим, ветка появилась, но не стала активной. Переключиться из одной ветки в другую можно с помощью `git checkout`:

```
vasya@host:~/work/prog1$ git checkout doubtful_changes
Switched to branch 'doubtful_changes'
vasya@host:~/work/prog1$ git branch
* doubtful_changes
  master
vasya@host:~/work/prog1$
```

В принципе эти два действия — создание ветки и переключение в неё — можно совместить, дав команду `checkout` с флажком `-b`; чаще всего так и делают:

```
vasya@host:~/work/prog1$ git checkout -b experiments
Switched to a new branch 'experiments'
vasya@host:~/work/prog1$ git branch
  doubtful_changes
* experiments
  master
vasya@host:~/work/prog1$
```

Как уже говорилось, `git checkout` работает очень быстро — гораздо больше времени тратится на то, чтобы эту команду набрать. В разных ветках у вас могут быть разные файлы и директории, один файл может в разных ветках иметь совершенно разное содержание и т. д.; систему этим не смутить.

В некоторых случаях переключение ветки могло бы привести к потере информации; в таких случаях `git` отказывается выполнять `checkout`. Простейший пример такой ситуации — если у вас в рабочей директории имеются изменённые файлы, *причём их версия в текущей ветке и в той, куда вы хотите перейти, отличается*. В этом случае вы получите примерно такое сообщение:

```
error: You have local changes to 'first.txt'; cannot switch branches.
```

Отметим, что внесённые в файл изменения не станут препятствием к смене ветки, если версия этого файла в обеих ветках одинакова; в этом случае система просто считает, что вы решили зарегистрировать ваши изменения в другой ветке.

Аналогичным образом, хотя и с другой диагностикой, `git` отказывается менять ветку, если в вашем текущем индексе присутствуют изменения, которые вы ещё не зарегистрировали в репозитории. Самый простой способ избежать всех этих проблем — это сначала полностью завершить сеанс работы с текущей веткой, ненужные изменения файлов отменить, нужные — отдать в репозиторий с помощью `commit`, и только после этого переключаться в другую ветку. К сожалению, иногда это оказывается неудобно — например, вы ещё не довели работу до логической точки, когда хотелось бы делать `commit`, но для продолжения работы вам необходимо заглянуть в какой-то файл, который есть только в другой ветке. Даже в такой странной ситуации `git` предоставляет довольно удобное решение: с помощью команды `git stash` вы можете временно сохранить «где-то» (на самом деле в репозитории, но это не важно, поскольку в общую историю такие сохранения не входят) все изменения, которые успели внести в рабочую директорию и индекс. Рабочая директория и индекс при этом очищаются, то есть приводятся в соответствие с последней версией текущей ветки, так что вы можете сменить ветку, поработать в другой ветке, затем перейти назад в исходную ветку и восстановить состояние рабочей директории и индекса, которое раньше сохранили с помощью `stash`; делается это командой «`git stash pop --index`». Если флаг `--index` не указать, восстановлены будут только файлы рабочей директории, а индекс будет считаться пустым; в большинстве случаев это приемлемо, ведь локальные файлы тоже содержат те изменения, которые вы отразили в индексе.

Пожалуй, самое нетривиальное в распределённой модели репозиториев — это обмен информацией между ними. Отметим для начала, что новый репозиторий может быть создан не только пустым, но и как копия другого (уже существующего) репозитория; для этого служит команда `git clone`. Если копируется репозиторий, находящийся на той же машине, достаточно указать путь к нему, то есть полное имя рабочей директории; создать копию репозитория, находящегося на другой машине, к которой имеется доступ через `ssh`, можно примерно так:

```
git clone ssh://vasya@gitserv.example.com/home/vasya/work/prog1
```

Существуют и другие способы указания репозитория, но их мы рассматривать не будем.

Прежде чем двигаться дальше, нам придётся разобраться с двумя концепциями, которые использует `git` при взаимодействии репозиториев между собой. Во-первых, для обмена данными с другим репозиторием этот другой репозиторий (называемый *удалённым*, англ. *remote*) должен быть зарегистрирован в настройках нашего (локального) репозитория и снабжён именем. Когда мы создаём пустой репозиторий, никаких удалённых репозиториев он изначально не знает; но когда репозиторий создаётся как копия (клон) другого, в создаваемом репозитории его исходный репозиторий (то есть тот, копией которого он является) автоматически регистрируется в качестве удалённого под именем `origin`.

Манипулировать списком зарегистрированных удалённых репозиториев позволяет команда `git remote`; в частности, если дать её без дополнительных параметров, она выдаст список зарегистрированных удалённых репозиториев, но покажет при этом только их имена, что не всегда удобно:

```
vasya@devserv:~/TMP/prog1$ git remote
devserv
origin
vasya@devserv:~/TMP/prog1$
```

Более подробные результаты можно получить, если добавить флажок «`-v`» (от слова *verbose*):

```
vasya@vbook:~/TMP/prog1$ git remote -v
devserv ssh://devserv.example.com/home/vasya/prog1 (fetch)
devserv ssh://devserv.example.com/home/vasya/prog1 (push)
origin  /home/vasya/work/prog1 (fetch)
origin  /home/vasya/work/prog1 (push)
vasya@vbook:~/TMP/prog1$
```

Здесь уже можно предположить, что данный репозиторий был создан как копия репозитория, находящегося на этой же машине и принадлежащего, судя по всему, тому же пользователю `vasya`, а позже здесь был зарегистрирован ещё один удалённый репозиторий, на сей раз находящийся на сервере `devserv.example.com`. Загадочные слова `fetch` и `push` — это команды, позволяющие получить новые изменения из удалённого репозитория и, наоборот, отправить свои изменения «на тот конец». Дело здесь в том, что `git` позволяет использовать разные способы доступа к одному и тому же репозиторию, и здесь в выдаче отражён тот факт, что для каждого из удалённых репозиториев один и тот же адрес используется как для получения информации, так и для её отправки.

Добавить новый удалённый репозиторий в этот список можно с помощью команды `git remote add`; например, `devserv` из предыдущего примера мог быть добавлен так:

```
git remote add devserv ssh://devserv.example.com/home/vasya/prog1
```

Указать, где находится добавляемый репозиторий, можно теми же способами, которые применяются для `git clone` (два из которых мы рассмотрели). Удалённый репозиторий можно переименовать с помощью `git remote rename` и удалить из списка с помощью `git remote rm`; есть и другие команды для манипуляции этим списком.

Вторая концепция, которую нужно понять — это ***удалённые ветки*** (*remote-tracking branches*). Для каждого удалённого репозитория `git`, когда у него есть такая возможность (то есть при получении «оттуда» информации), создаёт (локально) ветки, соответствующие состоянию веток удалённого репозитория. Эти ветки имеют «хитрые» имена, записываемые через слэш, что-то вроде «`remotes/origin/master`» или «`remotes/devserv/fixup21`»; первый вариант соответствует ветке `master` из удалённого репозитория `origin`, второй — ветке `fixup21` из репозитория `devserv`. Удалённые ветки не показываются командой `git branch`, если её об этом специально не попросить, а попросить её можно флажком `-a` (от слова *all*):

```
vasya@host:~/TMP/prog1$ git branch -a
* master
  remotes/devserv/HEAD -> devserv/master
  remotes/devserv/fixup21
  remotes/devserv/fixup15
  remotes/devserv/master
  remotes/origin/HEAD -> origin/experiments
  remotes/origin/doubtful_changes
  remotes/origin/experiments
  remotes/origin/master
vasya@host:~/work/prog1$
```

Как ни странно, в ходе обычной работы слово `remotes` в названии удалённых веток не пишут, сокращая название до имени репозитория и имени ветки в нём, записанных через слэш; зачем там это слово вообще появилось — вопрос из области философии.

Удалённые ветки отличаются от обычных тем, что в них нельзя ничего регистрировать, то есть делать в них `commit`. К удалённой ветке можно применить `git checkout`, чтобы посмотреть, как выглядит рабочая директория в той или иной ветке удалённого репозитория, только при этом лучше не пытаться вносить изменения в файлы[2] и регистрировать их в репозитории, поскольку на самом деле *текущей* удалённая ветка не становится, вместо этого у вас вообще исчезает текущая ветка. Впрочем, основная функция удалённых веток скорее идентификационная, то есть состоит в том, чтобы мы могли при необходимости сослаться на набор изменений, входящих в ту или иную ветку удалённого репозитория.

Ещё одно понятие, создающее изрядную путаницу, по-английски звучит как *tracking branch*, что можно перевести примерно как «отслеживающая ветка». Из-за слова *tracking* эти ветки часто путают с удалёнными ветками, которые мы только что обсуждали, поскольку в их английском названии это слово тоже есть[3]. На самом деле «отслеживающие ветки» никого и ничего не отслеживают. Всё их отличие от обычных веток состоит в том, что для такой ветки в конфигурации репозитория указана некая удалённая ветка, связанная с данной, или, если угодно, «вышестоящая» (англ. *upstream*) для данной. Проявляется эта связь исключительно в том, что в некоторых командах, подразумевающих обмен данными с удалёнными репозиториями, можно опустить имя ветки, которое должно использоваться для «того конца».

