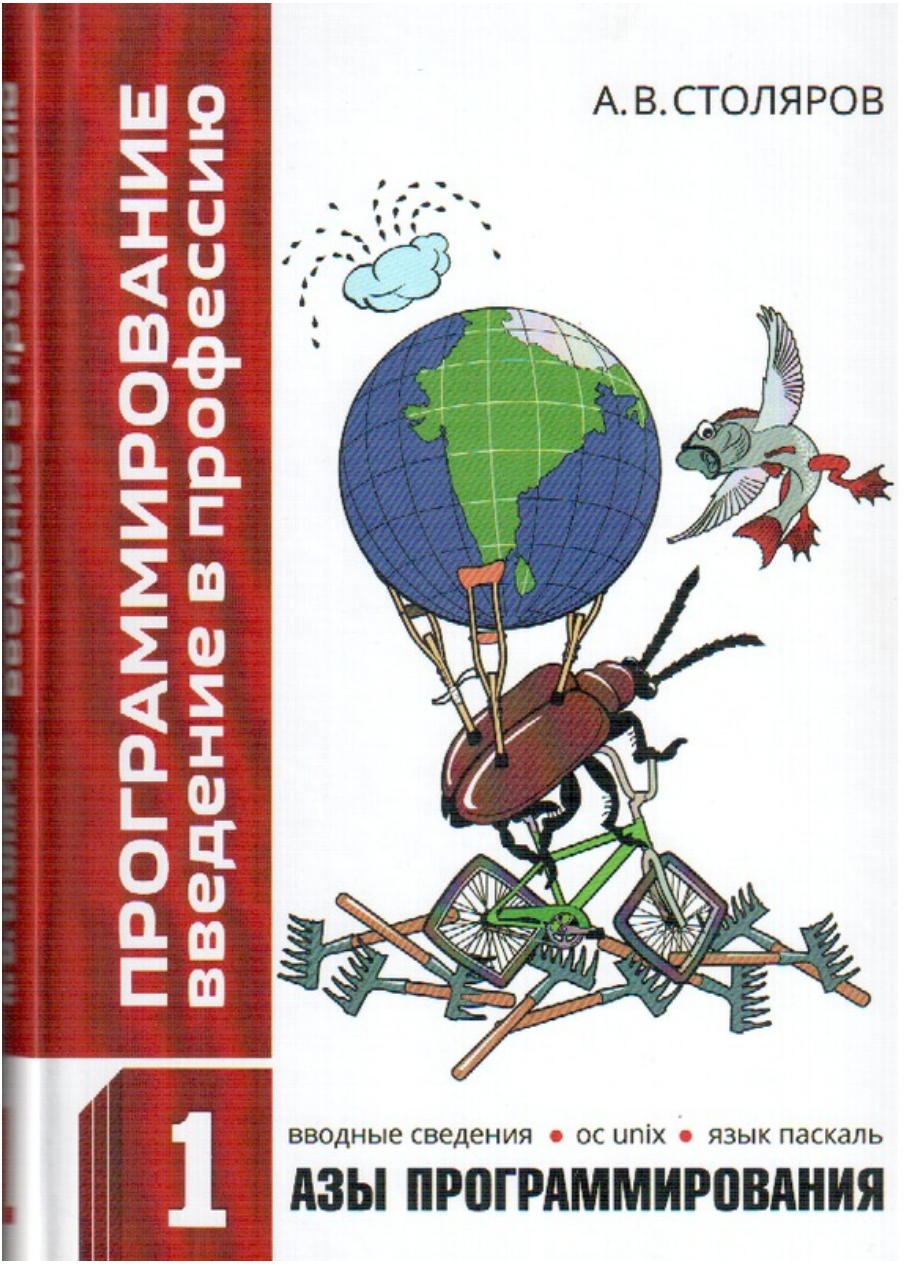

Любое использование данного файла означает ваше согласие с условиями лицензии (см. след. стр.) Текст в данном файле полностью соответствует печатной версии книги. Электронные версии этой и других книг автора вы можете получить на сайте `http://www.stolyarov.info`

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Программирование: введение в профессию. I: Азы программирования», опубликованное в издательстве МАКС Пресс в 2016 году, 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 искаженных 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, 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 файлообменные 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

I: АЗЫ ПРОГРАММИРОВАНИЯ

МАКС ПРЕСС

Москва — 2016

УДК 519.683+004.4
ББК 32.973.26-018.1
С81

Столяров А. В.
С81 Программирование: введение в профессию. 1: Азы программирования. – М.: МАКС Пресс, 2016. – 464 с.
ISBN 978-5-317-05222-5

В первую часть книги вошли избранные сведения из истории вычислительной техники, обсуждение некоторых областей математики, непосредственно используемых программистами (таких как алгебра логики, комбинаторика, позиционные системы счисления), математических основ программирования (теория вычислимости и теория алгоритмов), принципы построения и функционирования вычислительных систем, начальные сведения о работе с командной строкой ОС Unix.

Вторая часть посвящена начальным навыкам составления компьютерных программ на примере Free Pascal под ОС Unix. Материал ориентирован на изучение в будущем языка Си, так что, в частности, много внимания уделено работе с адресами и указателями, построению списков и других динамических структур данных; в то же время многие возможности Паскаля из рассмотрения исключены. Даются сведения о правилах оформления текстов программ, о тестировании и отладке.

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

УДК 519.683+004.4
ББК 32.973.26-018.1

ISBN 978-5-317-05222-5

© А. В. Столяров, 2016

Оглавление

Предисловие первое, философское 8
Предисловие второе, методическое 12
Предисловие третье, напутственное 28
Предисловие к первому тому 32

1. Предварительные сведения 34
1.1. Что такое компьютер: немного истории 34
1.2. История ОС Unix 48
1.3. Как компьютер работает 52
1.4. Как правильно использовать компьютер 60
1.5. Теперь немного математики 106
1.6. Программы и данные 162

2. Язык Паскаль и начала программирования 194
2.1. Первые программы 195
2.2. Выражения и переменные 206
2.3. Управление последовательностью выполнения 219
2.4. Подпрограммы 240
2.5. Ещё о выражениях 254
2.6. Конструирование программ 261
2.7. Символы и их коды; текстовые данные 271
2.8. Система типов Паскаля 287
2.9. Строки 308
2.10. Оператор выбора 317
2.11. Полноэкранные программы 319
2.12. Файлы 332
2.13. Адреса, указатели и динамическая память 348
2.14. Ещё о рекурсии 389
2.15. Ещё об оформлении программ 407
2.16. Тестирование и отладка 429
2.17. Модули и раздельная компиляция 446

Список литературы 458
Предметный указатель 459

Содержание

Предисловие первое, философское 8
Предисловие второе, методическое 12
Можно ли выучить программиста 13
Самообучение — это тоже не так просто 14
Выход есть, или «Почему Unix» 15
Причина первая — математическая 16
Причина вторая — психологическая 17
Причина третья — эргономическая 19
Причина четвёртая — педагогическая 19
Язык определяет мышление 21
Как испортить хорошую идею и как её спасти 26
Предисловие третье, напутственное 28
Предисловие к первому тому 32

1. Предварительные сведения **34**
1.1. Что такое компьютер: немного истории 34
1.2. История ОС Unix 48
1.3. Как компьютер работает 52
1.3.1. Процессор, память, шина 52
1.3.2. Принципы работы центрального процессора 54
1.3.3. Внешние устройства 56
1.3.4. Иерархия запоминающих устройств 57
1.3.5. Резюме 59
1.4. Как правильно использовать компьютер 60
1.4.1. Операционные системы и виды пользовательского интерфейса 60
1.4.2. Unix на домашней машине 70
1.4.3. Первый сеанс в компьютерном классе 73
1.4.4. Дерево каталогов. Работа с файлами 75
1.4.5. Шаблоны имён файлов 77
1.4.6. История команд и автодописывание имён файлов 78
1.4.7. Управление выполнением задач 79
1.4.8. Выполнение в фоновом режиме 84

1.4.9. Перенаправление потоков ввода-вывода 85
1.4.10. Редакторы текстов 87
1.4.11. Права доступа к файлам 92
1.4.12. Электронная документация 94
1.4.13. Командные файлы в Bourne Shell 96
1.4.14. Переменные окружения 99
1.4.15. Протоколирование сеанса работы 101
1.4.16. Графическая подсистема в ОС Unix 101
1.5. Теперь немного математики 106
1.5.1. Элементы комбинаторики 106
1.5.2. Позиционные системы счисления 122
1.5.3. Двоичная логика 131
1.5.4. Виды бесконечности 136
1.5.5. Алгоритмы и вычислимость 141
1.5.6. Алгоритм и его свойства 151
1.5.7. Последовательность действий тут ни при чём . . . 158
1.6. Программы и данные 162
1.6.1. Об измерении количества информации 162
1.6.2. Машинное представление целых чисел 167
1.6.3. Числа с плавающей точкой 172
1.6.4. Тексты и языки 175
1.6.5. Текст как формат данных. Кодировки 178
1.6.6. Бинарные и текстовые данные 184
1.6.7. Машинный код, компиляторы и интерпретаторы . 187

2. Язык Паскаль и начала программирования 194
2.1. Первые программы 195
2.2. Выражения и переменные 206
2.2.1. Арифметические операции и понятие типа 206
2.2.2. Переменные, инициализация и присваивание . . . 210
2.2.3. Идентификаторы и зарезервированные слова . . . 213
2.2.4. Ввод информации для её последующей обработки 214
2.2.5. Берегись нехватки разрядности! 217
2.3. Управление последовательностью выполнения 219
2.3.1. Простая последовательность операторов 219
2.3.2. Конструкция ветвления 221
2.3.3. Составной оператор 224
2.3.4. Логические выражения и логический тип 226
2.3.5. Понятие цикла; оператор `while` 227
2.3.6. Цикл с постусловием; оператор `repeat` 232
2.3.7. Арифметические циклы и оператор `for` 233
2.3.8. Вложенные циклы 236
2.4. Подпрограммы 240
2.4.1. Процедуры 241

2.4.2. Функции 244
2.4.3. Параметры-переменные 247
2.4.4. Локальные и глобальные переменные 250
2.4.5. Рекурсия 252
2.5. Ещё о выражениях 254
2.5.1. Побитовые операции 255
2.5.2. Именованные константы 256
2.5.3. Разные способы записи чисел 261
2.6. Конструирование программ 261
2.6.1. Концепция структурного программирования . . . 262
2.6.2. Исключения из правил: операторы выхода 263
2.6.3. Безусловные переходы 267
2.6.4. О разбиении программы на подпрограммы 269
2.7. Символы и их коды; текстовые данные 271
2.7.1. Средства работы с символами в Паскале 271
2.7.2. Посимвольный ввод информации 275
2.7.3. Чтение до конца файла и программы-фильтры . . 280
2.7.4. Чтение чисел до конца файла; функция `SeekEof` . 285
2.8. Система типов Паскаля 287
2.8.1. Встроенные типы и пользовательские типы 287
2.8.2. Диапазоны и перечислимые типы 289
2.8.3. Общее понятие порядкового типа 292
2.8.4. Массивы 293
2.8.5. Тип запись 301
2.8.6. Конструирование сложных структур данных . . . 302
2.8.7. Пользовательские типы и параметры подпрограмм 303
2.8.8. Преобразования типов 305
2.9. Строки 308
2.9.1. Строковые литералы и массивы `char`'ов 308
2.9.2. Тип `string` 311
2.9.3. Встроенные функции для работы со строками . . 313
2.9.4. Обработка параметров командной строки 315
2.10. Оператор выбора 317
2.11. Полноэкранные программы 319
2.11.1. Немного теории 320
2.11.2. Вывод в произвольные позиции экрана 322
2.11.3. Динамический ввод 324
2.11.4. Управление цветом 329
2.12. Файлы 332
2.12.1. Общие сведения 332
2.12.2. Текстовые файлы 338
2.12.3. Типизированные файлы 341
2.12.4. Блочный ввод-вывод 344
2.12.5. Операции над файлом как целым 347

2.13. Адреса, указатели и динамическая память 348
2.13.1. Что такое указатель 350
2.13.2. Указатели в Паскале 351
2.13.3. Динамические переменные 353
2.13.4. Односвязные списки 356
2.13.5. Стек и очередь 367
2.13.6. Проход по списку указателем на указатель 373
2.13.7. Двусвязные списки; деки 379
2.13.8. Обзор других динамических структур данных . . 384
2.14. Ещё о рекурсии 389
2.14.1. Взаимная рекурсия 389
2.14.2. Ханойские башни 390
2.14.3. Сопоставление с образцом 396
2.14.4. Рекурсия при работе со списками 400
2.14.5. Работа с двоичным деревом поиска 402
2.15. Ещё об оформлении программ 407
2.15.1. Оператор `if` с веткой `else` 408
2.15.2. Особенности оформления оператора выбора . . . 410
2.15.3. Метки и оператор `goto` 411
2.15.4. Максимальная ширина текста программы 414
2.15.5. Как разбить длинную строку 417
2.15.6. Пробелы и разделители 424
2.15.7. Выбор имён (идентификаторов) 425
2.15.8. Регистр букв, ключевые слова и идентификаторы 427
2.15.9. Как справиться с секциями описаний 428
2.16. Тестирование и отладка 429
2.16.1. Отладка в жизни программиста 429
2.16.2. Тесты 433
2.16.3. Отладочная печать 439
2.16.4. Отладчик `gdb` 441
2.17. Модули и раздельная компиляция 446
2.17.1. Модули в Паскале 448
2.17.2. Использование модулей друг из друга 453
2.17.3. Модуль как архитектурная единица 454
2.17.4. Ослабление сцепленности модулей 454

Список литературы 458
Предметный указатель 459

Предисловие первое, философское

Книга, которую вы читаете, представляет собой практически уникальное явление — но речь в данном случае идёт не о её содержании, этому аспекту пускай оценку дадут другие. Речь идёт о том, каким образом книга появилась на свет.

Идея написать книгу, подобную этой, появилась у меня достаточно давно, по меньшей мере лет пять назад. Опыт написания книг у меня к тому времени уже был, и достаточно большой, но ни одна из ранее написанных мной книжек не превосходила по объёму двести страниц. До сей поры я всегда обходился своими силами; задумав книгу, я просто садился и писал её. Некоторые из моих учебных пособий были изданы в учебных заведениях, где я работаю или работал, другие я благополучно издавал за свой счёт, окупая такое издание продажей части тиража: при тираже в сотню-другую экземпляров это, в принципе, не так сложно, хотя и долго. Несколько раз я предпринимал попытки взаимодействия с издательствами; если бы я согласился на их условия, то мне не пришлось бы издавать за свой счёт некоторые из моих книжек, но этих книжек не было бы на моём сайте в открытом доступе: издатели всегда и везде требуют полной передачи им имущественных прав на книгу, что полностью исключает легальное бесплатное распространение электронной версии. Спасибо, господа, не надо: я пишу свои книги, чтобы их читали, а не чтобы вы на них зашибали деньги, обдирая моих читателей.

Всё было хорошо, пока мои задумки не отличались масштабностью; мне всегда удавалось выкроить пару более-менее свободных недель, чтобы написать текст, и десять-пятнадцать тысяч рублей (эквивалент моей зарплаты примерно за полмесяца), чтобы издать написанную книжку. Но в этот раз реальность несколько отличалась. Во-первых, объём задуманной книги изначально предполагался достаточно большим; издание такого увесистого тома даже самым небольшим тиражом, увы, превосходит мои личные финансовые возможности. Во-вторых, с самого начала было ясно, что для написания книги мне предстоит серьёзный трудовой марафон. Части этой книги, посвящённые программированию на языке ассемблера, теории операционных систем и языку Си++, уже существовали в виде отдельных книг; написать вводную часть, а также части, посвящённые языкам Паскаль и Си, предстояло с нуля. Используя имеющийся опыт, предстоящие трудозатраты я оценил в 500 часов, и, как выяснилось, почти не ошибся.

Пятьсот часов рабочего времени — это никак не пара недель и даже не пара месяцев; с учётом наличия основной работы на написание книги должно было уйти по меньшей мере полгода, причём при условии, что на это время я откажусь от фрилансерских подработок и частных уроков. В дополнение к этому сумма, которую предстояло отдать за издание книги, по самым оптимистичным подсчётам соответствовала моей зарплате за полгода. Всё это вместе превращало проект в прожект.

Можно было бы обратиться к издателям; скорее всего, мне удалось бы найти кого-нибудь, кто согласился бы заключить со мной так называемый договор авторского заказа и даже заплатил бы мне какой-нибудь символический гонорар. Но дело тут не в гонораре. Распространение книги в электронном виде, как я делаю это со всеми своими книгами, оказалось бы тогда полностью исключено; прочитать книгу можно было бы либо только на бумаге, либо, что ещё хуже, **купив** электронную версию. Такой вариант противоречит моим убеждениям; в частности, я глубоко убеждён, что платить за электронные книги можно разве что электронными же сканами денег.

Прежде чем окончательно отказаться от своей идеи, я решил, притом без особой надежды, попробовать последнюю возможность, которую видел: модный в последнее время *краудфандинг*; попросту говоря, я решил попросить денег на проект у публики.

Посмотрев на сайты, ориентированные на краудфандинг, я вынужден был отказаться от их использования. Такими сайтами вообще нельзя пользоваться: работа с ними требует регистрации, при которой необходимо принять «условия использования», а эти условия, в частности, разрешают тамошним хозяевам рассылать зарегистрированным пользователям рекламу и вообще всё, что им там вздумается — и таковы условия использования на **всех** краудфандинговых сайтах, которые я видел. К тому же они дружно отказываются работать при отключённом в браузере JavaScript. Я попросту не могу никого просить посещать подобные сайты, это исключено.

Кроме того, я не приемлю модные в последние годы так называемые «социальные сети», которые, вообще говоря, совсем и не сети[1]; довершает картину то обстоятельство, что я крайне щепетилен в выборе средств рекламы и пиара и категорически не готов терпеть ничего даже отдалённо напоминающего спам — в конце концов, моя диссертация по философии называлась «Информационная свобода и информационное насилие», а выросла она из исследования частного вопроса о том, по каким конкретно причинам нельзя считать спам проявлением свободы слова.

Между тем, собрать нужно было солидную сумму в 600 000 рублей. Предполагалось, что половина этой суммы уйдёт на частичную компенсацию моего рабочего времени, что позволило бы остаться на плаву, не тратя время на случайные подработки; вторая половина суммы должна была уйти на издание бумажной книги. Выкладывая на свой сайт `stolyarov.info` анонс и материалы проекта, я, если совсем честно, почти ни на что не надеялся, но всё-таки продумал систему поощрений для тех, кто окажет мне материальную поддержку: за по-

[1] Подробности о моём отношении к так называемым «социальным сетям» изложены в моей статье «Театр контентного абсурда. Социальные сети: история одной терминологической деформации», которую легко найти в Интернете с помощью поисковых машин.

жертвование от 300 рублей я обещал упоминание в списке спонсоров, который будет помещён в книге, за пожертвование от 500 рублей — «фирменный» компакт-диск с автографом автора; при пожертвовании от 1500 рублей — экземпляр бумажной книги, опять же, с автографом, а от 3000 рублей — книгу в подарочном исполнении, в котором она будет изготовлена в соответствии с количеством таких пожертвований.

Почти сразу — во всяком случае, существенно раньше, чем поступили первые пожертвования — несколько человек задали мне вопрос, что будет с деньгами, если нужная сумма не соберётся; в ответ я написал отдельную страничку, где пообещал *хотя бы что-то* сделать в любом случае, даже если не получу вообще ни одного пожертвования. Конкретно я пообещал, что если сумма собранных пожертвований окажется меньше 25 000 рублей, я всё равно напишу часть книги, посвящённую языку Си, и издам её отдельной книгой, плюс к тому в очередной раз доработаю текст своей книги по Си++ и переиздам её в четвёртый раз. При сумме пожертвований от 25 до 55 тысяч я пообещал доработать и переиздать также свою старую книжку по NASM, при сумме от 55 до 100 тысяч — переработать и переиздать «Введение в операционные системы», при сумме от 100 до 120 тысяч — написать часть, посвящённую Паскалю, и издать её отдельной книгой. Наконец, при достижении порога в 120 тысяч я пообещал, что задуманную книгу напишу полностью и продолжу сбор денег, чтобы сделать возможным её издание. Датой принятия решения я назначил 1 сентября 2015 года, тогда как описываемые события происходили в начале января — анонс проекта был датирован 7.01.2015.

После анонса два дня было тихо; первое пожертвование поступило на следующий день, но это можно было не считать: проект решил поддержать один из моих старых знакомых, известный под ником Gremlin. Карусель завертелась лишь 10 января, когда за день мне пришло семь пожертвований на общую сумму свыше 14 тысяч. Пользуясь случаем, я хотел бы самым искренним образом поблагодарить Григория Крайнова, который, прислав второе по счёту пожертвование, не поленился и донёс информацию о проекте до широкой публики через пресловутые «социальные сети».

Первый из намеченных рубежей в 25 000 проект преодолел уже 12 января, второй рубеж (55 000) — 16 января; 4 февраля сумма перевалила за 100 000, а 10 февраля — за магические 120 000, так что все мои «запасные варианты» разом утратили актуальность; книгу теперь нужно было заканчивать хоть тушкой, хоть чучелом.

Конечно, не всё было столь радужно; ближе к весне первая волна пожертвований окончательно иссякла, так что мне даже пришлось прервать работу над рукописью, чтобы заработать денег на стороне. Летом я догадался сообщить о проекте на сайте Linux.Org.Ru, отдельное спасибо его владельцу Максиму Валянскому за разрешение на это; анонс породил вторую волну пожертвований. Проект много раз «уходил в

минус» и снова выныривал, до последнего было непонятно, хватит ли денег на издание и в каком виде. Сейчас, когда готовится к печати первый том, денег на это достаточно, но при этом, видимо, резервы будут полностью исчерпаны, так что со вторым томом придётся либо подождать, пока не соберётся нужная сумма, либо изыскивать какие-то ещё ресурсы. Но все эти трудности с лихвой перекрывает одно важное обстоятельство: никто, включая меня самого, не ожидал, что удастся собрать в виде пожертвований сумму, превышающую 400 000, то есть больше двух третей от заявленной цели.

Здесь я привожу полный (на 19 февраля 2016 г.) список донэйторов, кроме тех, кто пожелал сохранить инкогнито; каждый участник включён в список под таким именем, которое указал сам:

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 Вихрева, 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.

Я хотел бы выразить свою глубочайшую признательность всем, кто принял участие в финансировании проекта. Кроме того, отдельное огромное спасибо: Глебу Семёнову и Алексею Виссарионову, которые не поленились прочитать рукопись и буквально завалить меня замечаниями, позволившими сделать текст гораздо лучше, чем он был; Андрею

Вороненко за ценные замечания относительно «математической» части книги; Екатерине Ясиницкой за героический корректорский труд, граничащий с подвигом; Елене Доменновой за прекрасный рисунок для обложки и дизайн этой самой обложки.

Кстати об обложке. Сюжет рисунка, где земной шар стоит на трёх костылях на спине бага, едущего по полю с граблями на велосипеде с квадратными колёсами, а вокруг летает офигевшая рыба с крыльями и перепончатыми лапами, я позаимствовал из произведения неизвестного автора, которое легко отыскивается в Интернете по названию «Удивительный мир программирования». Исходное произведение представляло собой рисунок фломастерами по whiteboard'у, который кто-то не поленился сфотографировать. Большое спасибо автору оригинального рисунка за идею, обеспечившую меня изрядной дозой позитивного настроения; надеюсь, что наш римейк ему понравится `:-)`

Опыт этого проекта заставил меня во многом переосмыслить своё отношение к окружающей действительности и в каком-то плане даже поверить в человечество. Вряд ли можно придумать другое столь же убедительное доказательство тому, что моя деятельность востребована и я не напрасно трачу время на свои книги. Но главный вывод из успеха нашего с вами, дорогие донейторы, проекта состоит в том, что **мы можем, действительно можем обойтись без копирайтных паразитов и вообще института так называемого «авторского» (а на деле сугубо издательского) права**. Создатели свободного программного обеспечения в своей области показали это уже давно; в области художественной литературы это обстоятельство уже тоже практически очевидно, порукой тому множество «самиздатовских» сайтов в Интернете и обилие любительских переводов зарубежной «художки»; книга, которую вы держите в руках — это ещё один весьма наглядный гвоздь в крышку гроба традиционного (т. е. копирайтного) издательско-медийного бизнеса, построенного на информационном насилии, и очень серьёзный шаг к построению свободного информационного общества, в котором люди вольны вступать в коммуникацию по обоюдному согласию передающего и принимающего, не оглядываясь при этом на всевозможных цензоров и «правообладателей». С этой весьма убедительной, хотя и небольшой победой я хотел бы сегодня поздравить и вас, и себя.