При клонировании репозитория локально создаётся, как уже говорилось, всего одна ветка, имя которой соответствует имени *активной* ветки клонируемого оригинала (чаще всего это ветка `master`). Введя понятия удалённых и отслеживающих веток, мы к этому утверждению можем добавить, что единственная создаваемая ветка сразу же по умолчанию становится «отслеживающей» для той ветки, которая послужила ей оригиналом, то есть чаще всего (а именно — если в оригинальном репозитории активной была ветка `master`), при клонировании получается ветка `master`, для которой `origin/master` выступает в качестве «вышестоящей». Можно создать локальные аналоги и для других веток исходного репозитория, например, так:

```
vasya@host:~/TMP/prog1$ git branch --track experiments origin/experiments
Branch experiments set up to track remote branch experiments from origin.
vasya@host:~/TMP/prog1$
```

Узнать, для каких веток заданы «вышестоящие» ветки (и какие), можно с помощью команды «`git branch -vv`» (две буквы `v` означают «очень подробно»).

Для получения свежей информации из удалённых репозиториев служит команда `git fetch`. У этой команды много всевозможных флажков, параметров и опций, но в большинстве случаев достаточно просто указать имя удалённого репозитория (дать команду вроде «`git fetch origin`») или флаг `--all`, чтобы скачать всё новое из *всех* удалённых репозиториев, указанных в локальных настройках. Важно понимать, что **команда `git fetch` только скачивает сведения о свежих изменениях из удалённых репозиториев в локальный** и больше ничего не делает. Она не меняет ни файлы в вашей рабочей директории, ни содержимое вашего индекса, ни содержимое локальных веток, в том числе «отслеживающих». Иначе говоря, в вашем репозитории появляется информация об изменениях, зарегистрированных в других репозиториях, но вы вполне можете эту информацию *не*

[2] Конечно, никто не может помешать вам редактировать файлы. Регистрация внесённых изменений при этом теоретически возможна, но имеет целый ряд особенностей, подробное обсуждение которых заняло бы много места. Поскольку всё равно так обычно не делают, мы эту возможность обсуждать не будем. Если очень интересно, поищите нужные материалы самостоятельно по ключевой фразе «`detached HEAD state`».

[3] Мы перевели термин «*remote-tracking branch*» словосочетанием «удалённая ветка»; этот перевод, разумеется, неточен, поскольку вообще никак не учитывает слово «*tracking*», но более точный перевод, который при этом не был бы до крайности косноязычным, нам ни придумать, ни найти не удалось.

заметить. Конечно, `git fetch` меняет состояние *удалённых веток* (то есть веток с именами наподобие `origin/master` или `devserv/fixup21`), но на эти ветки в ходе обычной работы вполне можно не обращать внимания.

Чтобы увидеть изменения, «приехавшие» из других репозиториев, потребуется в явном виде затребовать их внесение в рабочую копию; соответствующее действие нам уже знакомо по описанию системы CVS и называется *слиянием* (англ. *merge*). Команда, выполняющая это действие, так и называется `git merge`. Возможности этой команды довольно широки, но в подавляющем большинстве случаев используют её ровно одним способом. Для начала нужно убедиться в том, что ни в вашей рабочей директории, ни в индексе нет изменений, которые ещё не зарегистрированы в репозитории; если такие изменения есть (`git status` сообщает об их наличии), их следует либо зарегистрировать (сделать `commit`), либо отменить. Затем, если вы находитесь не в той ветке, в которую собираетесь «вливать» изменения, нужно сделать соответствующий `git checkout`, чтобы нужная ветка стала активной (текущей). После этого вы можете присоединить к текущей ветке изменения, имеющиеся в другой ветке, *в том числе удалённой*. Выглядит это примерно так:

```
vasya@host:~/TMP/prog1$ git fetch --all
Fetching origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From /home/vasya/work/prog1
   f23c5f2..908687f  experiments -> origin/experiments
vasya@host:~/TMP/prog1$ git branch
* experiments
  master
vasya@host:~/TMP/prog1$ git merge origin/experiments
Updating f23c5f2..908687f
Fast-forward
 hello.c    |    2 +-
 second.txt |    1 +
 2 files changed, 2 insertions(+), 1 deletions(-)
vasya@host:~/TMP/prog1$
```

Эта процедура может показаться довольно сложной для очевидно рутинного действия, и здесь как раз может пригодиться рассмотренное ранее свойство *отслеживающих* веток. Если ветка, в которой мы работаем, настроена как отслеживающая ту удалённую ветку, из которой мы хотим получить изменения, то мы можем не делать ни `git fetch`, ни `git merge`, для которой необходимо довольно громоздкое указание имени удалённой ветки; со всей этой работой справится одна команда `git pull`. Как водится, эта команда позволяет указывать всевозможные сложные комбинации параметров, чтобы потом просто вызвать `git fetch` и `git merge`, но когда такие параметры нужны, пожалуй, проще будет вызвать эти две команды в явном виде, нежели вникать, как именно их вызовет `git pull`; совсем другое дело — `git pull` без параметров при условии, что текущая ветка настроена как отслеживающая. В этом случае `git pull` для начала затребует свежие данные из репозитория, в котором находится вышестоящая ветка, а затем вызовет `merge` для внесения изменений в нашу текущую ветку и заодно в рабочую копию.

Для привыкших к системе CVS отметим, что `git pull` делает именно то, чего вы привыкли ожидать от `cvs update` — отражает в рабочей копии изменения, которые кто-то другой внёс в одну с вами ветку.

Команда `git push` позволяет передать данные в обратную сторону, то есть отправить изменения, накопленные в локальном репозитории, в один из удалённых репозиториев. Проще всего пользоваться этой командой, если вы работаете в отслеживающей ветке и хотите отправить изменения в вышестоящую ветку; в этом случае `git push` можно запустить без параметров. Впрочем, полезно знать, что у вас есть возможность отправить текущую версию любой из локальных веток в любой из удалённых репозиториев, причём «на том конце» воспользоваться любой из имеющихся веток или создать новую. Например, команда

```
git push boss_computer crazy_experiment:stable_version
```

отправит содержимое вашей локальной ветки с именем `crazy_experiment` в удалённый репозиторий `boss_computer`, где это содержимое будет отражено в ветке, имеющей имя `stable_version`; если такой ветки там пока нет, она будет создана.

С командой `push` связано одно довольно неочевидное ограничение: `git` по умолчанию отказывается выполнять `push` в ветку, которая на удалённом репозитории является текущей. Обусловлено это тем, что в результате такого `push` версия файлов, находящихся в рабочей директории удалённого репозитория, окажется устаревшей в сравнении с тем, что зарегистрировано в самом репозитории, а это может повлечь довольно странные эффекты (особенно странные для человека, который не знает или не понимает, что́ в действительности произошло). Решений здесь возможно несколько. Если у вас есть два репозитория и вы из одного в другой часто передаёте информацию с помощью `push`, можно в этом втором репозитории завести специальную неиспользуемую ветку, назвав её, например, `dummy`, `fake`, `parking` или как-то ещё, и *всегда оставлять репозиторий переключённым на эту ветку*. С тем же успехом можно оставлять репозиторий в состоянии «detached HEAD», например, с помощью команды `git checkout HEAD^0` (к сожалению, у нас нет возможности подробно объяснить, что и почему при этом происходит; если будете так делать, постарайтесь не забыть, что данный репозиторий находится в «безголовом» состоянии, и перед очередным сеансом работы обязательно верните его в одну из существующих веток).

Более «штатное» решение состоит в том, чтобы в настройках репозитория, куда делается `push`, в явном виде разрешить принимать изменения в текущую (активную) ветку. Делается это так:

```
git config receive.denyCurrentBranch warn
```

После этого отправка изменений в активную ветку такого репозитория вызовет предупреждение, но благополучно сработает. Предупреждение будет выглядеть примерно так:

```
remote: warning: updating the current branch
```

Увидев его, не забудьте исправить некорректное положение вещей; сделать это довольно просто: нужно зайти в рабочую директорию удалённого репозитория и дать там страшновато выглядящую команду «`git reset --hard HEAD`». На самом деле ничего особо страшного в этой команде нет, она предписывает `git`'у привести файлы в рабочей директории в соответствие с состоянием `HEAD`, не обращая внимания на любые несоответствия. Впрочем, здесь тоже есть о чём беспокоиться: необходимо точно знать, что в рабочей директории не осталось изменений, не зарегистрированных в репозитории, но при этом нужных — их-то вы, очевидно, потеряете.

«Совсем штатное» решение — завести для обмена данными репозиторий специального типа, так называемый `bare repository`, который, кратко говоря, представляет собой репозиторий без рабочей директории. Этот подход тоже имеет свои ограничения, к тому же такие репозитории — это одна из тех возможностей `git`, которые хотелось бы оставить за рамками обсуждения в нашей книге. Так или иначе, возможны достаточно сильно различающиеся варианты построения рабочего процесса с использованием `git`, и от избранного варианта зависит, как правильнее будет решить техническую заминку с `push`.