Предисловие второе, методическое

Это предисловие я адресую, как ни странно, не тем, кого считаю основной аудиторией моей книги, то есть не тем, кто решил изучать программирование; для них у меня припасено ещё одно предисловие, которое я назвал «напутственным». Что до методического предисловия, то оно предназначено скорее для тех, кто программировать уже умеет, а

также и для моих коллег-преподавателей; здесь я попытаюсь объяснить мой подход, а заодно и причины появления этой книги.

Можно ли выучить программиста

Ситуация, сложившаяся сегодня с подготовкой новых программистов, при первом взгляде вызывает ощущение изрядного абсурда. С одной стороны, программист — одна из самых востребованных, высокооплачиваемых и при этом дефицитных специальностей: кадровый голод в этой сфере не исчезает во время самых суровых кризисов. Зарплаты квалифицированных программистов сравнимы с зарплатами топ-менеджмента средних, а иногда и крупных компаний, и даже на такую зарплату кандидата приходится подолгу искать. С другой стороны, *фактически программированию нигде не учат.* Большинство преподавателей высших учебных заведений, ведущих «программистские» дисциплины, сами никогда не были программистами и имеют об этом виде деятельности весьма приблизительное представление; оно и понятно, большинство тех, кто может программировать за деньги, в современных условиях именно программированием и зарабатывают. В единичных «топовых» ВУЗах среди преподавателей всё же встречаются бывшие, а иногда и действующие программисты, но ситуацию в целом это не спасает. Люди, одновременно умеющие и программировать, и учить, встречаются довольно редко, но даже среди них немногие способны адекватно представить себе общую методическую картину становления нового программиста; судя по результатам, наблюдаемым на выходе, если такие люди и есть, воплотить своё видение программистского образования в конкретный набор рассматриваемых в ВУЗе дисциплин им не удаётся, слишком велико сопротивление среды.

С обучением в ВУЗах есть и другая проблема. Абитуриенты поступают на «программистские» специальности, имея в большинстве случаев весьма приблизительное представление о том, чем им предстоит заниматься. Программирование — это отнюдь не такой вид деятельности, которому можно научить кого угодно; здесь необходимы весьма специфические способности и склонности. По большому счёту, все программисты — изрядные извращенцы, поскольку ухитряются получать удовольствие от работы, от которой любой нормальный человек бежал бы без оглядки. Но распознать будущего программиста на вступительных экзаменах в ВУЗ или даже на собеседовании (какого нигде никто не проводит) совершенно нереально, особенно если учесть, что в школе программирование либо вообще не изучается, либо изучается так, что лучше бы оно не изучалось. Выйдет из человека толк или нет, становится видно ближе ко второму курсу, но ведь в имеющихся условиях (в отличие, заметим, от большинства западных университетов) сменить выбранную специальность если и возможно в теории, то на практике слишком сложно для массового применения; большинство студентов

предпочитает доучиваться на той специальности, куда изначально поступили, не смотря на очевидность ошибки в её выборе. В итоге даже среди студентов ВМК МГУ, где имеет честь преподавать автор этих строк, будущих программистов наблюдается в лучшем случае треть, а будущих *хороших* программистов — процентов десять.

Больше того, есть основания предполагать, что создание программиста в рамках ВУЗа вообще принципиально невозможно: ремесло не передаётся в стенах учебных заведений, ремесло передаётся только в мастерской — от действующего мастера к ученику, причём непосредственно в процессе работы, и касается это отнюдь не только программирования.

При всём при этом программисты откуда-то появляются, и вывод из этого, возможно, неутешителен, но несомненен: **стать программистом человек может только и исключительно в результате самообучения**. Заметим, это подтверждается и личным опытом автора этих строк, и опытом других программистов: на вопрос, сам человек учился программированию или его научили в ВУЗе, второго ответа пока что не дал ни один из знакомых автору профессионалов.

Самообучение — это тоже не так просто

Автор этих строк начал всерьёз программировать примерно четверть века назад — именно тогда, когда эта профессия вдруг перестала быть уделом узкого круга никому не известных людей и превратилась в массовое явление. Но в те времена мир был устроен несколько иначе. Господствующей платформой (если вообще можно так выразиться в применении к реалиям того времени) была система MS-DOS и её многочисленные клоны, а типичный внешний вид экрана компьютера образовывали синие панельки Norton Commander. Написать программу для MS-DOS было несложно, средств для этого было — хоть отбавляй, так что на первую половину 1990-х пришёлся уникальный расцвет любительского программирования. Многие из тех любителей стали потом профессионалами.

Современные условия качественно отличаются от эпохи начала девяностых. Все господствующие ныне платформы делают акцент на графический интерфейс пользователя; создание программы с GUI требует понимания принципов событийно-ориентированного построения приложений, умения мыслить в терминах объектов и сообщений, то есть, попросту говоря, чтобы сделать программу, снабжённую графическим интерфейсом пользователя, *необходимо* ***уже*** *быть программистом*, так что варианты «попробовал — понравилось» или «попробовал — получилось» отсекаются сугубо технически. Более того, начать освоение программирования с рисования окошек в большинстве случаев означает необратимо травмировать собственное мышление; такая травма полностью исключает достижение высокой квалификации в будущем. Даже безобидный, казалось бы, факт построения каждой программы по

событийно-ориентированному шаблону искажает мышление; некоторые начинающие ухитряются за написанием обработчиков событий начисто упустить факт существования главного цикла, восприятие программы перестаёт быть цельным.

Единственным прибежищем программистов-любителей внезапно оказалась веб-разработка. К сожалению, начав в этой области, люди обычно ею же и заканчивают. Разницу между скриптами, составляющими веб-сайты, и серьёзными программами можно сравнить, пожалуй, с различием между мопедом и карьерным самосвалом; кроме того, привыкнув к «всепрощающему» стилю скриптовых языков типа того же PHP, большинство неофитов оказывается принципиально неспособно перейти к программированию на строгих языках типа Джавы или тем более Си, а хитросплетения Си++ для таких людей оказываются за горизонтом понимания. Веб-кодеры, как правило, называют себя программистами и часто даже получают неплохие деньги, не подозревая при этом, что такое настоящее программирование и что они для себя потеряли.

Выход есть, или «Почему Unix»

Всякий раз, когда положение начинает казаться безвыходным, есть смысл поискать выход там, где его ещё не искали. В данном конкретном случае выход из положения немедленно обнаруживается, стоит только сделать шаг в сторону от офисно-домашнего компьютерного мейнстрима наших дней. Операционные системы семейства Unix на протяжении всей истории сети Интернет прочно и незыблемо удерживали за собой сектор серверных систем; начиная с середины 1990-х Unix-системы проникли на компьютеры конечных пользователей, а сегодня их доля на настольных компьютерах и ноутбуках такова, что игнорировать её больше не получается. Особенно интересной становится эта ситуация, если учесть, что MacOS X, используемая на «роскошных» макбуках, представляет собой не что иное как Unix: в основе MacOS X лежит система Darwin, которая относится к семейству BSD.

Несмотря на наличие в юниксоподобных системах графических интерфейсов, по своей развесистости часто превосходящих их аналоги в системах мейнстрима, основным инструментом профессионального пользователя этих систем всегда была и остаётся командная строка — просто потому, что для человека, умеющего с ней обращаться, правильно организованная командная строка оказывается существенно удобнее «менюшечно-иконочных» интерфейсов. Возможности графического интерфейса ограничены фантазией его разработчика, тогда как возможности командной строки (разумеется, при грамотной её организации) ограничены только характеристиками компьютера; работа в командной строке происходит быстрее, иногда в десятки раз; наконец, руки, освобождённые от необходимости постоянно хвататься за мышку,

устают существенно меньше, перестаёт болеть правый плечевой сустав и запястье. Между прочим, для человека вообще более естественно выражать свои мысли (в данном случае — пожелания) словами, а не жестами. В системах семейства Unix командная строка организована столь грамотно, что её господствующему положению в качестве основного интерфейса ничто не угрожает. В контексте нашей проблемы здесь важен тот факт, что **написать программу, предназначенную для работы в командной строке, много проще, чем программу с GUI**; наличие в Unix-системах командной строки в качестве основного инструмента работы делает возможным то самое любительское программирование, которое казалось безвозвратно утраченным, когда в мейнстриме «старый добрый» MS-DOS сменился системами линейки Windows.

При внимательном рассмотрении именно системы семейства Unix оказываются не только подходящим, но и (в современных условиях) единственным возможным вариантом, коль скоро речь идёт об обучении программированию. Я позволю себе выделить здесь четыре причины.

Причина первая — математическая

Любая компьютерная программа есть, как известно, запись некоторого *алгоритма* на избранном языке программирования. Что такое алгоритм — никто в действительности не знает и знать, что интересно, не может, иначе пришлось бы выбросить на свалку всю теорию вычислимости заодно с теорией алгоритмов, забыть тезис Чёрча-Тьюринга и вообще отказаться от теоретической составляющей computer science. Тем не менее принято считать, что всякий алгоритм выполняет *преобразование из множества слов (цепочек символов) над некоторым алфавитом в само это же множество.* Конечно, не всякое такое преобразование может быть выполнено алгоритмом, ведь преобразований таких, как несложно показать, континуум, тогда как алгоритмов — множество не более чем счётное; более того, алгоритм *сам по себе* не есть такое преобразование, ведь речь иногда может идти об *эквивалентных* алгоритмах, то есть таких, которые из одного и того же входного слова всегда «делают» одно и то же выходное; иначе говоря, для одного и того же преобразования может существовать больше одного алгоритма (если быть точным, для каждого преобразования алгоритм либо не существует, либо их существует бесконечное количество). Тем не менее, всякий алгоритм выполняет именно такое преобразование, и только этим он, вообще говоря, интересен. Если угодно, алгоритм — это такая штука, которая берёт некое входное слово («прочитывает» его), что-то там такое *конструктивное* делает и выдаёт другое слово; неопределённость понятия здесь заключена в слове «конструктивное», которое тоже, разумеется, невозможно определить.

Многие программы в системах семейства Unix работают именно так: читают данные из стандартного потока ввода и записывают их в стандартный поток вывода. Такие программы имеют даже своё название — *фильтры*. Благодаря развитым средствам командной строки такие программы-фильтры можно комбинировать «на лету», решая самые разнообразные задачи, сводящиеся к преобразованию текста. Поскольку текстовое представление практически универсально, алгебра консольных программ оказывается средством неожиданно мощным. Каждая новая консольная программа, сколь бы простой она ни была, становится частью этой системы, делая саму систему ещё немного мощнее, а диапазон решаемых задач — ещё немного шире. При этом программы-фильтры полностью соответствуют пониманию алгоритма как преобразования из входного слова в выходное.

Но при обучении главное даже не это. Развитая культура консольных приложений даёт возможность начинающему программисту *написать настоящую программу*, а не игрушечный этюд. Почему это столь важно, нам поможет понять

Причина вторая — психологическая

Программирование — это, в конечном счёте, не более чем ремесло, а ремесло *выучить* невозможно, ему можно только *научиться*. Прежде чем новичок превратится в программиста, ему необходимо сделать ряд очень важных шагов. **Первый шаг** — это переход от задач из учебника к задачам, поставленным самостоятельно, при этом не вымученным, не делаемым «потому что надо», а таким, которые делаются, потому что данному конкретному субъекту показалось *интересно* подчинить себе компьютер и заставить его решить именно такую задачу.

Второй шаг — переход от этюдов к реальному решению реально вставшей перед учеником проблемы, пусть сколь угодно простой, но настоящей. Это может быть календарь или записная книжка, напоминалка о днях рождения друзей, какой-нибудь простой преобразователь текстов (а хоть бы и для удаления лишних пробелов), всё что угодно. Для автора этих строк в своё время такой программой стала «ломалка» для игры «F-19», которая подправляла байтик в файле списка пилотов, «оживляя» тех, кто помечен как погибший. Найти нужный байтик оказалось существенно сложнее, нежели потом написать программу на Паскале, которая в нужные позиции нужного файла записывает нули, и тем не менее именно эта примитивная, на один экран программка позволила прыгнуть, как говорят, на следующий уровень, о чём сам ваш покорный слуга догадался лишь лет через десять.

Третий шаг будущий программист замечает, когда этот шаг давно уже сделан. Новое качество в этот раз состоит в том, что у какой-то пусть даже очень и очень примитивной программы, написанной вами, появляется сторонний пользователь. Конечно, ни о каких деньгах тут

речи не идёт: речь идёт о том, что вам удалось написать не просто полезную программу, а такую, полезность которой оценил (реально, а не на словах) кто-то кроме вас самих. Иначе говоря, нашелся кто-то, кто согласен тратить время, пользуясь вашей программой, потому что получаемые результаты оказываются для него ценнее потраченного времени. Во многих случаях такой программой оказывается какая-нибудь простенькая игрушка, реже — что-то более серьёзное, какой-нибудь несложный каталогизатор или что-нибудь ещё. При этом интересно, что программу-то вы, быть может, и напишете, но вы никак не можете заранее знать, что кто-то станет её использовать, так что собственный переход на следующий уровень здесь обычно не осознаётся. И лишь когда вы вдруг обнаруживаете, что кто-то действительно начал пользоваться вашей поделкой, причём не потому, что вы его слёзно умоляли, а сам, добровольно, — вот в этот момент можете себя поздравить от всей души: **вы стали программистом**.

Конечно, будет ещё и четвёртый рубеж — получение денег за работу по написанию программ. Состоявшимся профессионалом в большинстве случаев себя можно считать лишь после этого. Однако разница между профессионалом и любителем отнюдь не столь значительна, как между программистом и не-программистом. В конце концов, истории известны примеры, когда к моменту первого монетарного эффекта от своей программистской деятельности человек имел уже такую высокую квалификацию, что усомниться в его профессионализме никто бы не рискнул; взять хоть Линуса Торвальдса.

Так вот, **обучая человека программировать с использованием систем семейства Windows, мы тем самым лишаем его возможности сделать все три вышеперечисленных шага**. Настоящую программу под Windows, под каковой можно понимать исключительно оконное приложение, можно написать, только *уже будучи программистом*; текстовые программки, про которые ученикам никто даже не объясняет, как их правильно запускать (ага, привет readln'у в конце каждой программы), настоящими не выглядят ни анфас, ни в профиль, а потому у такой программы никогда не появится стороннего пользователя, да и сам автор пользоваться этим непонятно чем не станет. Больше того, если результат столь убог и нет никаких шансов сделать его сколько-нибудь похожим на нечто настоящее, вряд ли нашего обучаемого заинтересует перспектива потратить несколько часов оставшейся ему жизни, чтобы сделать вот такое вот никуда не годное решение пусть даже очень интересной задачи.

Именно поэтому обучаемому жизненно важно не просто программировать под Unix, ему необходимо *жить* под Unix'ом, то есть именно Unix (будь то Linux или любая другая Unix-система) использовать в повседневной работе, для прогулок по Интернету, общения по электронной почте и через разнообразные мессенджеры, для работы над текстами, для просмотра фильмов и фотографий, вообще для всего, для чего

обычно используют компьютеры. Только в этом случае у нашего обучаемого на более-менее ранних стадиях развития может возникнуть такая потребность *из повседневной жизни*, для которой может понадобиться написать программу.

Причина третья — эргономическая

К каким бы ухищрениям ни прибегали создатели графических пользовательских интерфейсов, по эффективности использования и тому, что называется английским словом *usability*, им никогда не переплюнуть и не обогнать старую добрую командную строку. Не соглашаются с этим утверждением лишь те, кто в командной строке работать не умеет, а не умеют обычно те, кто никогда этого делать всерьёз не пробовал.

Не представляя своей жизни без GUI, вы никогда не поймёте, как на самом деле должна выглядеть работа с компьютером. Большинство современных программ с точки зрения usability представляет собой уродливых монстров, на борьбу с изъянами которых у пользователей уходит девять десятых всех сил, при этом пользователи ухитряются этого положения в упор не замечать, так как просто не знают, что может быть как-то иначе. Именно поэтому необходимо освоить командную строку (пусть даже не строя планов по прекращению использования GUI: это произойдёт само собой), и сделать это нужно как можно раньше, пока мозг не потерял способности к быстрому обучению и мгновенной адаптации к непривычным условиям: после 25 лет изучать нечто принципиально новое становится настолько сложнее, что мотивация для этого требуется качественно более высокая.

Ну а действительно полноценные средства командной строки за пределами семейства Unix, извините, не водятся. Так уж получилось.

Причина четвёртая — педагогическая

Если среди учеников, пытающихся освоить программирование в нечеловеческих виндовых условиях, всё же окажется будущий программист, страшные псевдо- и недопрограммы, которые его заставляют писать (чаще всего в какой-нибудь безнадёжно мёртвой среде вроде Turbo Pascal), очень быстро перестанут такого субъекта удовлетворять, и ему захочется чего-то настоящего. Учитель вряд ли станет объяснять продвинутому ученику, как писать оконные программы под Windows (большинство учителей не умеют этого сами), но будущего программиста такие мелочи не остановят. Взяв в руки первую попавшуюся книжку, он освоит рисование окошек самостоятельно.

В некоторых (увы, довольно редких) случаях даже это не сможет его испортить, и через несколько лет такой ученик, на самом деле прирождённый программист, станет грамотным и матёрым спецом, за которого передерутся работодатели. Люди такого класса, такие, кого

с правильного пути не своротит никакой учитель и никакой министр образования, реально существуют; больше того, на этих уникумах держится вся современная индустрия. Проблема в том, что таких людей очень, очень, **очень** мало.

Гораздо чаще наблюдается совершенно иная картина: начав с рисования окошек, новичок необратимо травмирует собственное мышление, поставив мир с ног на голову: обдумывать программу он начинает не с предметной области, а с элементов графического интерфейса, они же (точнее, обработчики их событий) становятся своеобразным скелетом любой его программы, на которые навешивается «мясо» функциональности. Перспектива такого действия, как, например, смена используемой библиотеки виджетов, приводит такого программиста в ужас, внешне выражающийся фразой «что вы, это совершенно невозможно, программу для этого придётся переписать с нуля»; о том, что вид пользовательского интерфейса вообще-то можно сделать *сменным*, он даже подумать боится. Такие люди часто применяют совершенно феерическую технику, которую более грамотные программисты в шутку называют «рисованием на обратной стороне экрана» — когда не хватает обработчиков событий, в диалоговом окне создаётся невидимый (!) графический объект, через который другие объекты обмениваются информацией.

В нынешних условиях такой программист окажется вполне востребован, больше того, ему даже будут платить неплохую зарплату; однако он никогда не поймёт, **что** он в действительности потерял и насколько более интересной могла бы быть его работа, если бы в своё время он не схватился за пресловутые окошки.

Увы, большинство учителей и преподавателей с упорством, достойным лучшего применения, продолжает использовать в учебном процессе компьютеры под управлением ОС Windows; на самом деле при этом учеников и студентов учат программировать под MS-DOS — тот самый MS-DOS, о котором сейчас, спустя почти двадцать лет после его окончательной смерти, даже вспоминать как-то неловко. В таких условиях любые аргументы, высказываемые в пользу сохранения Windows в качестве системы на учебных компьютерах, заведомо оказываются лишь отговорками, а реальная причина здесь одна: иррациональный страх перед освоением всего нового. Windows не годится на роль учебного пособия; продолжать использовать эту систему при наличии заведомо лучших (причём лучших по **всем** параметрам; Windows не имеет абсолютно никаких достоинств в сравнении с Unix-системами) альтернатив — это, мягко говоря, странно. Для человека, взявшегося учить кого-то программированию, не должно быть проблемой освоение непривычной операционной среды.

Язык определяет мышление

Кроме операционной среды, используемой для обучения, важнейшую роль играет также выбор языков программирования. Времена Бейсика с пронумерованными строками, к счастью, прошли; к сожалению, часто (особенно в спецшколах) встречается противоположная крайность. Несостоявшиеся программисты, решившие попробовать себя в роли школьных учителей, «обучают» ни в чём не повинных школьников «профессиональным» языкам, таким как Джава, C# и даже Си++. Конечно, пятиклассник, которому в мозги запихивают Си++ (реальный случай в реально существующем учебном заведении), в результате не поймёт абсолютно ничего, разве что запомнит «волшебные слова» `cin` и `cout` (отметим, что как раз это к реальному программированию на Си++ отношение имеет весьма сомнительное), но таким «гениальным учителям» возможности аудитории совершенно не указ, тем более что способы контроля, естественно, выбираются такие, при которых ученики без особых проблем «проскакивают» контрольные работы и другие «препятствия», так ничего и не поняв из выданного им материала. Автору этих строк встречались школьники, *не понимающие, что такое цикл*, но при этом получающие у себя в школе пятёрки по информатике, где им «преподают Си++».

Учителям такой категории вообще, судя по всему, всё до лампочки: в Си++ используется библиотека STL, а значит, надо рассказывать ученикам STL; разумеется, дальше `vector`'а и `list`'а обучение никогда не заходит (как раз эти два контейнера, пожалуй, самые бесполезные из всего STL), но самое интересное, что ученики, разумеется, так и не понимают, о чём идёт речь. В самом деле, как можно объяснить разницу между `vector` и `list` человеку, который никогда в жизни не видел ни динамических массивов, ни списков и вообще не понимает, что такое указатель? Для такого ученика `list` отличается от `vector` тем, что в нём нет удобной операции индексирования (почему её там нет? ну, нам что-то объясняли, но я ничего не понял), так что вообще-то всегда надо использовать `vector`, ведь он гораздо удобнее. Что? Добавление в начало и в середину? Так оно и для вектора есть, какие проблемы. Ну да, нам говорили, что это «неэффективно», но ведь работает же! Переучить такого ученика практически нереально: попытки заставить его создать односвязный список вручную обречены на провал, ведь есть же `list`, а тут столько ненужного геморроя! Собственно говоря, всё: если нашему обучаемому дали в руки STL раньше, чем он освоил динамические структуры данных, то знать он их уже не будет *никогда*; путь в серьёзное программирование ему, таким образом, закрыт.

Не менее часто встречается и другой вариант: учить пытаются, судя по всему, *чистому Си* (тому, который без плюсов), то есть не рассказывают ни классы, ни контейнеры, ни STL (что, в общем, правильно), ни ссылки, но при этом невесть откуда выскакивают `cin/cout`, тип `bool`

(которого в чистом Си отродясь не было), строчные комментарии и прочие примочки из Си++. Объяснение тут довольно простое: стараниями Microsoft с их VisualStudio в сознании виндовых программистов, особенно начинающих, разница между чистым Си и Си++ временами совсем теряет очертания. В мире Unix с этим всё гораздо лучше: во всяком случае, эти два *совершенно разных* языка никто не путает; но, как уже говорилось, Unix в нашей школе днём с огнём не найдёшь, учителя предпочитают платить государственные деньги за коммерческий софт, бороться с вирусами путём еженедельной переустановки всех компьютеров (опять же реальная ситуация в реальной школе), уродовать мозги учеников, лишь бы только не изучать ничего за пределами мейнстрима.

Впрочем, даже чистый Си в качестве первого языка программирования — это откровенный нонсенс. Причины этого я несколько лет назад подробно расписывал в эссе «Язык Си и начальное обучение программированию», повторяться здесь не буду, отмечу только главное: **к изучению Си нужно подходить, уже понимая указатели и умея с ними обращаться**; более того, желательно иметь некоторый опыт написания программ, чтобы отличать хорошее от плохого, ведь сам язык Си развивает что угодно, кроме культуры программистского мышления.

Несмотря на всё сказанное, я убеждён, что изучать язык Си и низкоуровневое программирование необходимо; просто не следует с этого начинать. Программиста, не знающего Си, работодатели вряд ли воспримут всерьёз, даже если писать на Си от кандидата не требуется, и этому есть причины. Человек, не чувствующий на уровне подсознания, как конкретно компьютер делает то или это, попросту не может писать качественные программы, сколь бы высокоуровневые языки программирования он ни использовал. Изучать азы взаимодействия с операционной системой тоже лучше всего на Си, всё остальное не даёт полноты ощущений.

Необходимость изучения Си, если таковую постулировать, приводит нас к проблеме освоения указателей. Работа с указателями кажется простой только тем, кто давно и прочно умеет с ними обращаться, а для большинства новичков указатели — это огромный и труднопреодолимый барьер. В Си использовать адресные выражения приходится сходу, это нужно, чтобы просто прочитать с клавиатуры целое число. Попытки обучать языку Си людей, не умеющих обращаться с указателями и адресами, сродни небезызвестному принципу воспитания через отбор: кто выплывет — молодец, кто утонул — тех не жалко.