Следует отметить, что для выполнения команды `push` у вас должен быть доступ к удалённому репозиторию на запись. Если это ваш собственный репозиторий, проблем с записью в него, скорее всего, не будет, но у вас вряд ли получится отправить свои изменения в репозиторий, принадлежащий кому-то другому, пусть даже это ваш коллега по команде разработки и вообще лучший друг. Теоретически это можно обойти, если установить на файлы и директории, составляющие репозиторий, соответствующие права доступа, но, как показывает опыт, работать в репозитории, в который в любой момент кто-то из коллег может что-то залить, оказывается довольно неудобно.

Некоторые коллективы решают эту проблему, создав один центральный «разделяемый» репозиторий (обычно как раз `bare`, то есть не имеющий своей рабочей директории), который используется всеми участниками разработки для обмена данными между собой. Делается это примерно так же, как при создании централизованного репозитория для CVS и других централизованных систем (см. описание на стр. 220). Обычно в таких случаях участники проекта создают свои рабочие репозитории в виде клонов центрального репозитория, а рабочий процесс в целом начинает напоминать работу с централизованной системой контроля версий.

Мы возьмём на себя смелость порекомендовать другую схему организации рабочего процесса, в которой участникам разработки нет нужды отправлять изменения в *чужие* репозитории. Для реализации этой схемы потребуется Unix-машина, доступная по `ssh` всем участникам разработки; иметь такую машину в любом случае очень удобно и, как мы видели, при использовании схемы с центральным репозиторием наличие та-

кой системы также весьма желательно. Здесь и далее мы будем называть эту машину «сервером разработки».

Каждый участник работы использует по меньшей мере *два* репозитория: один — на сервере, второй — на своей рабочей машине; здесь важно то, что именно он сам выступает в роли *владельца* обоих своих репозиториев, то есть репозиторий на сервере он заводит под своей учётной записью и в своей домашней директории. Конечно, ограничиваться этим разработчик не обязан, репозиториев на своих машинах он может наделать сколько угодно, да и на сервере, в принципе, может создать их больше одного (хотя от этого польза уже весьма сомнительна).

Репозиторий на сервере заводится как клон какого-то другого репозитория на сервере — например, как клон репозитория, принадлежащего руководителю разработки. Права на репозитории выставляются таким образом, чтобы сделать их доступными для чтения всем участникам разработки (например, можно объединить их в одну группу и дать права на чтение этой группе); доступ на запись в свой репозиторий каждый разработчик оставляет только для себя. Репозиторий на личной рабочей машине заводится клонированием *своего* репозитория с сервера.

В репозитории, расположенном на личной машине, создаётся список удалённых репозиториев, в который включаются репозитории других разработчиков, находящиеся на сервере. Свои результаты работы каждый разработчик отправляет с помощью `git push` в *свой* репозиторий на сервере, откуда их могут забрать другие разработчики. Как-то специально организовывать обмен данными между репозиториями на сервере не обязательно, поскольку все изменения в любом случае в итоге попадут в каждый из них; в самом деле, «втянув» изменения из серверных репозиториев, принадлежащих коллегам, в свой рабочий репозиторий, разработчик рано или поздно решит отправить *свои* изменения в свой репозиторий на сервере, но при этом туда будут переданы также и те изменения, которые были ранее загружены из репозиториев других разработчиков. Впрочем, поскольку в такой схеме репозитории на сервере вполне могут быть обычными, то есть иметь свои рабочие директории, при желании разработчики могут «таскать» изменения друг у друга также и на сервере.

Несомненное достоинство этой схемы состоит в отсутствии необходимости давать права записи в один и тот же репозиторий всем разработчикам сразу. Содержимое разделяемой директории всегда находится под угрозой порчи, пусть и непреднамеренной; в нашей схеме каждый может испортить только свой собственный репозиторий, что далеко не столь страшно, поскольку вся информация, содержащаяся в нём, также имеется в репозиториях других разработчиков, причём как в серверных, так и в рабочих.

В заключение обсуждения групповой работы с `git` дадим один, возможно, неожиданный совет. В Интернете можно найти множество сайтов, предлагающих бесплатный сервис хостинга git-репозиториев — BitBucket, GitHub, SourceForge и многие другие. Несмотря на их популярность, мы возьмём на себя смелость рекомендовать воздержаться от использования этих и любых других подобных сервисов, насколько это возможно. Если у вас нет возможности получить в своё распоряжение unix-машину с глобально-видимым ip-адресом, арендуйте для своей команды VPS (виртуальный частный сервер); стоимость аренды VPS начинается от двух-трёх долларов в месяц, а обслуживать такой сервер может не один десяток проектов, то есть вы можете пригласить к себе всех знакомых, занимающихся разработкой программ, сколько бы их у вас ни было, и мощности одного сервера хватит на всех.

Если такое предостережение покажется вам странным, поищите в Интернете статью автора книги, озаглавленную «Осторожно, частный сервис»; там вы найдёте подробное изложение причин, по которым следует относиться с осторожностью к любым «бесплатным сервисам», предлагаемым в Интернете.

Рассмотренных возможностей `git` вам хватит, чтобы начать работу с этой системой, но для серьёзного использования желательно знать о ней гораздо больше. Так, мы не рассмотрели ни средства просмотра журналов изменений, ни разнообразные способы идентификации отдельных «коммитов», ни загадочную на первый взгляд, но очень важную и нужную команду `git rebase`; остались за кадром и многие другие возможности. К счастью, в Интернете в изобилии представлены всевозможные руководства, обучающие тексты, примеры и прочие материалы, посвящённые системе `git`; не пренебрегайте возможностями самостоятельного изучения!

Заключение

Наше «введение в профессию» окончено, но если вы читали эту книгу не зря, то для вас сейчас всё только начинается.

Оказавшись в гуще IT-индустрии, вы обнаружите, что большинство людей, работающих там, совершенно не разделяют многое из сказанного в нашей книге. В общем случае это нормально, ведь каждый имеет право на убеждения; кстати, это касается и лично вас, а ваши знания и навыки к нынешнему моменту уже достаточны — пора формировать свою собственную позицию, которая вполне может по многим вопросам не совпадать с позицией автора книги, этой или любой другой.

Здесь важно одно: помните, насколько иногда бывает тонка грань между личным мнением и попранным здравым смыслом. К сожалению, с этим в нынешней индустрии всё довольно печально: временами кажется, что здесь буквально болван на олухе сидит и придурком погоняет. Люди ухитряются в упор не видеть очевидного, из множества решений едва ли не всегда выживает самое уродливое, новые версии программ часто — *слишком* часто — оказываются хуже предыдущих, технологии деградируют, но именно эту деградацию почему-то называют техническим прогрессом.

К сожалению, вам вряд ли удастся избежать плотного взаимодействия и с самими дураками, и с результатами их деятельности. Не всегда стоит увольняться с работы только потому, что начальник попался настоящий остолоп; на новой работе начальник может оказаться ещё хуже, да и вообще, за хорошую зарплату иногда вполне можно смириться с некоторыми трудностями. Только убедитесь, что ваша зарплата *действительно хорошая*; профилактическая смена места работы раз в несколько лет весьма способствует росту доходов, если только не делать этого слишком часто. Но не стоит ждать, что на новой работе не окажется дураков; скорее всего, вы будете разочарованы.

Помните одно: **для того, чтобы эффективно работать с дураками и среди дураков, совершенно не обязательно становиться дураком самому**. Становиться дураком вообще не надо. Никогда.

И да сопутствует вам удача!

Литература

[1] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug 1978. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 - 1985), М.: МИР, 1993.

[2] R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 – 1985), М.: Мир, 1993.

[3] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, Oct 1972. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 - 1985), М.: МИР, 1993.

[4] D. D. Spinellis. *Programming paradigms as object classes: a structuring mechanism for multiparadigm programming.* PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

[5] T. S. Kuhn. *The structure of scientific revolutions.* University of Chicago Press, Chicago, second, enlarged edition, 1970. Русский перевод: Т. Кун, Структура научных революций. М.: Прогресс, 1997.

[6] D. W. Barron. *An introduction to the study of programming languages.* Cambridge University Press, Cambridge, London, New York, Melbourne, 1977. Русский перевод: Д. Баррон, Введение в языки программирования. М.: МИР, 1980.

[7] T. Budd. *An introduction to object-oriented programming.* Addison Wesley, Reading, Massachusets, 1997. Русский перевод: Т. Бадд, Объектно-ориентированное программирование в действии. СПб.: Питер, 1997.

[8] B. Stroustrup. *The design and evolution of C++.* Addison-Wesley, Reading, Massachusets, 1994. Русский перевод: Бьерн Страуструп, Дизайн и эволюция языка C++, М.: ДМК, 2000.