Итак, если иметь в виду в будущем изучение Си, то предварительно нужно освоить какой-то язык, в котором а) есть указатели, причём в полный рост, без всякой сборки мусора; б) без указателей можно обходиться, пока обучаемый не окажется более-менее готов к их восприятию; и в) начав использовать указатели, обучаемый расширит свои возможности, то есть в указателях должна быть реальная потребность.

Замечу, без пункта в) можно было бы использовать C++/STL, но когда в распоряжение ученика попадают всякие `vector`, `list` и пр., стимул для применения низкоуровневых указателей неизбежно пропадает вместе с возможностью когда-нибудь научиться самостоятельно создавать сложные проблемно-ориентированные структуры данных. А вот всем трём пунктам одновременно удовлетворяет только Паскаль; этот язык позволяет подойти к указателям плавно и издали, не используя и не вводя их до тех пор, пока уровень обучаемого не станет для этого достаточным; в то же время с момента их введения указатели в Паскале проявляют практически все свойства «настоящих» указателей, за исключением разве что адресной арифметики. Поиск другого языка с аналогичными возможностями по изучению указателей оказался безрезультатным; похоже на то, что Паскалю просто нет альтернативы.

С другой стороны, если мы рассматриваем изучение Паскаля как подготовительный этап перед Си, можно для экономии времени оставить за кадром часть его возможностей, таких как тип-множество, оператор `with` и вложенные подпрограммы. Следует помнить, что целью изучения здесь является не «язык Паскаль», а *программирование*. Нет совершенно никакого смысла в настойчивом вдалбливании ученику формального синтаксиса, таблиц приоритетов операций и прочей подобной ерунды: на выходе нам нужно получить не *знание языка Паскаль*, который, возможно, ученику никогда больше не понадобится, а умение писать программы. Наиболее высокими барьерами на пути ученика здесь оказываются, во-первых, всё те же указатели, и, во-вторых, рекурсия, с которой тоже можно научиться работать на примере Паскаля. Отметим, что модуль CRT, нежно любимый нашими педагогами[2], во Free Pascal'е под Linux и FreeBSD замечательно работает, позволяя создавать полноэкранные терминальные программы; на Си это сделать гораздо труднее, автор этих строк в своё время, будучи уже опытным программистом, потратил несколько дней на то, чтобы более-менее разобраться с библиотекой `ncurses`.

Вторая «неизбежность» — это программирование на языке ассемблера. Здесь мы вообще имеем нечто, весьма напоминающее небезызвестные взаимоисключающие параграфы. С одной стороны, на языке ассемблера лучше вообще никогда и ничего не писать, за исключением коротких фрагментов в ядрах операционных систем (например, точки входа в обработчики прерываний и всевозможное управление виртуальной памятью) и в прошивках микроконтроллеров. Всё остальное правильней писать на том же Си, эффективность по времени исполнения от этого совершенно не страдает и даже в некоторых случаях повышается благодаря оптимизации; при этом выигрыш по трудозатратам может достигать десятков раз. Большинство программистов не встречает ни

[2]Настолько, что сакраментальное «`uses crt;`» часто можно увидеть в программах, не использующих никакие возможности из него, причём даже в учебниках.

одной «ассемблерной» задачи за всю свою жизнь. С другой стороны, опыт работы на языке ассемблера квалифицированному программисту абсолютно необходим; в отсутствие такого опыта люди *не понимают, что делают.* Поскольку на практике языки ассемблера почти никогда не применяются, единственным шансом получить хоть какой-то опыт становится период обучения, и потому ясно, что пренебречь ассемблером мы никак не можем.

В ходе изучения языка ассемблера можно заодно продемонстрировать, что такое ядро операционной системы, зачем оно нужно и как с ним взаимодействовать; системный вызов перестаёт быть чем-то магическим, когда его приходится делать вручную на уровне машинных команд. Поскольку цель здесь, опять же, не освоение конкретного ассемблера и даже не программирование на уровне ассемблера как таковое, а исключительно *понимание того, как устроен мир*, не следует, разумеется, снабжать ученика уже готовыми библиотеками, которые сделают за него всю работу, в частности, по переводу числа в текстовое представление; наоборот, написав на языке ассемблера простенькую программу, которая считывает из стандартного потока ввода два числа, перемножает их и выдаёт полученное произведение, ученик поймёт и прочувствует гораздо больше, чем если ему предложить написать на том же ассемблере что-то сложное и развесистое, но при этом перевод из текста в число и обратно выполнить за него в какой-нибудь библиотеке макросов. Здесь же следует посмотреть, как организуются подпрограммы с локальными переменными и рекурсия (нет, не на примере факториала, который высосан из пальца и всем уже надоел, скорее на примере сопоставления строки с образцом или ещё на чём-то подобном), как строится стековый фрейм, какие бывают соглашения о связях.

Коль скоро изучать программирование на ассемблере всё равно придётся, логично это сделать *до* изучения Си, поскольку это во многом помогает понять, почему язык Си именно таков, каков он есть. Концепция «Си как заменитель языка ассемблера» очень сильно помогает в изучении этого достаточно странного языка. Адресная арифметика, присваивание как операция, отдельные операции инкремента и декремента и многое другое — всё это гораздо легче понять и принять, если уже знать к этому времени, как выглядит программирование на уровне команд центрального процессора. С другой стороны, идею *начать* обучение программированию с языка ассемблера не хочется даже обсуждать, это заведомый абсурд.

С учётом сказанного выстраивается довольно однозначная цепочка языков для начального обучения: Паскаль, язык ассемблера, Си. В эту цепочку практически в любом месте можно что-нибудь добавить, но ни убирать из неё элементы, ни переставлять их местами, судя по всему, нельзя.

Зная Си, можно вернуться к изучению явления, именуемого операционной системой, и её возможностей с точки зрения программиста,

создающего пользовательские программы. Наш ученик уже понимает, что такое системный вызов, так что можно рассказать ему, какие они бывают, пользуясь уровнем терминологии, характерным для этой предметной области — именно, уровнем описания системных вызовов в терминах функций Си. Файловый ввод-вывод, управление процессами в ОС Unix (которое, к слову, организовано гораздо проще и понятнее, чем в других системах), способы взаимодействия процессов — всё это не только концепции, демонстрирующие устройство мира, но и новые возможности для возникновения у ученика собственных идей, ведущих к самостоятельным разработкам. Освоение сокетов и неожиданное открытие того, сколь просто писать программы, взаимодействующие между собой через компьютерную сеть, даёт ученикам изрядную дозу энтузиазма.

В какой-то момент в курсе стоит упомянуть разделяемые данные и многопоточное программирование, подчеркнув, что с тредами лучше не работать, даже зная, как это делается; иначе говоря, надо знать, как работать с тредами, хотя бы для того, чтобы осознанно принять решение об отказе от их использования. В то же время любому квалифицированному программисту необходимо понимать, зачем нужны мьютексы и семафоры, откуда возникает потребность во взаимоисключении, что такое критическая секция и т. п., в противном случае, к примеру, читая описание архитектуры ядра Linux или FreeBSD, человек просто не поймёт, о чём идёт речь.

Именно такова традиционная последовательность программистских курсов на факультете ВМК: в первом семестре курс «Алгоритмы и алгоритмические языки» поддерживается практикумом на Паскале, во втором семестре лекционный курс так и называется «Архитектура ЭВМ и язык ассемблера», а лекционный курс третьего семестра — «Операционные системы» — предполагает практикум на Си.

Несколько сложнее с четвёртым семестром; лекционный курс там называется «Системы программирования» и представляет собой довольно странное сочетание введения в теорию формальных грамматик и объектно-ориентированного программирования на примере Си++. Если говорить о моём мнении, то Си++ вряд ли можно назвать удачным языком для первичного освоения ООП, да и вообще этому языку не место в программах основных курсов: те из студентов, которые станут профессиональными программистами, этот язык могут освоить (и осваивают) сами, тогда как тем, кто будет работать по родственным или вообще другим специальностям, поверхностное знакомство с узкоспециальным профессиональным инструментом, каким, несомненно, является язык Си++, не прибавляет ни общего кругозора, ни понимания устройства мира.

С читателями, на которых ориентирована эта книга, ситуация выглядит несколько иначе: те, кому программирование как вид деятельности не интересно, просто не станут её читать, а те, кто изначально хотел

стать программистом, но при более близком знакомстве с этим видом деятельности поменял свои планы, скорее всего, бросят чтение где-нибудь на второй-третьей части и до конца в любом случае не доберутся. В то же время тем, кто одолеет эту книгу целиком — то есть будущим профессионалам — может оказаться полезен не то чтобы язык Си++ как таковой, ведь его можно изучить по любой из сотен существующих книг, а скорее тот особый взгляд на этот язык, который я всегда стараюсь донести до студентов на семинарах четвёртого семестра: взгляд на Си++ не как на профессиональный инструмент, а как на уникальное явление среди существующих языков программирования, каким Си++ был до того, как его безнадёжно испортили авторы стандартов. Поэтому я счёл нужным включить Си++ в число языков, описанных в этой книге; об особенностях моего подхода к этому языку говорится в предисловии к седьмой части, которая как раз и посвящена Си++.

Как испортить хорошую идею и как её спасти

К сожалению, мелочи и частности делают серию программистских курсов, принятую на ВМК, безнадёжно далёкой от совершенства. Так, на лекциях первого семестра студентам зачем-то вдалбливают так называемый «стандартный Паскаль», который представляет собой монстра, годного разве что для устрашения и в природе не встречающегося; при этом на семинарах тех же студентов заставляют программировать на Turbo Pascal под всё тот же MS-DOS — на мёртвой системе в мёртвой среде, вдобавок не имеющей никакого отношения к «стандартному» Паскалю, о котором рассказывают на лекциях. Больше того, лекции построены так, будто целью является не научиться программировать, а изучить в подробностях именно язык Паскаль как таковой, причём в его заведомо мёртвой версии: тратится море времени на формальное описание его синтаксиса, многократно подчёркивается, в какой конкретно последовательности должны идти секции описаний (любая реально существующая версия Паскаля позволяет расставить описания в произвольной последовательности, и уж точно нет ничего хорошего в том, чтобы, следуя соглашениям стандартного Паскаля, описывать в самом начале программы *метки*, которые будут использоваться только в головной программе, то есть в самом конце текста программы, но видны при этом будут зачем-то во всех процедурах и функциях).

Во втором семестре язык ассемблера, что уже, увы, ожидаемо, демонстрируется тоже под MS-DOS, то есть изучается система команд 16-битных процессоров Intel (8086 и 20286). Программирование под заведомо мёртвую систему расхолаживает студентов, культивирует презрительное отношение к предмету (и это отношение часто переносится на преподавателей). Впрочем, в действительности нет особой разницы, какой конкретно ассемблер изучать, главное — поймать логику работы с регистрами и областями памяти; но как в этом курсе расставляют-

ся акценты — со стороны понять очень трудно, во всяком случае, на выходе большинство студентов не понимает, что такое прерывание, и практически никто не знает, как выглядит стековый фрейм.

Всё более-менее приходит в норму лишь на втором курсе, где используется Unix (FreeBSD) и чистый Си изучается именно в этой, идеально подходящей для Си среде. Однако перед этим целых два семестра тратятся, мягко говоря, с сомнительной эффективностью.

Несколько лет назад на ВМК на одном из трёх потоков начался эксперимент по внедрению новой программы. Надо отдать должное авторам эксперимента, гнилой труп MS-DOS они из учебного процесса устранили, но, к сожалению, вместе с откровенной мертвечиной экспериментаторы выкинули и язык Паскаль, начиная обучение прямо с Си. Это было бы вполне нормально, если бы все поступающие на первый курс абитуриенты имели хотя бы зачаточный опыт программирования: для человека, уже видевшего указатели, Си не представляет особых сложностей. Увы, даже ЕГЭ по информатике не может обеспечить наличие хотя бы самых примитивных навыков программирования у поступающих: сдать его на положительный балл вполне можно, совершенно не умея программировать, не говоря уже о том, что указатели в школьную программу информатики (и, соответственно, программу ЕГЭ) не входят. Большинство первокурсников приходят на факультет с абсолютно нулевым уровнем понимания, что такое программирование и как это всё выглядит; Си становится для них первым в жизни языком программирования, а на выходе мы получаем откровенную катастрофу.

Принятый на ВМК порядок подачи программистских дисциплин на младших курсах представляется мне потенциально самым удачным из всего, что я когда-либо видел — если бы не упоминавшиеся выше «мелочи» вроде использования заведомой мертвечины, убивающие всю идею. Упорное нежелание преподавателей отказаться от Windows (точнее, от MS-DOS) привело к тому, что будущее всей концепции оказалось под сомнением; модернизация, которая рано или поздно попросту неизбежна, может теперь носить характер тотальной революции, способной окончательно разрушить фундаментальный характер программистской подготовки на ВМК, превратив её в малоосмысленную техническую дрессировку. Книга, которую вы держите в руках, представляет собой попытку её автора сохранить хотя бы в каком-то виде уникальный методический опыт, которому грозит полное забвение. Я убеждён, что для модернизации серии программистских курсов, имеющихся на ВМК, совершенно не обязательно эту серию полностью разрушать, достаточно отказаться от использования мёртвых инструментов в пользу инструментов живых — но это само по себе возможно только с переходом на системы семейства Unix.

И последнее. Если вы, кто читает сейчас этот текст, входите в число моих коллег-преподавателей и хотите задействовать эту книгу в учебном процессе, я должен вас честно предупредить об одной очень важной

вещи. Вашим собственным **основным** способом общения с компьютером в повседневной жизни должна быть (стать?) командная строка. Книга расчитана на то, что ученик использует командную строку, отдавая ей предпочтение перед графическими интерфейсами — только так у него есть шанс сделать перечисленные выше шаги к профессии. Если вы сами копируете файлы, перетаскивая иконки мышкой, вы вряд ли сможете убедить ваших учеников, что командная строка эффективнее и удобнее, ведь вы сами в это не верите. Насколько я могу судить, в таком случае моя книга для вас бесполезна.

Предисловие третье, напутственное

Это предисловие, последнее из трёх, адресовано тем, кто решил воспользоваться моей книгой, чтобы изучать программирование — один из самых увлекательных видов человеческой деятельности для тех, кто не обделён интеллектом.

Издавна самым умным и умелым людям хотелось создавать что-то такое, что *действует само по себе*; до появления электричества такое было доступно только механикам-часовщикам. В XVIII веке Пьер Жаке-Дро[3] создал несколько уникальных механических кукол, которые назвал «автоматонами»: одна из этих кукол играет на органе пять разных мелодий, при этом нажимает пальцами нужные клавиши органа, пусть и сделанного специально для неё, но при этом реально управляющегося клавишами; другая рисует на бумаге довольно сложные картинки — любую из заданных трёх. Наконец, последняя, самая сложная кукла, «Пишущий мальчик» или «Каллиграф», пишет на бумаге фразу, обмакивая гусиное перо в чернильницу; фраза состоит из сорока букв и «программируется» поворотами специального колеса. Этот механизм, завершённый в 1772 году, состоит из более чем шести тысяч деталей.

Конечно, самое сложное при создании такого автомата — это *придумать* всю его механику, найти такое сочетание деталей, которое заставит механические руки совершать столь сложные и точные движения; несомненно, создатель «Пишущего мальчика» был уникальным гением в области механики. Но коль скоро вы имеете дело с механикой, одной гениальности вам не хватит. Пьеру Жаке-Дро пришлось изготовить каждую из шести тысяч деталей, выточить их из металла с фантастической точностью; конечно, часть работы он поручил наёмным работникам принадлежащей ему мастерской, но факт остаётся фактом: кроме гениальности конструктора таких механических изделий, для их появления необходимо ещё огромное количество человеческого труда, притом такого, который невозможно назвать творческим.

[3] В ряде источников — «Дроз», но это неправильно; последняя буква во французской фамилии *Droz* не произносится.

Автоматоны Жаке-Дро представляют собой своего рода экстремальную иллюстрацию возможностей творящего человеческого разума в сочетании с вложением большого количества рутинного труда по изготовлению материальных деталей; но тот же принцип мы можем наблюдать едва ли не в любом виде конструкторской деятельности. Гениальный архитектор может нарисовать эскиз прекрасного дворца и создать его подробный проект, но дворец никогда не появится, если не найдётся желающих оплатить труд тысяч людей, задействованных во всей цепочке производства строительных материалов, а затем и в самом строительстве. Гениальный конструктор может придумать новый автомобиль или самолёт, которые так и останутся задумкой, пока тысячи других людей не согласятся (скорее всего, за деньги, которые тоже должны откуда-то взяться) изготовить все нужные детали и агрегаты, а потом, соединив их все вместе, провести цикл испытаний и доработок. Повсюду творческий технический гений натыкается на материальную прозу жизни; мы воочию видим результаты работы гениальных конструкторов, если сопротивление материальной среды удаётся преодолеть, но мы можем лишь догадываться о том, сколько столь же гениальных идей бездарно пропало, так и не найдя возможности воплотиться в металле.

С появлением программируемых компьютеров стало возможно создать нечто, действующее само по себе, избежав при этом сложностей, связанных с материальным воплощением. Проект дома, самолёта или автомобиля — это лишь формальное описание, по которому затем необходимо создать сам автомобиль или дом, иначе от такого проекта не будет никакого толку. Компьютерная программа — это тоже *формальное описание* того, что должно произойти, но, в отличие от технических проектов, **программа сама по себе есть готовое изделие**. Если бы Пьер Жаке-Дро мог материализовывать свои задумки, просто выполнив чертежи, он бы наверняка удивил публику чем-нибудь гораздо более сложным, нежели «Пишущий мальчик»; можно сказать, что у программистов такая возможность есть. Пожалуй, не будет преувеличением заявить, что программирование — самая творческая из всех инженерно-технических профессий, и этим программирование привлекает не только профессионалов, но и огромное число любителей; вечный вопрос о том, чего в программировании больше — техники или искусства — не решён ни в чью пользу и вряд ли когда-нибудь будет решён.

Полёт инженерной мысли, не связанный производственной рутиной, неизбежно приводит к наращиванию сложности программирования как дисциплины, и этим обусловлены некоторые особенности этой уникальной профессии. Известно, что программиста нельзя выучить, человек может стать программистом только сам — или не стать им вовсе. Высшее образование при этом желательно, поскольку хорошее знание математики, физики и других наук приводит мозги в порядок и резко повышает потенциал саморазвития. «Программистские» предметы, изучаемые

в ВУЗе, тоже могут быть полезны, давая информацию и навыки, которые в противном случае пришлось бы выискивать самостоятельно; но, наблюдая за развитием будущих программистов, можно вполне определённо сказать, что роль «программистских» предметов в этом развитии гораздо скромнее, чем принято считать: не будь преподавателя, будущий программист нашёл бы всё нужное сам; да он так и поступает, поскольку усилия преподавателей обеспечивают его потребности в специальных знаниях хорошо если на четверть. Сам будучи университетским преподавателем, я, тем не менее, должен признать, что знаком со многими прекрасными программистами, имеющими непрофильное высшее образование (химическое, медицинское, филологическое) или даже не имеющими вообще никакого диплома; с другой стороны, будучи профессиональным программистом, пусть теперь уже, возможно, и бывшим, я могу совершенно определённо сказать, что профильное университетское образование, конечно, помогло мне в плане профессионального роста, но в целом программистом я сделал себя сам, иной вариант попросту невозможен. Получается так, что высшее образование для программиста желательно, но не обязательно, а вот самообучение, напротив, категорически необходимо: если потенциальный программист не сделает себя сам, другие из него программиста не сделают и подавно.

Книга, которую вы сейчас читаете, получилась в результате попытки собрать воедино основные сведения, которые нужны при самостоятельном изучении программирования, чтобы их не приходилось выискивать в разных местах и источниках сомнительного качества. Конечно, стать программистом можно и без этой книги; существует множество различных путей, которыми можно пройти, чтобы в конце прийти к пониманию программирования; эта книга покажет вам определённые путевые точки, но даже с учётом этого ваш путь к цели останется только вашим, уникальным и не таким, как у других.

Скажу вам сразу, что одной этой книги, чтобы стать программистом, не хватит; всё, что вы можете из неё извлечь — это общее понимание того, что́ же такое программирование как вид человеческой деятельности и как приблизительно это следует делать. Кроме того, эта книга останется для вас абсолютно бесполезным ворохом бумаги, если вы решите просто читать её, не пытаясь при этом писать программы на компьютере. И ещё одно: **эта книга ничему вас не научит, если командная строка ОС Unix не станет вашим основным средством повседневной работы с машиной**.

Объяснение этому очень простое. Чтобы стать программистом, вам для начала придётся начать писать программы так, чтобы они работали; затем в какой-то момент нужно перейти от этюдов к попыткам извлечь из собственных программ какую-то пользу; и, наконец, необходимо сделать последний важнейший шаг — довести полезность своих программ до такого уровня, чтобы ими начал пользоваться кто-то кроме вас. Написать сколько-нибудь полезную программу, имеющую графический

интерфейс, довольно сложно — чтобы это сделать, нужно *уже* быть программистом, но чтобы им стать, вам, как уже было сказано, нужно начать писать полезные программы. Этот заколдованный круг можно разорвать, выбросив из рассмотрения графический интерфейс, но программы, не имеющие такового и при этом полезные, бывают только в ОС Unix, больше нигде.

К сожалению, я вынужден сейчас сказать вам ещё одну не очень приятную вещь. Далеко не каждый человек может стать программистом, и дело здесь не в уровне интеллекта или каких-то там «способностях», а в том, каковы ваши индивидуальные склонности. Программирование — это очень тяжёлая работа, требующая предельного интеллектуального напряжения, и выдержать эту пытку могут лишь те сравнительно редкие *извращенцы*, которые способны от процесса создания компьютерных программ получать удовольствие. Вполне возможно, что в ходе изучения этой книги вы поймёте, что программирование — это «не ваше»; ничего страшного, в мире есть много других хороших профессий. Если моя книга «всего лишь» позволит вам вовремя понять, что это не ваш путь, и не тратить лучшие годы жизни на бесплодные попытки учиться в ВУЗе по какой-нибудь программистской специальности — что же, это само по себе немало: лучшие годы, потраченные впустую, вам потом никто не вернёт, и чем раньше вы поймёте, что́ вам нужно (точнее, **не** нужно), тем лучше.

Впрочем, хватит о грустном. В первой, вводной части этой книги собраны сведения, которые вам потом понадобятся в программировании, но которые сами по себе программистских упражнений не требуют. На изучение вводной части у вас может уйти от одного дня до нескольких недель; за это время постарайтесь поставить себе на компьютер какой-нибудь Linux или FreeBSD и начать именно эту систему использовать в повседневной работе. Для этого вам подойдёт практически любой сколь угодно старый компьютер, который ещё не рассыпался на кусочки; вы вряд ли найдёте сейчас «живой» Pentium-1, а машины класса Pentium-II для работы многих дистрибутивов Linux вполне достаточно. Кстати, можете использовать появление в вашем хозяйстве нужной операционной системы в качестве проверки собственной готовности к дальнейшему: если прошло три-четыре недели, а ничего юниксоподобного на ваших компьютерах всё ещё нет, можете не обманывать себя: дальнейшие попытки «научиться программировать» вам просто не нужны.

Получив в своё распоряжение Unix, для начала постарайтесь как можно больше своих обычных «компьютерных дел» делать именно в нём. **Да**, там можно слушать музыку, смотреть фотографии и видео, там можно выходить в Интернет, там есть вполне адекватные заменители привычных офисных приложений, **там можно делать всё**. Поначалу, возможно, будет непривычно и из-за этого тяжело; не волнуйтесь, этот период скоро пройдёт. Добравшись до начала второй части

нашей книги, берите в руки редактор текстов, компилятор Паскаля и пробуйте. Пробуйте, пробуйте, пробуйте, пробуйте! Знайте, компьютер от ваших программ не взорвётся, пробуйте смелее. Пробуйте и так, и эдак, пробуйте и это, и то. Если какая-то задача кажется вам интересной — решайте её, от этого будет заведомо больше пользы, чем от задач из задачника. И помните: всё это должно быть «в кайф»; вымучивать программирование бесполезно.

Всем тем, кто не испугался, я искренне и от всей души желаю успехов. На то, чтобы написать первые два тома этой книги, у меня ушёл целый год; очень хочется надеяться, что он ушёл не зря.