[9] G. Booch. *Object-oriented Analyses and Design.* Addison-Wesley, Reading, Massachusets, second edition, 1994. Русский перевод: Г. Буч, Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / М.: «Издательство Бином», 1999.

[10] J. Alger. *C++ for real programmers.* AP Professional, Boston, 1998. Русский перевод: Джефф Элджер, C++: библиотека программиста. СПб., Питер, 2001.

[11] J. McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8), Aug 1978. Online: `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`

[12] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, Apr 1960.

[13] Herbert Stoyan. Early LISP history (1956 – 1959) In: LFP '84 Proceedings of the 1984 ACM Symposium on LISP and functional programming. Austin, Texas, USA. August 06–08, 1984. Pg. 299–310

[14] Paul Graham. Revenge of the Nerds. 2002. `http://www.paulgraham.com/icad.html`

[15] Harold Abelson, Gerald Jay Sussman, Julie Sussman. Structure and Interpretation of Computer Programs. Second Edition. MIT Press, Cambridge, Massachusetts, 1996. `https://mitpress.mit.edu/sites/default/files/sicp/index.html`

[16] Chaitanya Gupta. Reader Macros in Common Lisp. 2014. `https://lisper.in/reader-macros`

[17] Mattew Might. Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines. `http://matt.might.net/articles/programming-with-continuations-exceptions-backtracking-search-threads-generators-coroutines/`

[18] Henry G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. ACM Sigplan Notices 30, 9 (Sept. 1995), 17-20. `http://home.pipeline.com/~hbaker1/CheneyMTA.html`

[19] John C. Reynolds. The Discoveries of Continuations. Lisp and Symbolic Computation: An International Journal, 6, 233-247, 1993.

[20] A. Calmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983. Русский перевод см. в: Логическое программирование. Сборник статей. М.: МИР, 1988.

[21] David H. D. Warren. An abstract Prolog instruction set. Menlo Park, CA, USA: Artificial Intelligence Center at SRI International, October 1983. URL: `http://www.ai.sri.com/pubs/files/641.pdf`

[22] Hassan Aït-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Massachusetts, 1991. См. также: `http://wambook.sourceforge.net/`

[23] Братко И. Программирование на языке Пролог для искусственного интеллекта. М.: Мир, 1990.

[24] A. J. Field and P. G. Harrison. Functional Programming. Addison-Wesley, Reading, Massachusets, 1988. Русский перевод: Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993. *К сожалению, качество данного перевода оставляет желать много лучшего.*

[25] Mike Vanier. The Y Combinator (Slight Return) or How to Succeed at Recursion Without Really Recursing. 2008. `https://mvanier.livejournal.com/2897.html`

[26] Ross Paterson. A Hope Interpreter — Reference. April 18, 2000.

[27] John K. Ousterhout. Tcl: An Embeddable Command Language. In: Proceedings of Winter USENIX Conference, 1990, pp. 133-146. *Draft:* `https://web.stanford.edu/~ouster/cgi-bin/papers/tcl-usenix.pdf`

[28] Ray Brigleb. The X Window User HOWTO. v2.0, 1999. Chapter 8: The X Resources. `http://linuxdocs.org/HOWTOs/XWindow-User-HOWTO-8.html`

[29] Richard Stallman. Why you should not use Tcl. Newsgroup comp.lang.tcl, 23.09.1994. `Id: 9409232314.AA29957@mole.gnu.ai.mit.edu` `https://vanderburg.org/old_pages/Tcl/war/0000.html`

[30] Glenn Vanderburg. The Tcl War. (message archive) 1995. `https://vanderburg.org/old_pages/Tcl/war/`

[31] John Ousterhout. "Re: Why you should not use Tcl". Newsgroup comp.lang.tcl, 26.09.1994. `Id: 367307$1un@engnews2.Eng.Sun.COM`. `https://vanderburg.org/old_pages/Tcl/war/0009.html`

[32] John Ousterhout. Scripting: Higher Level Programming for the 21st Century. IEEE Computer magazine, 1998.

[33] Douglas Crockford. JavaScript: The World's Most Misunderstood Programming Language. 2001. `http://www.crockford.com/javascript/javascript.html`

[34] Brian Cantwell Smith. *Procedural Reflection in Programming Languages.* Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Massachusetts Institute of Technology. February 1982. `http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf`

[35] J. Malenfant, M. Jacques and F.-N. Demers. *A Tutorial on Behavioral Reflection and its Implementation.* Département d'informatique et recherche opérationnelle, Université de Montréal, Montréal, Québec, CANADA, 1996. https://www.researchgate.net/publication/243671255_A_Tutorial_on_Behavioral_Reflection_and_its_Implementation

[36] Столяров А. В., Французов О. Г., Аникина А. С. Чистая компиляция как парадигма программирования // Труды Института системного программирования РАН, 2018. Т. 30, № 2. Стр. 7–24.

Предметный указатель

buffer overflow exploit, 2:44

callback-функция, 3:31, 3:93, 3:104

escape-последовательность, 1:28, 1:119, 2:18

glue record, 2:156

IBM PC-совместимый компьютер, 1:172
ip-адрес, 2:148
ip-подсеть, 2:148

JIT-компиляция, 1:75

Linux, 1:31
  дистрибуции, 1:27

MMU, 1:172, 2:113, 2:195, 2:199, 2:201, 2:206
monkey patching, 3:208

NAT, 2:150

Posix Threads, 2:186

RAID, 2:211
REPL, 3:110, 3:133, 3:191

S-выражение, 3:112
System V IPC, 2:125

Unix, 1:30, 1:31

VLA, 2:30

X Window System, 2:96
X-клиент, 2:97
X-протокол, 2:96
X-ресурсы, 3:202
X-сервер, 1:44, 2:97
X-терминал, 2:97

Амдала закон, 2:171

абстракция файла, 2:216
автоматическая очистка, 3:64
адрес, 1:24, 1:170
  вершины стека, 1:191
  виртуальный, 1:171, 2:113
  возврата, 1:94, 1:192
  исполнительный, 1:181
  недействительный, 1:131
  области памяти, 2:13, 2:28
  порта ввода-вывода, 1:26
  сетевой, 2:157
  сокета, 2:157
  физический, 1:171, 2:113, 2:210
  функции, 2:75
  ячейки памяти, 1:74
адресация
  косвенная, 1:181
  относительная, 1:185
  прямая, 1:181
адресная арифметика, 2:28, 2:30
аккумулятор, 1:177
аксессор, 3:51
активное ожидание, 1:207, 2:162, 2:176
алгебра двоичной логики, 1:55
алгоритм, 1:12, 1:59
  Дейкстры, 1:222
  детерминированность, 1:62
  дискретность, 1:62
  завершаемость, 1:62, 1:63
  конечность, 1:62
  массовость, 1:62
  объективность, 1:62
  Петерсона, 2:175
  планирования, 2:215
    CFS, 2:203
    жадный, 2:215
    лифтовый, 2:215
    на основе очереди, 2:202
    предчувствующий, 2:215
    циклический, 2:202
  применимость к входному слову, 1:63
алгоритмическая неразрешимость, 1:63
алфавит, 1:24, 1:57, 1:59
аналоговые ЭВМ, 1:62
арбитр шины, 2:219
аргумент, 3:145
  командной строки, 1:118, 2:111
  команды, 1:34
арифметика адресов, *см.* адресная арифметика
арифмометр
  Лейбница, 1:20
  Паскаля, 1:20
  Шиккарда, 1:20
арность, 2:20, 3:145
ассемблер, 1:75, 1:170
атом, 3:112, 3:148
  в Прологе, 3:145
атомарность, 2:175, 2:176

база данных, 3:161
базис рекурсии, 1:99
байт, 1:53
Беббиджа машина
  аналитическая, 1:21
  разностная, 1:20
бектрекинг, 3:150
бесконечность
  континуальная, 1:57
  счётная, 1:57
библиотека, 1:165, 1:175, 1:217, 2:14
  виджетов, 3:91
  динамическая, 3:207
  стандартная, 2:9, 2:11, 2:14
бит, 1:53, 1:65
битовое поле, 2:49
блок, 2:22, 2:217
  косвенный, 2:217
блок-схема, 1:85
блокировка, 1:205, 2:164, 2:201
бод, 2:134, 2:144
буфер, 2:116, 2:210, 2:213
буферизация, 1:163, 2:11, 2:42, 2:45
  построчная, 2:45

вариадическая функция, 2:39
ввод-вывод, 2:209
  высокого уровня, 2:37
  низкого уровня, 2:46
  форматированный, 2:9, 2:39, 3:129
верхняя половина, 2:197
ветвление, 1:63, 1:85, 1:170, 1:186, 2:22, 3:181
ветка, 3:217
взаимное исключение, 2:172, 2:174
видеопамять, 1:123
виджет, 1:28, 2:87, 3:91, 3:92, 3:193
  активный, 3:93, 3:196
  ввода, 3:93
  вывода, 3:93
  пассивный, 3:93, 3:196
виртуальная машина, 3:207
виртуальный
  адрес, 1:24
  метод, 3:67
витая пара, 2:142
возврат
  значения, 1:95, 1:195, 2:10, 2:17, 2:23, 3:150
  управления, 1:94, 1:171, 2:23
время существования переменной, 2:27
выборка события, 2:163, 3:8, 3:32, 3:87
вывод
  асинхронный, 2:105, 2:214
  синхронный, 2:214
вызов
  макроса, 1:200
  остаточный, 3:22
  подпрограммы, 1:93, 1:170, 1:192
  процедуры, 1:94
  системный, *см.* системный вызов
выполнение, 1:179, 2:201
  одновременное, 1:205
  процесса, 2:201
    в режиме ядра, 2:114, 2:201
выравнивание, 2:48
выравнивающие байты, 2:48
выражение, 2:31
  арифметическое, 1:80, 2:18, 3:155
  критическое, 1:199
  леводопустимое, 2:20, 2:31, 3:44
  логическое, 1:86, 1:87, 1:96
  условное, 3:167
вычисление
  S-выражения, 3:113
  функции, 3:12
вычислительная система, 2:91, 2:212
вычитание с переносом, 1:183

генерализация, 1:100
гибибайт, 1:67
гигабайт, 1:66, 1:67
гипервизор, 2:198
главная часть программы, 1:78, 2:9
главный цикл, 3:8, 3:106, 3:188
голодание, 2:174, 2:181, 2:183, 2:192
готовность
  к выполнению, 1:205, 2:201
  к записи, 2:165
  к чтению, 2:164
граф ожидания, 2:182
группа
  пользователей, 2:99
  процессов, 2:135
    текущая, 2:135
    фоновая, 2:135

данные, 1:22, 3:108, 3:112, 3:149
  бинарные, 1:73, 2:37, 2:46, 3:187, 3:188
  входные, 1:62
  исходные, 1:23
  разделяемые, 2:172, 3:8
  текстовые, 1:70, 1:72, 1:103, 2:32, 3:8
  устаревшие, 2:173
Дейкстры алгоритм, 1:222
дейтаграмма, 2:152
дека, 1:141
декомпозиция, 1:102, 2:51, 3:11
  ложная, 1:103
декорирование имён, 3:40
демон, 2:134, 2:140
дерево
  высота, 1:142
  двоичное, поиска, 1:142
  красно-чёрное, 2:203
  разбалансированное, 1:142
дескриптор
  индексный, 2:100, 2:216
  сегмента, 2:207
  файла, 1:38, 1:211, 2:42, 2:103, 2:104
деструктор, 3:35, 3:39
детализация, 1:100
  пошаговая, 1:100, 1:136
диалект, 1:77, 3:109
дизъюнкция, 1:55
директива, 1:173, 2:9, 3:173
  ассемблера, 1:75
  задания исходных данных, 1:179
  макропроцессора, 1:200, 2:56
  резервирования памяти, 1:179
    неинициализированной, 1:179
  условной компиляции, 1:163, 1:202
  форматная, 2:39
директория, 2:100, 2:101, 2:107, 2:109
диск, 2:93, 2:108
  виртуальный, 2:211
диспетчеризация процессов, 2:202
диспозиция сигнала, 2:112, 2:126, 2:204
дисциплина линии, 2:134
домен, 2:155
  корневой, 2:155
доменная зона, 2:156
драйвер, 1:25, 1:26, 2:93, 2:197, 2:210
  физического устройства, 2:200
«дырки» в файлах, 2:105

единица трансляции, 2:15

журнализация, 2:140

зависимость, 3:209
  взаимная, 1:168
  времени исполнения, 3:209
  времени сборки, 3:209
  односторонняя, 1:168
заглушка, 1:100
заголовок
  класса, 3:49
  модуля, 1:166
  программы, 1:78
  процедуры, 1:94
  функции, 1:95, 2:10, 2:15
загрузочный сектор, 2:198
загрузчик операционной системы, 2:198
задача, 1:205
  о пяти философах, 2:179
  о спящем парикмахере, 2:184
  о читателях и писателях, 2:183
  производителей и потребителей, 2:178
заём из несуществующего разряда, 1:68
замыкание, 3:122, 3:169, 3:175, 3:190, 3:208
запрос прерывания, 1:207
зарезервированное слово, 1:83
затравка, 3:19
защита, 3:36
  памяти, 1:171
звено списка, 2:49
знаковость, 2:16
знаковый бит, 1:219
значение
  возвращаемое, 1:95
  исключения, 3:61
  логическое, 1:86
  переменной, 1:82
    начальное, 1:82
  по умолчанию, 3:48
зомби, 2:118

идентификатор, 1:78, 1:82, 3:180
  группы пользователей, 2:98
  группы процессов, 2:111, 2:136
  пользователя, 2:98
  процесса, 2:111
    родительского, 2:111
  сеанса, 2:111, 2:135
иерархия типов, 3:64
избыточность, 3:38
импликация, 1:55
имя
  библиотеки, 1:218
  доменное, 2:155
  команды, 1:34, 1:42, 1:197
  макроса, 1:174, 1:199, 2:56
  перечисления, 2:25
  структуры, 2:47
  файла, 1:34, 1:125, 2:41, 2:101
    абсолютное, 1:33, 1:125, 2:41, 2:100
    краткое, 1:33
    относительное, 1:33, 1:125
инверсия предикатов, 3:151, 3:153
инициализатор, 2:15
инициализация, 1:82, 2:15, 3:12, 3:37, 3:43
  подобъектов, 3:50
инкапсуляция, 1:166, 1:215, 3:15

инстанциация, 3:75
интерпретатор, 3:108, 3:206
  командной строки, 1:28
интерпретация, 3:30, 3:118, 3:178, 3:184, 3:206, 3:207
  частичная, 3:207
интерфейс
  драйвера, 2:211
  класса, 3:16
  модуля, 1:166
  пользовательский, 1:27
  сетевой, 2:144
информатика, 1:24, 1:47
информация, 1:24, 1:59, 1:65
  отладочная, 1:164, 2:11
исключение, 1:184, 1:208, 1:224, 2:91, 2:196, 3:61
исходный текст, 1:74, 1:77
итерация, 1:87

кадр (памяти), 2:207
канал, 2:125, 2:131
  именованный, 2:125, 2:132
  неименованный, 2:125, 2:131
Кантора диагональный метод, 1:57
карринг, 3:174
каталог, 2:100, 2:101, 2:107, 2:109
  домашний, 1:33
  корневой, 1:33, 2:100, 2:112, 2:198, 2:216
  родительский, 1:33
  страничных таблиц, 2:208
  текущий, 2:100, 2:112
квант времени, 1:206, 2:92
килобайт, 1:66, 1:67
класс, 3:16, 3:35, 3:38
  абстрактный, 3:69
  базовый, 3:65
  дочерний, 3:65
  полиморфный, 3:74
  порождённый, 3:65
  родительский, 3:65
  унаследованный, 3:65
клиент, 2:153, 2:160
ключ поиска, 1:142
ключевое слово, 1:83, 1:86
коаксиальный кабель, 2:141
код
  дополнительный, 1:68
  завершения процесса, 1:43, 1:101, 1:176, 1:209, 2:10, 2:11, 2:117
  исходный, 1:77
  команды, 1:22
  машинный, 1:25, 1:74, 1:174, 2:10
  объектный, 1:165, 1:175, 1:197
  операции, 1:25
  символа, 1:66, 1:103, 1:180, 2:16
  управляющий, 1:71
коллизия (в хеш-таблице), 1:143
кольцо защиты, 1:171, 1:172
команда
  безусловного перехода, 1:185
  встроенная, 1:28
  вызова, 1:171
  декремента, 1:183
  инкремента, 1:183
  командной строки, 1:28
  перехода, 1:25
  привилегированная, 1:171
  условного перехода, 1:185
    по одному флагу, 1:185
командная строка, 1:28
комбинатор неподвижной точки, 3:175
комментарий, 1:173, 1:181, 1:197, 2:9
  строчный, 3:35
коммутатор, 2:142
коммутация пакетов, 2:141
компилятор, 1:74
компиляция, 3:30, 3:83, 3:110, 3:118, 3:133, 3:144, 3:206, 3:207
  частичная, 3:208
  чистая, 3:213
компоновщик, 1:79, 1:165, 1:175
компьютерная сеть, 2:141
конвейер, 1:38, 2:94, 2:132
конвенция CDECL, 1:196
конец файла, *см.* ситуация конца файла
конечный автомат, 2:166, 3:32
конечный пользователь, 1:27
константа, 1:198, 3:112
  восьмеричная, 1:93
  времени исполнения, 1:92
  времени компиляции, 1:92, 2:24, 2:26
  двоичная, 1:93
  именованная, 1:92
  описание, 1:92
  с плавающей точкой, 1:198
  символьная, 1:198
  строковая, 1:78, 1:198, 2:10
  типизированная, 1:93
  целочисленная, 1:198
  шестнадцатеричная, 1:93
конструктор, 3:40, 3:165
  копирования, 3:47, 3:48
  неявный, 3:52
  объекта, 3:35, 3:36
  по умолчанию, 3:42, 3:48
  преобразования, 3:43, 3:48
контейнер, 3:54
  обобщённый, 3:82
контекст
  выполнения процесса, 2:201
  лексический, 3:121
континуация, 3:135, 3:137, 3:208
континуум, 1:57
контроллер, 1:24, 1:26, 2:209
  прямого доступа к памяти, 2:219
конфликт имён, 1:218
концентратор (hub), 2:142
конъюнкция, 1:55
кортеж, 3:148, 3:165
критическая секция, 2:172, 2:174
  модифицирующая, 2:174
куча, 1:178, 2:113, 2:127