Предисловие к первому тому

Изначально разделять книгу на тома не планировалось; эта идея возникла, когда была уже практически готова рукопись исходно запланированных семи частей. Объём рукописи существенно превысил ожидания, вынуждая обходиться без дополнительных частей и отдельных глав; кроме того, текущая финансовая ситуация проекта не позволяла немедленно издать всю книгу целиком, даже отказавшись от всего, от чего только можно отказаться. Постепенное издание в виде отдельных томов частично снимает возникшие проблемы или, по крайней мере, снижает их остроту, тем более что материал книги удалось достаточно удачно и естественно разделить на три тома, более-менее сбалансированных по объёму; при этом оказалось, что текст первого тома готов к публикации немедленно, ко второму тому желательно дописать две или три главы, что же касается третьего тома, то к настоящему времени готова только одна из по меньшей мере четырёх частей его текста; работа там предстоит достаточно серьёзная и издан третий том будет ещё не скоро, но это не страшно: материал той его части, которая уже готова, посвящённой языку Си++, несколько раз публиковался отдельной книгой и давно доступен в Интернете.

В первый том, который вы держите в руках, вошло всего две части книги из имеющихся семи. Первая из них посвящена предварительным знаниям, необходимым будущему программисту; она содержит сведения из истории, из математики (преимущественно дискретной), популярное изложение основ теории вычислимости и теории алгоритмов и, наконец, рассказывает в общих чертах, как использовать компьютер, работающий под управлением операционной системы семейства Unix. Вторая часть обычно называлась «паскалевской», что на самом деле не совсем корректно: изучение языка Паскаль как такового никогда не было целью этой книги, в том числе и её второй части, где вроде бы как раз этот язык и рассматривается. Правильнее будет сказать, что вторая часть посвящена приобретению базовых навыков написания компьютерных программ, для чего именно Паскаль подходит наилучшим образом.

Будь мир идеален, всё содержание этого тома входило бы в программу средней школы; к сожалению, пока что идеал представляется недостижимым. Так или иначе, материал первого тома призван дать общее представление о программировании и *научить, как писать программы.*

В тексте встречаются фрагменты, набранные уменьшенным шрифтом без засечек. При первом прочтении книги такие фрагменты можно безболезненно пропустить; некоторые из них могут содержать ссылки вперёд и предназначаться для читателей, уже кое-что знающих о программировании. Примеры того, **как не надо делать**, помечены вот таким знаком на полях:

Вводимые новые понятия выделены ***жирным курсивом***. Кроме того, в тексте используется *курсив* для смыслового выделения и **жирный шрифт** для выделения фактов и правил, которые желательно не забывать, иначе могут возникнуть проблемы с последующим материалом. Домашняя страница этой книги в Интернете расположена по адресу

`http://www.stolyarov.info/books/programming_intro`

Здесь вы можете найти архив примеров программ, приведённых в книге, а также электронную версию самой книги. Для примеров, включённых в архив, в тексте указаны имена файлов.

Часть 1

Предварительные сведения

1.1. Что такое компьютер: немного истории

Имея дело с многообразием компьютерных устройств, окружающих нас сегодня, мы часто забываем, что исходная функция компьютера — **считать**; большинство из нас не помнит, когда в последний раз использовали компьютер для вычислений. Впрочем, даже если попытаться это сделать, например, запустив программу «Калькулятор» или какую-нибудь цифровую таблицу вроде LibreOffice Calc или Microsoft Excel, можно заметить один любопытный факт: на рисование окошек, кнопочек, рамок таблицы и вообще на организацию диалога с пользователем компьютер при этом потратит в миллионы раз больше операций, чем на расчёты как таковые. Иначе говоря, устройство, предназначенное для проведения вычислений[1], занимается чем угодно, только не вычислениями. Понять, как так получилось, нам поможет небольшой экскурс в историю.

В качестве первой в истории вычислительной машины называют механический арифмометр Вильгельма Шиккарда, созданный в 1623 году. Машина называлась «счётными часами», поскольку была сделана из механических деталей, характерных для часовых механизмов. «Счётные часы» оперировали шестиразрядными целыми числами и способны были производить сложение и вычитание; переполнение отмечалось звоном колокольчика. До наших дней машина не сохранилась, но в 1960 году была создана работающая копия. По некоторым сведениям, машина Шиккарда могла быть и не самой первой механической

[1] Это следует даже из его названия: английское слово *computer* буквально переводится как «вычислитель», а официальный русский термин «ЭВМ» образован от слов «электронная **вычислительная** машина».

счётной машиной: известны эскизы Леонардо да Винчи (XVI в.), на которых изображен счётный механизм. Был ли этот механизм воплощён в металле, неизвестно.

Самой старой из счётных машин, сохранившихся до наших дней, является арифмометр Блеза Паскаля, созданный в 1645 году. Паскаль начал работу над машиной в 1642 году в возрасте 19 лет. Отец изобретателя имел дело со сбором налогов и вынужден был проводить долгие изнурительные подсчёты; своим изобретением Блез Паскаль надеялся облегчить работу отца. Первый образец имел пять десятичных дисков, то есть мог работать с пятизначными числами. Позднее были созданы машины, имевшие до двадцати дисков. Сложение на машине Паскаля выполнялось легко, что же касается вычитания, то для него приходилось использовать метод девятичных дополнений.

Если у нас всего пять разрядов, то перенос в шестой разряд, как и заём из него, благополучно теряется, что позволяет вместо вычитания числа выполнять *прибавление* некоторого другого числа. Например, если мы хотим вычесть из числа 500 (то есть, на пяти разрядах, из 00500) число 134 (00134), то вместо этого можно *прибавить* число 99866. Если бы у нас был шестой разряд, то получилось бы 100366, но поскольку шестого разряда нет, результат получится 00366, то есть ровно то, что нужно. Как легко догадаться, «магическое» число 99866 получено путём вычитания нашего *вычитаемого* из 100000; с точки зрения арифметики мы вместо операции $x - y$ выполняем $x + (100000 - y) - 100000$, причём последнее вычитание происходит само собой за счёт переноса в несуществующий шестой разряд.

Хитрость здесь в том, что получить из числа y число $100000 - y$ оказывается неожиданно просто. Перепишем выражение $100000 - y$ в виде $99999 - y + 1$. Поскольку число y по условиям задачи не более чем пятизначное, вычитание $99999 - y$ в столбик произойдёт без единого займа, то есть попросту **каждая цифра числа y будет заменена на цифру, дополняющую её до девятки**. Останется только прибавить единичку, и дело сделано. В нашем примере цифры 00134 заменяются на соответствующие им 99865, затем прибавляется единица и получается «магическое» 99866, которое мы прибавляли к 500, вместо того чтобы вычитать 134.

На арифмометрах Паскаля вычитание выполнялось несколько хитрее. Сначала нужно было набрать девятичное дополнение уменьшаемого (число $99999 - x$, для нашего примера это будет 99499), для чего барабаны с цифрами результата, видимые через специальные окошки, содержали по две цифры — основную и дополняющую до девятки, а сама машина была снабжена планкой, с помощью которой «ненужный» ряд цифр закрывали, чтобы он не отвлекал оператора. К набранному девятичному дополнению прибавлялось вычитаемое, в нашем примере 00134, то есть получалось число $99999 - x + y$. Однако оператор продолжал смотреть на цифры девятичных дополнений, где отображалось $99999 - (99999 - x + y)$, то есть просто $x - y$. Для чисел из нашего примера результатом прибавления стало бы число 99633, девятичное дополнение которого — число 00366 — представляет собой верный результат операции $500 - 134$.

Сейчас этот способ представляется нам чем-то вроде фокуса, любопытного, но не слишком нужного в современных реалиях. Но вот с вычислением до-

полнения, требующим прибавления единицы, мы ещё встретимся, когда будем обсуждать представление отрицательных целых чисел в компьютере.

Тридцать лет спустя знаменитый немецкий математик Готфрид Вильгельм Лейбниц построил механическую машину, способную выполнять сложение, вычитание, умножение и деление, причём умножение и деление выполнялись на этой машине примерно так, как мы выполняем умножение и деление в столбик — умножение производится как последовательность сложений, а деление — как последовательность вычитаний. В некоторых источниках можно встретить утверждение, что машина якобы умела вычислять квадратные и кубические корни; на самом деле это не так, просто вычислить корень, имея устройство для умножения, изрядно проще, чем без такового.

История механических арифмометров продолжалась достаточно долго и закончилась уже во второй половине XX столетия, когда механические счётные устройства были вытеснены электронными калькуляторами. Для нашего исторического экскурса важно одно общее свойство арифмометров: они не могли без участия человека проводить расчёты, состоящие из более чем одного действия; между тем решение даже сравнительно простых задач требует выполнения долгих последовательностей арифметических действий. Разумеется, арифмометры облегчали труд расчётчиков, но сохранялась необходимость выписывать на бумаге промежуточные результаты и набирать их вручную с помощью колёс, рычагов, в более поздних вариантах — с помощью кнопок.

Английский математик Чарльз Беббидж (1792–1871) обратил внимание[2] на то, что труд расчётчиков может быть автоматизирован полностью; в 1822 году он предложил проект более сложного устройства, известного как ***разностная машина***. Эта машина должна была интерполировать полиномы методом конечных разностей, что позволило бы автоматизировать построение таблиц разнообразных функций. Заручившись поддержкой английского правительства, в 1823 году Беббидж начал работу над машиной, но технические сложности, с которыми он столкнулся, несколько превысили его ожидания. Историю этого проекта разные источники излагают по-разному, но все сходятся на том, что общая сумма правительственных субсидий составила огромную по тем временам сумму в 17000 фунтов стерлингов; некоторые авторы добавляют, что аналогичную сумму Беббидж потратил и из своего состояния. Факт состоит в том, что работающую машину Беббидж так и не построил, причём в ходе проекта, затянувшегося почти на два десятилетия, он сам охладел к своей затее, заключив, что метод конечных разностей — это лишь одна (пусть и важная) из огромного множества расчётных задач; следующая задуманная изобретателем машина должна была

[2]На самом деле известно более ранее описание разностной машины — в книге немецкого инженера Иоганна фон Мюллера, изданной в 1788 году. Использовал ли Беббидж идеи из этой книги, достоверно не известно.

стать универсальной, то есть настраиваться на решение произвольной задачи.

В 1842 году, так и не получив никакого работающего устройства, английское правительство отказалось от дальнейшего финансирования деятельности Беббиджа. Основываясь на принципах, предложенных Беббиджем, швед Георг Шутц в 1843 году завершил построение работающей разностной машины, а в последующие годы построил ещё несколько экземпляров, один из которых продал британскому правительству, другой — правительству Соединённых Штатов. В конце XX века два экземпляра разностной машины Беббиджа были построены на основе его оригинальных чертежей, одна для музея науки в Лондоне, другая — для музея истории компьютеров в Калифорнии; таким образом было наглядно показано, что разностная машина Беббиджа могла бы работать, если бы её изготовление было завершено.

Впрочем, в историческом плане оказывается интереснее не разностная машина, а задуманная Беббиджем универсальная вычислительная машина, которую он называл ***аналитической***. Сложность этой машины была такова, что даже выполнить её чертежи Беббиджу не удалось; задуманное устройство превосходило возможности техники того времени, да и его собственные возможности тоже. Так или иначе, именно в работах Беббиджа, посвящённых аналитической машине, во-первых, возникла идея *программного управления*, то есть выполнения действий, предписанных программой; и, во-вторых, появились действия, не имеющие прямого отношения к арифметике: перенос данных (промежуточных результатов) из одного устройства хранения в другое и выполнение тех или иных действий в зависимости от результатов анализа (например, сравнения) данных.

В том же году, когда британское правительство прекратило финансирование проекта разностной машины, Беббидж прочитал в университете Турина лекцию, посвящённую в основном аналитической машине; итальянский математик и инженер Федерик Луиджи Менабреа опубликовал на французском языке[3] конспект этой лекции. По просьбе Беббиджа леди Августа Ада Лавлейс[4] перевела этот конспект на английский, снабдив свой перевод развернутыми комментариями, значительно превышающими по размеру саму статью. В одном из разделов этих комментариев приводится полный набор команд для вычисления

[3]Менабреа достаточно много своих работ публиковал именно на французском, который в те времена был более популярен в качестве международного, нежели английский.

[4]Ада Лавлейс была единственным законным ребёнком поэта Байрона, но отец видел свою дочь всего один раз в жизни, через месяц после её рождения. Согласно наиболее правдоподобной версии, своё пристрастие к математике Ада переняла от своей матери, Анны Изабеллы Байрон. В число знакомых Ады Лавлейс входили, кроме Чарльза Беббиджа, такие знаменитости, как Майкл Фарадей и Чарльз Диккенс, а её наставницей в юности была знаменитая женщина-учёный Мэри Сомервилль.