лексема, 2:56
лексический анализ, 2:43, 2:56
ленивые вычисления, 1:64, 1:201, 3:13, 3:163
лидер сеанса/группы, 2:136
линкер, 1:175
листинг, 1:197
литерал, 1:92, 2:17
  восьмеричный, 1:93
  двоичный, 1:93
  символьный, 2:18
  строковый, 1:78, 1:115, 2:10, 2:18
  шестнадцатеричный, 1:93
лямбда-исчисление, 3:175
лямбда-список, 3:113, 3:119, 3:135
лямбда-функция, 3:119

макровызов, 1:173, 1:174, 1:200, 2:56
макродиректива, 1:200, 2:56
макроимя, 2:56
макроопределение, 1:200, 2:56
макропеременная, 1:201
макроподстановка, 1:200
макропроцессор, 1:199, 2:56
макрорасширение, 1:200
макрос, 1:199, 2:56, 3:33, 3:113
  многострочный, 1:200
  однострочный, 1:200
малое упрятывание, 1:208
мантисса, 1:69, 1:93, 1:219
маршрутизатор, 2:144
маска подсети, 2:151
массив, 1:111
  ассоциативный, 3:189
  разреженный, 3:56
машинная инструкция, 1:170
машинное слово, 1:67, 1:74, 1:179
мебибит, 1:67
мегабайт, 1:66, 1:67
менеджер
  геометрический, 3:198
  оконный, *см.* оконный менеджер
метапрограммирование, 3:33, 3:75, 3:118, 3:179
метафора рабочего стола, *см.* окружение рабочего стола
метка, 1:101, 1:174, 1:197, 3:40
  внешняя, 1:215
  глобальная, 1:215
  локальная, 1:195
метод, 3:35, 3:36
  виртуальный, 3:67, 3:68
  девятичных дополнений, 1:20
  константный, 3:45
  статический, 3:59, 3:105
  чисто виртуальный, 3:69
микропроцессор, 1:23, 1:172
мнемоника, 1:75, 1:173
модель
  ISO OSI, 2:146
  адресации
    плоская, 1:172
  виртуальной памяти
    база-предел, 2:206
    простейшая, 2:206
    сегментная, 2:206
    сегментно-страничная, 2:209
    страничная, 1:172, 2:207
  вычислительная, 3:113
  клиент-серверная, 2:153, 3:31
модем, 2:134
модификатор разрядности, 2:39
модуль, 1:165, 1:175, 2:15
  объектный, 1:79, 1:175, 2:10
монада, 3:13
монтирование, 2:100, 2:216
мультизадачность, 1:171, 1:172, 1:205, 2:91
  невытесняющая, 1:206, 2:91
  пакетная, 1:206, 2:91, 2:201
  разделения времени, 1:206, 2:91, 2:201
  реального времени, 1:206, 2:91
мультиплексирование, 2:162, 2:164
Мура диаграмма, 2:167
мусор, 1:132
мьютекс, 2:176, 2:177

Ньютона бином, 1:50

набор тестов, 1:161
надкласс, 3:84
наследование, 3:16, 3:64, 3:65
научная нотация, 1:81, 1:93
фон Неймана
  архитектурные принципы, 1:170
  машина, 1:22
нижняя половина, 2:197
номер
  порта, 2:152
  сигнала, 2:125

область
  видимости, 2:27, 3:39, 3:49
  индексных дескрипторов, 2:217
  памяти, 1:25, 1:66, 2:13, 3:45
обработка
  информации, 1:24
  исключений, 3:61, 3:131, 3:136, 3:138, 3:158, 3:182, 3:186
  события, 2:163, 3:8
обработчик
  исключения, 3:62
  прерывания, 1:207, 1:208
  события, 2:163
обратный ход рекурсии, 3:24
обращение
  к переменной, 1:42, 1:82, 3:45, 3:178, 3:180
  к подпрограмме, *см.* вызов
объект, 3:15
  анонимный, 3:47
  временный, 3:43, 3:47
объявление, 2:63, 3:166
  глобальное, 2:15
  функции, 2:15
окно, 3:92
  верхнего уровня, 3:92
  диалоговое, 3:92
  дочернее, 3:92
  модальное, 3:104
оконный менеджер, 1:28, 2:97, 3:92, 3:195
окружение
  виртуальное, 2:199
  процесса, 1:44, 2:111, 2:116
  рабочего стола, 1:32, 1:45, 2:97
октет, 1:66
операнд, 1:173, 1:197
  адресный, 1:181
  непосредственный, 1:181
  неявный, 1:184
  регистровый, 1:181
  типа «память», 1:181
оперативное запоминающее устройство, 1:24
оператор, 1:79, 3:42
  безусловного перехода, 1:101, 2:24
  ввода, 1:83
  ветвления, *см.* ветвление
  возврата, 2:10
  выбора, 1:118, 2:24
  вывода, 1:79
    с переводом строки, 1:78
  вызова процедуры, 1:93, 1:94, 3:27
  вычисления выражения, 2:21, 2:36
  присваивания, 1:82
  пустой, 1:106
  составной, 1:86, 2:7, 2:22
  условный, 1:85, 2:22
  цикла, *см.* цикл
операторные скобки, 1:86, 2:7
операционная система, 1:25, 1:27, 1:77, 1:101, 1:106, 1:124, 1:125, 1:128, 1:131, 1:171, 1:173, 1:178, 1:205, 2:7, 2:9, 2:34, 2:41, 2:88, 2:91, 2:147, 2:148, 2:152, 2:195, 3:7, 3:30, 3:118, 3:178, 3:207
  мультизадачная, *см.* мультизадачность
  пакетная, 1:206
операция
  бинарная, 1:55
  блочного чтения и записи, 1:125
  ввода, 1:83
  взятия адреса, 1:131, 2:13, 2:28
  взятия остатка, 2:18
  вызова функции, 2:20, 3:56
  вычитания адресов, 2:30
  декремента, 2:20
  деления с остатком, 1:81
  записи, 1:24
  индексирования, 2:20
  инкремента, 2:20
  исключающего или, 1:55
  карринга, 3:174
  конкатенации строк, 1:116
  логическая, 1:55, 1:87, 2:19
  отрицания, 1:55
  побитовая, 1:91
  преобразования типа, 2:21, 3:56, 3:73
  присваивания, 2:19
  разрушающая, 3:12, 3:116
  разыменования, 1:131, 2:28, 2:30
  сравнения, 1:86
  унарная, 1:55
  условная, 2:20, 2:36, 3:42, 3:114, 3:167
  целочисленного деления, 1:81
  чтения, 1:24
описание, 3:166
  глобальное, 2:15
  переменной, 1:82
  функции, 2:15
основание системы счисления, 1:52
откат транзакции, 2:183
откачка, 2:92, 2:201, 2:205, 2:206
отладка, 1:160
отладочная печать, 1:163
отладчик, 1:163
отложенная запись, 2:218
отношение, 3:15, 3:147

отображение, 2:113
отрицание, 3:153
отсечение, 3:148, 3:153
Оустерхаута дихотомия, 3:205, 3:212
очередь, 1:136, 1:137
  задач, 1:206
ошибка округления, 1:70

пакет данных, 2:141
память
  ассоциативная, 2:208
  виртуальная, 1:171, 2:113, 2:201, 2:206
  динамическая, 2:30
  оперативная, 1:24, 1:74, 1:170
паравиртуализация, 2:199
парадигма программирования, 3:7
параметр
  выходной, 2:29, 2:70
  значение, 1:96
  ключевой, 3:127
  макроса, 1:174
  накапливающий, 3:23, 3:116, 3:152
  неявный, 3:36
  переменная, 1:97, 1:112, 1:113, 2:13
  подпрограммы, 1:94, 1:192
  фактический, 3:116
  шаблона, 3:75
Паскаля
  арифмометр, 1:20
  треугольник, 1:51
перегрузка имён функций, 3:39
передача управления, *см.* переход
переменная, 1:42, 1:82, 1:174, 2:13, 2:28, 2:31, 3:146
  анонимная, 3:37, 3:149, 3:167
  блокировочная, 2:175
  глобальная, 1:97
  динамическая, 1:130, 1:131
  имя, 1:82
  локальная, 1:94, 1:192, 2:12
  неинициализированная, 1:82
  окружения, 1:44, 1:45, 1:178, 2:111, 2:116
  свободная, 3:119, 3:147
  связанная, 3:147
  типовая, 3:169
  файловая, 1:125
  цикла, 1:89
перенос в несуществующий разряд, 1:20, 1:68, 1:84
переносимость программ, 2:95
переполнение, 1:69
переход, 1:170, 1:175, 1:185
  безусловный, 1:25, 1:171, 1:185
  близкий, 1:185
  дальний, 1:185
  короткий, 1:185
  косвенный, 1:185
  прямой, 1:185
  с запоминанием адреса возврата, 1:94, 1:171, 1:193
  условный, 1:25, 1:171, 1:174, 1:178, 1:185
Петерсона алгоритм, 2:175
плагин, 3:193
планирование
  времени ЦП, 1:206, 2:202
  дисковых обменов, 2:215
  долгосрочное, 2:202
  краткосрочное, 2:202
  среднесрочное, 2:202
планировщик, 2:111, 2:202, 2:203
побитовые операции, 1:187
побитовый сдвиг, 1:187, 2:19
  арифметический, 1:188, 2:19
  циклический, 1:188
побочный эффект, 1:14, 1:94, 1:98, 1:106, 1:120, 2:33, 2:68, 2:205, 3:12, 3:15, 3:26, 3:114, 3:142, 3:163, 3:180
повторновходимость, 3:11, 3:26
повторное включение, 2:64
подкласс, 3:84
подпрограмма, 1:43, 1:91, 1:192
  вариадическая, 1:194
  декларация, 1:143
  описание, 1:143
подстановка имён файлов, 1:35
поиск
  линейный, 1:142
поколение ЭВМ
  второе, 1:23
  первое, 1:22
  третье, 1:23
  четвёртое, 1:24
поле
  записи, 1:113
  статическое, 3:58
  структуры, 2:47, 2:48
полиморфизм, 3:16, 3:83
  ad hoc, 3:84
  адресов, 3:65, 3:84
  динамический, 3:72, 3:84
  параметрический, 3:75, 3:84
  статический, 3:40, 3:42, 3:84
порт
  ввода-вывода, 2:210
  сетевой, 2:152, 2:158
порядок, 1:93
  машинный, 1:69
    смещённый, 1:69, 1:219
постоянное запоминающее устройство, 2:197
поток
  ввода-вывода, 1:36, 1:38, 1:125, 2:95, 3:118, 3:128–3:130, 3:135, 3:157, 3:173
    бинарный, 3:131
    позиционируемый, 1:128, 2:47, 2:95, 2:105, 2:108
    стандартный, *см.* поток стандартный
  стандартный, 1:38, 1:127, 1:210, 3:129, 3:130, 3:135, 3:157
    ввода, 1:210, 3:164
    вывода, 1:209, 1:210
    диагностики, 1:38, 1:127, 1:210
права доступа, 2:102, 2:107
правило
  вывода, 3:15
  обособления, 1:103
  одной фразы, 1:103, 1:167
  универсальности, 1:103
предикат, 3:15, 3:114, 3:145, 3:148
  динамический, 3:161
предложение, 3:149
преобразование типов, 1:115
  неявное, 1:115
  по закону полиморфизма, 3:65
  явное, 1:115
прерывание, 1:207, 2:91, 2:196
  аппаратное, 1:207, 2:91, 2:196
  внешнее, 1:207, 2:91, 2:196
  внутреннее, 1:184, 1:208, 2:91, 2:196
  программное, 1:205, 1:209, 2:91, 2:196
  страничное, 2:207
префикс команды, 1:190, 1:191
префиксное дерево, 2:54
приглашение, 1:28, 1:33
приём соединения, 2:160
признак присутствия, 2:207
принудительная синхронизация, 2:215
принцип хранимой программы, 1:22, 1:170
приоритет
  операции, 1:81, 1:87
  процесса, 1:206, 2:120
    динамический, 2:111
    статический, 2:111
присваивание, 2:19, 3:10, 3:37
проблема
  останова, 1:63
  очерёдности действий, 2:160
  самоприменимости, 1:63
программа, 1:59, 1:170
программирование
  автоматное, 3:32
  в терминах явных состояний, 2:169, 3:32
  декларативное, 3:15, 3:141
  императивное, 3:8, 3:10
  командно-скриптовое, 3:11, 3:178
  логическое, 3:14, 3:141
  многопоточное, 2:171, 2:202
  обобщённое, 3:83
  объектно-ориентированное, 1:64, 3:7, 3:8, 3:15
  параллельное, 3:7, 3:8
  прикладное, 3:7
  процедурное, 3:11
  событийно-управляемое, 2:163, 2:172, 3:7, 3:8, 3:28, 3:31, 3:87, 3:106
  фоннеймановское, 3:11
  функциональное, 3:8, 3:12
продолжение, 3:137
промах, 2:208
пространство
  адресное, 1:24, 1:26, 2:199
    виртуальное, 1:171, 1:178, 2:109, 2:113, 2:206, 2:207, 2:209
    плоское, 1:172
    портов ввода-вывода, 2:210
    физическое, 1:24, 2:210
  имён, 1:165, 1:215, 2:48, 3:34, 3:49
    глобальное, 1:168, 2:64
протокол, 2:146
профиль функции, 2:75, 3:40
процедура, 1:93, 3:11, 3:27, 3:145, 3:149, 3:184
  конструктивная, 1:59
процедурная семантика, 3:15
процесс, 1:33, 1:36, 1:44, 1:205, 2:92, 2:110, 2:200
  init, 2:197
  легковесный, 2:171, 2:202
  спящий, 2:120
прямой ход рекурсии, 3:24
псевдокоманда, 1:198
псевдометка, 1:198
псевдопроцессы, 2:196
псевдотерминал, 2:139
псевдоустройство, 1:38, 2:108, 2:212
путь, 1:33, 1:34
  абсолютный, 2:41

развилка, 3:150
раздел диска, 2:211
разделение времени, 2:91
раздельная трансляция, 1:165
  единица, 1:165
разделяемая память, 2:110, 2:124
разрушающее действие, 3:12, 3:108, 3:116
регистр, 1:25, 1:74, 1:170, 1:177
  аккумулятор, 1:177
  общего назначения, 1:170, 1:177
  сегментный, 1:177
  специальный, 1:177
  указатель
    базы, 1:177
    инструкции, *см.* указатель инструкции
    стека, 1:177, 1:191
  флагов, 1:174, 1:177
регрессия, 3:217
редактор связей, 1:165, 1:175
редукция, 3:19, 3:171
  графа ожидания, 2:183
реентерабельность, 3:11, 3:26
режим
  диалоговый, 1:28
  задачи, 1:171
  канонический, 2:83, 2:137
  мультизадачности, *см.* мультизадачность
  неблокирующий, 2:104, 2:162
  невытесняющий, 1:206
  неканонический, 2:137
  «отсоединённый», 2:186
  пакетный, 1:28, 1:206, 2:91
  разделения времени, 2:91
  супервизора, 1:171
  ЦП
    непривилегированный, 1:207
    ограниченный, 1:171, 1:207, 2:91
    пользовательский, 1:171, 2:91
    привилегированный, 1:171, 1:207, 2:91
реификация, 3:208
рекурсия, 1:61, 1:193, 3:8
  взаимная, 3:19
  высшего порядка, 3:19
  остаточная, 3:13
  параллельная, 3:19
  простая, 3:19
  хвостовая, 3:13
репозиторий, 3:217
ресурсное голодание, 2:215
рефлексия, 3:207

самодостаточность, 3:209
свёртка, 3:19
свободное программное обеспечение, 2:95
связывание
  динамическое, 3:109, 3:117, 3:120
  лексическое, 3:121
сеанс, 2:135
сегмент, 1:173, 1:178, 2:112, 2:206
  данных, 2:16
секция, 1:173, 1:178, 2:16, 2:112, 2:113
  BSS, 1:178
  данных, 1:178
  интерфейса модуля, 1:166
  кода, 1:178
  критическая, *см.* критическая секция
  локальных описаний, 1:94
  неинициализированных данных, 1:178
  описания
    констант, 1:92
    переменных, 1:82
    типов, 1:109
  реализации модуля, 1:166
селектор сегмента, 2:206
семантика, 2:74, 3:44
  декларативная, 1:64, 3:8, 3:142, 3:151–3:153
  ленивая, *см.* ленивые вычисления
  логическая, *см.* семантика декларативная
  пользовательская, 3:66
  процедурная, 3:142
  реализаторская, 3:66
семафор, 2:177
  двоичный, 2:176, 2:177, 2:181
семейство протоколов (адресации), 2:157
сервер, 2:153, 2:160
  приложений, 2:97
сигнал, 1:36, 1:104, 2:112, 2:117, 2:118, 2:120, 2:124, 2:125, 2:204
символ, 2:16
  ASCII, 1:71
  алфавита, 1:24
  в Лиспе, 3:112
  возврата каретки, 1:71, 2:18
  перевода строки, 1:71, 1:73, 2:10, 2:18, 2:43, 2:83
  пробел, 1:71
  раскрытия области видимости, 3:49
  служебный, 2:18
  табуляции, 1:71, 2:18
  управляющий, 1:71
синдром утёнка, 3:205
система
  вычислительная, *см.* вычислительная система
  доменных имён, 2:155
  команд, 1:25, 1:75
  операционная, *см.* операционная система
  счисления
    восьмеричная, 1:53

двоичная, 1:52–1:54
десятичная, 1:52
позиционная, 1:51
шестнадцатеричная, 1:53
файловая, *см.* файловая система
системные часы, 2:119
системный вызов, 1:171, 1:205, 1:208, 2:14, 2:87, 2:91, 2:195, 2:196, 3:7
ситуация
гонок (состязаний), 2:115, 2:173, 3:130
конца файла, 1:36, 1:106, 1:129, 1:175, 1:210, 2:37, 2:135, 3:128–3:130, 3:135, 3:157, 3:188
сквозная запись, 2:218
скрипт, 1:29, 1:42, 3:109, 3:143, 3:173, 3:178, 3:179, 3:182, 3:195, 3:203, 3:204
стартовый, 2:198
скриптинг, 3:203, 3:212
сложение с переносом, 1:183
событие, 2:163, 2:164, 3:86
клавиатурное, 2:135
совершенная дизъюнктивная нормальная форма, 1:57
соединение, 2:160
сокет, 2:125, 2:156
слушающий, 2:160
сопоставление с образцом, 3:167
сопроцессор, 1:219
состояние, 1:60, 2:110, 2:166, 3:10
блокировки, 1:205
покоя, 2:195
специализация
частичная, 3:78
явная, 3:78
спецификатор
размера, 1:183
список, 3:112, 3:181
двусвязный, 1:140, 2:52
звено, 1:132
неправильный, 3:112
односвязный, 1:132, 2:49
пустой, 3:112
точечный, 3:112, 3:146
формальных параметров, 1:94
циклический, 3:117, 3:147, 3:170
ссылка, 3:43
висячая, 2:101
жёсткая, 2:101, 2:106, 2:216
неопределённая, 1:218
символическая, 2:101, 2:107
стек, 1:136
протоколов, 2:143
стековый фрейм, 1:169, 1:193, 3:13, 3:21
страница, 2:207
виртуальная, 2:201
стратегия планирования, 2:215
строка, 2:18, 2:32
форматная, 2:12, 3:129
структурные отступы, 1:80
суперблок, 2:100, 2:217
суперкласс, 3:84
суперкомпьютер, 1:24
суперпользователь, 2:98
суффикс
доменного имени, 2:155
имени файла, 1:35, 1:77
сцепленность модулей, 1:168
счётчик
команд, 1:74, *см.* указатель инструкции
цикла, 1:88, 1:186

таблица, 1:114, 3:15, 3:148
виртуальных методов, 3:68
дескрипторов, 2:115
прерываний, 1:208
сегментов, 2:207
файловых, 2:112
истинности, 1:56
страниц, 2:207
страничная
многоуровневая, 2:208
тавтология, 1:56
тайм-аут, 2:86
таймер, 1:207
тасклет, 2:197
текст, 1:24
текстовый редактор, 1:77
тело
макроса, 1:200
процедуры, 1:94
функции, 1:95, 2:15
цикла, 1:87
терабайт, 1:66
терм, 3:144
сложный, 3:145
терминал, 1:28, 2:37, 2:45, 2:125, 2:133
виртуальный, 2:125
управляющий, 2:135
тест, 1:161
тестирование, 1:161
тип
анонимный, 1:109, 1:111
взаимодействия, 2:157
встроенный, 3:165
выражения, 1:81, 1:82
данных
абстрактный, 3:16
составной, 2:47
исключения, 3:61
неполный, 2:65
параметрический, 3:166
переменной, 1:82
перечислимый, 1:110, 2:25, 3:165
пользовательский, 1:109, 3:165
порядковый, 1:90, 1:110
структурный, 2:47
устройства, 2:211
файла, 2:108, 2:159
файловый, 1:125
точечная пара, 3:112, 3:146
точка
входа, 1:174, 1:175, 2:210
драйвера, 2:211
монтирования, 2:100, 2:216
останова, 1:164
отмены, 2:187
следования, 2:36
точность представления, 2:39
трамплин, 3:124
транзакция, 2:183
транзистор, 1:23
полевой, 1:23
трансляция сетевых адресов, 2:150
трассировка, 2:125, 2:133
тред, 1:18, 2:171, 2:186, 2:202
трит, 1:66
тупик, 2:179, 2:180, 3:14
Тьюринга
машина, 1:60
универсальная, 3:108
тезис, 1:60

узел дерева, 1:142
указатель, 1:130
инструкции, 1:25, 1:74, 1:170, 1:177
на массив, 2:75
на функцию, 2:75
нетипизированный, 1:131
нулевой, 1:131
стека, 1:177, 1:191, 1:192
типизированный, 1:130
унификация, 3:146, 3:147
Уоррена машина, 3:142
управление, 1:75, 1:170
программное, 1:21
устройствами, 2:209
условие, 1:85, 1:86
устройство
блочное, 2:108, 2:200, 2:211
виртуальное, 2:95, 2:211
внешнее, 2:209
логическое, 2:211
периферийное, 1:74
потоковое, 2:108, 2:211
символьное, 2:108, 2:211
утечка памяти, 1:132

файл, 1:25, 1:27, 2:93, 2:100, 2:108
заголовочный, 2:10, 2:63
исполняемый, 1:79
нетипизированный, 1:128
размер блока, 1:128
объектный, 1:79
текстовый, 1:126
типизированный, 1:125, 1:127
устройства, 1:38, 2:108, 2:217
файловая система, 2:100, 2:101, 2:216
виртуальная, 2:200, 2:212, 2:216
искусственная, 2:216
корневая, 2:100, 2:216
факт, 3:14, 3:149, 3:161
факториал, 1:49
фильтр (программа), 1:106
флаг, 1:178
фокус ввода, 3:96
форма
верхнего уровня, 3:115
вычислимая, 3:113
специальная, 3:113, 3:190
фунарг-проблема, 3:109
восходящая, 3:121, 3:124, 3:125
нисходящая, 3:121, 3:124
функтор, 3:56, 3:145
главный, 3:145
функционал, 3:119
функция, 1:93, 1:95, 3:12, 3:113
вариадическая, 2:39, 2:81, 3:119, 3:135
виртуальная, 3:67, 3:68
высшего порядка, 3:119
вычислимая, 1:59
дружественная, 3:51
математическая, 1:24
нестрогая, 3:172
полиморфная, 3:169
примитивно-рекурсивная, 1:64
частично-рекурсивная, 1:59
чисто виртуальная, 3:69
член класса, 3:35

хакер, 2:44
хеш-таблица, 1:142, 3:112
хост, 2:148

целостность, 3:38
данных, 2:173
состояния, 1:168
цель, 3:149, 3:150
центральный процессор, 1:24, 1:25, 1:74, 1:170
цикл, 1:87, 1:170
арифметический, 1:89, 2:23
бесконечный, 1:63, 2:23, 3:14
вложенный, 1:90
главный, 3:28
обработки команд, 1:25, 1:74, 1:170
с постусловием, 1:89
с предусловием, 1:88, 1:186, 2:22, 3:183
цифры
арабские, 1:51, 1:52, 1:71
двоичные, 1:53
шестнадцатеричные, 1:53

чередование, 2:175
Чёрча тезис, 1:60
число
денормализованное, 1:220
псевдослучайное, 1:124, 2:80
с плавающей точкой, 1:66, 1:69, 1:81, 1:93, 2:11
случайное, 2:80
целое, 1:66, 2:9

шаблон, 3:75
шаг автомата, 3:32
шина, 1:24, 1:26, 1:74
адресов, 1:24, 1:66, 1:172
данных, 1:24
управления, 1:24
шлюз, 2:144

экспорт, 1:165
элемент массива, 1:111

ядро операционной системы, 1:205, 2:7, 2:90, 2:91, 2:93, 2:195, 3:204
язык
алгоритмически полный, 1:71
ассемблера, 1:75
низкого уровня, 1:18
программирования, 1:70
императивный, 1:64
компилируемый, 1:77
общего назначения, 3:204
предметно-ориентированный, 3:109, 3:204, 3:212
строго типизированный, 1:82
разметки, 1:71
Тьюринг-полный, 1:71
фоннеймановский, 1:18, 1:64
формальный, 1:70
ячейка памяти, 1:24, 1:74, 1:170, 1:171

http://www.stolyarov.info

Учебное издание

СТОЛЯРОВ Андрей Викторович

ПРОГРАММИРОВАНИЕ: ВВЕДЕНИЕ В ПРОФЕССИЮ
в трёх томах
Издание 2,92 (электронное), с исправлениями
Том III: ПАРАДИГМЫ

Рисунок и дизайн обложки *Елены Доменновой*

Официально опубликовано 30 апреля 2026 г.
на сайте http://www.stolyarov.info