чисел Бернулли на аналитической машине; этот набор команд считается первой в истории компьютерной программой, а саму Аду Лавлейс часто называют первым программистом. Любопытно, что Ада Лавлейс, размышляя над возможностями аналитической машины, смогла уже тогда заглянуть в будущее компьютеров; помимо прочего, её комментарии содержали следующий фрагмент: *«Суть и предназначение машины изменятся от того, какую информацию мы в неё вложим. Машина сможет писать музыку, рисовать картины и покажет науке такие пути, которые мы никогда и нигде не видели»*. По сути, Ада Лавлейс заметила, что задуманная Беббиджем машина может рассматриваться как инструмент для обработки информации в широком смысле, тогда как решение расчётных математических задач представляет собой лишь частный случай такой обработки.

Если работающая разностная машина, как уже упоминалось выше, всё же была построена в середине XIX века, хотя и не Беббиджем, то идея *программируемой* вычислительной машины опередила уровень техники почти на сто лет: первые работающие вычислительные машины, управляемые программно, появились лишь во второй четверти XX века. В настоящее время считается, что хронологически первой программируемой вычислительной машиной была Z1, построенная Конрадом Цузе в Германии в 1938 году; машина была полностью механической, электричество использовалось только в моторе, приводившем механизмы в движение. Z1 использовала в работе двоичную логику, причём элементы, вычислявшие логические функции, такие как конъюнкция, дизъюнкция и т. п., были реализованы в виде наборов из металлических пластин, снабжённых хитрыми вырезами. Заинтересованному читателю мы можем порекомендовать найти в Интернете видео, демонстрирующее эти элементы на увеличенной модели: впечатление, производимое их работой, безусловно стоит потраченного времени.

Машина Z1 работала не слишком надёжно, механизмы часто заедали, искажая получаемый результат, так что практической пользы от этой машины получить не удалось, но за ней годом позже последовала Z2, использовавшая ту же механику для хранения информации («памяти»), но осуществлявшая вычислительные операции с помощью электромагнитных реле. Обе машины выполняли инструкции, получаемые с перфоленты; перематывать ленту назад они не умели, что сильно сужало их возможности, не позволяя организовать повторения участков программы, то есть циклы. Позже, в 1941 году, Цузе построил машину Z3, использовавшую только реле и хранившую программу на пластиковой перфоленте (по некоторым данным, для этого использовалась обыкновенная киноплёнка — бракованные дубли и прочие отходы деятельности киностудий). Эта машина позволяла организовывать циклы, но не имела инструкции для условного перехода, что также несколько ограничивало её возможности. Z3 разрабатывалась в качестве секрет-

ного правительственного проекта; Цузе обратился к правительству за дополнительным финансированием с целью замены реле на электронные схемы, но в этом ему было отказано. Последняя машина в этом ряду, Z4, по своим принципам была похожа на Z3, но уже, наконец, позволяла организовать ветвление. Постройка Z4 была завершена в 1944 году, и из всех машин Конрада Цузе только она уцелела, остальные были уничтожены при бомбардировке Берлина авиацией союзников.

Долгое время о работах Цузе не было известно за пределами Германии. Между тем на окрестности Второй мировой войны пришёлся по обе стороны океана настоящий бум создания вычислительных устройств, как электромеханических (в том числе основанных на реле), так и *электронных*, основанных на электронно-вакуумных лампах.

Радиолампа (см. рис. 1.1) представляет собой электронный прибор, выполненный в виде запаянной стеклянной колбы с электродами, из которой откачан воздух. В простейшей радиолампе — диоде — предусмотрены два рабочих электрода (анод и катод), а также спираль, разогревающая катод до температур, при которых начинается *термоэлектронная эмиссия*, когда отрицательно заряженные электроны покидают катод и создают в пространстве лампы своеобразное электронное облако; под действием разности потенциалов электроны притягиваются к аноду и поглощаются им. В обратном направлении заряд передавать некому: анод, оставаясь холодным, эмиссии электронов не производит, а заряженных ионов в колбе нет, потому что там вакуум. Таким образом, через диод ток может идти только в одном направлении; поскольку заряд электрона считается отрицательным, в терминах электродинамики перемещение электрического заряда происходит навстречу электронам, то есть от анода к катоду. Если изменить полярность включения диода в цепь, то электроны, покинув разогретый электрод, тут же будут притягиваться к нему обратно под действием положительного потенциала, а до второго электрода (анода, ставшего катодом в результате переполюсовки) частицы долетать не будут, отталкиваясь от него из-за наличия на нём отрицательного потенциала.

Добавив ещё один электрод — так называемую *сетку* — мы получим новый тип радиолампы, называемый *триодом*. Сетка устанавливается внутри колбы таким образом, чтобы оказаться на пути электронов, летящих от катода к аноду. При подаче на сетку отрицательного потенциала она начинает отталкивать электроны, не позволяя им достичь анода; если подать на сетку модулированный сигнал, например, полученный с микрофона, ток, проходящий через триод, будет повторять изменения потенциала на сетке, но при этом может быть гораздо сильнее. Изначально триоды как раз и предназначались для усиления сигнала.

Взяв два триода и соединив анод каждого из них с сеткой другого, мы получим устройство, называемое *триггером*. Оно может находиться в одном из двух устойчивых состояний: через один из двух триодов идёт ток (говорят, что он *открыт*), благодаря этому на сетке второго триода имеется потенциал, не позволяющий току идти через второй триод (триод *закрыт*). Кратковременно подав отрицательный потенциал на сетку открытого триода, мы тем самым прекратим ток через него, в результате второй триод откроется и закроет первый; иначе говоря, триоды поменяются ролями и триггер перейдёт в противоположное устойчивое состояние. Триггер можно использовать, например, для хранения од-

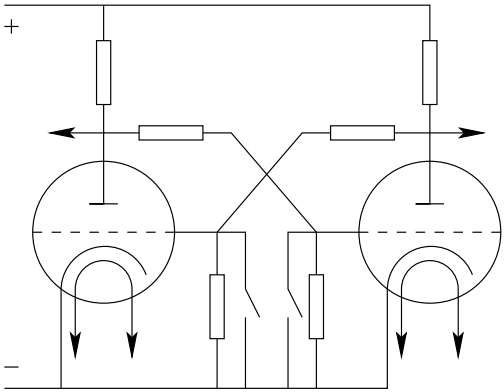

Рис. 1.1. Радиолампа (двойной триод) в действии[5] (слева); схема триггера на двух триодах (справа)

ного бита информации. Другие схемы соединения триодов позволяют построить *логические вентили*, реализующие конъюнкцию, дизъюнкцию и отрицание. Всё это позволяет использовать радиолампы для построения электронного вычислительного устройства.

За счёт отсутствия механических деталей машины на электронно-вакуумных лампах работали гораздо быстрее, но, к сожалению, сами радиолампы — предмет достаточно ненадёжный: разгерметизация колбы приводит к выходу лампы из строя; спираль, нагревающая катод, может перегореть, что также приводит к отказу лампы. Одна из первых полностью программируемых ЭВМ — ENIAC — содержала 18 000 электронных ламп, а работать машина могла только при условии, что все лампы исправны. Естественно, несмотря на беспрецедентные меры, принятые для повышения надёжности, машину очень часто приходилось ремонтировать.

Создали ENIAC американский учёный Джон Моушли и его ученик Дж. Эккерт; работы были начаты во время Второй мировой войны и финансировались военными, но, к счастью для создателей машины, завершить её до конца войны они не успели, так что проект, когда он в 1946 году был завершён, удалось рассекретить. Пионерам лампового компьютеростроения из Великобритании повезло меньше: построенные в обстановке строжайшей секретности машины Colossus Mark I и Colossus Mark II после завершения войны по личному приказу

[5] Фото с сайта Википедии; оригинал может быть загружен со страницы `https://en.wikipedia.org/wiki/File:Dubulttriode_darbiibaa.jpg`. Используемые здесь и далее изображения с сайтов фонда Wikimedia разрешены к распространению под различными вариантами лицензий Creative Commons; подробную информацию, а также оригиналы изображений в существенно лучшем качестве можно получить на соответствующих веб-страницах. В дальнейшем мы опускаем подробное замечание, ограничиваясь только ссылкой на веб-страницы, содержащие оригинальное изображение.

Черчилля[6] были уничтожены, а их автор Томми Флауэрс, повинуясь всё тому же приказу, был вынужден собственноручно сжечь всю конструкторскую документацию, что сделало воссоздание машин невозможным. Широкой публике стало известно об этом проекте лишь тридцать лет спустя, а его участники лишились заслуженного признания и были фактически отлучены от мирового развития вычислительной техники. Достижения создателей Colossus к моменту рассекречивания проекта представляли разве что исторический интерес, к тому же большая их часть была утрачена при уничтожении машин и документации.

Часто можно встретить утверждение, что машины Colossus предназначались для дешифровки сообщений, зашифрованных немецкой электромеханической шифровальной машиной Enigma, а в проекте участвовал (и едва ли не руководил им) знаменитый математик, один из основоположников теории алгоритмов Алан Тьюринг. Это не так; никакого участия в проекте Colossus Тьюринг не принимал, а машина, построенная с его непосредственным участием и действительно предназначавшаяся для взлома кодов «Энигмы», называлась Bombe, была чисто электромеханической и в принципе, строго говоря, не являлась компьютером, как и сама «Энигма». Машины Томми Флауэрса предназначались для взлома шифрограмм, составленных с помощью машины Lorenz SZ, шифр которой был гораздо более стоек к взлому, нежели шифр «Энигмы», и не поддавался электромеханическим методам.

Впрочем, Томми Флауэрсу действительно довелось некоторое время работать под руководством Тьюринга в одном из британских криптоаналитических проектов, и именно Тьюринг порекомендовал кандидатуру Флауэрса для проекта, связанного с Lorenz SZ.

Вычислительные машины, построенные на радиолампах, принято называть ***ЭВМ первого поколения***; следует обратить внимание, что по поколениям различают только *электронные* вычислительные машины, а всевозможные механические и электромеханические вычислители к ним не относят. В частности, машины Конрада Цузе электронными не были, так что они не считаются «ЭВМ первого поколения» и вообще ЭВМ.

Возможности машин этой эпохи были весьма ограниченными: из-за громоздкой элементной базы приходилось довольствоваться мизерными (по современным меркам) объёмами памяти. Тем не менее именно к первому поколению относится одно из самых важных изобретений в истории вычислительных машин — ***принцип хранимой программы***, который подразумевает, что программа в виде последовательности ***кодов команд*** хранится в той же самой памяти, что и данные, а сама память однородна и коды команд от данных ничем принципиально не

[6] По одной из версий, основной целью Черчилля было не допустить огласки того факта, что о массовых бомбардировках города Ковентри ему было известно заранее из перехваченных шифрограмм, но он ничего по этому поводу не предпринял, чтобы не выдать возможностей Британии по вскрытию немецких шифров; впрочем, мнения историков на этот счёт расходятся. Более того, версию о сознательном принесении Ковентри в жертву опровергает ряд свидетельств непосредственных участников событий того времени. Вопрос, зачем было уже после завершения войны уничтожать технику, использовавшуюся для вскрытия шифров, остаётся открытым.

отличаются. Машины, соответствующие этим принципам, традиционно называют ***машинами фон Неймана*** в честь Джона фон Неймана.

История названия довольно своеобразна. Одной из первых электронных машин, хранящих программу в памяти, стал компьютер EDVAC; его построили знакомые нам по ENIAC'у Моушли и Эккерт, причём обсуждение и проектирование новой машины они вели уже во время постройки ENIAC'а. Джон фон Нейман, участвовавший в качестве научного консультанта в проекте «Манхэттен»[7], заинтересовался проектом ENIAC в том числе и потому, что проект «Манхэттен» требовал огромных объёмов вычислений, над которыми работала целая армия девушек-расчётчиц, пользовавшихся механическими арифмометрами. Естественно, фон Нейман принял активное участие в обсуждении с Моушли и Эккертом архитектурных принципов новой машины (EDVAC); в 1945 году он обобщил результаты обсуждений в письменном документе, который известен как «Первый черновик сообщения о машине EDVAC» (*First Draft of the Report on the EDVAC*). Фон Нейман не считал документ оконченным: в этой версии текст предназначался только для обсуждения членами исследовательской группы Моушли и Эккерта, в которую входил в числе прочих учёных Герман Голдстайн. Господствующая версия исторических событий такова, что именно Голдстайн поручил перепечатать рукописный документ, поставив на его титульном листе только имя фон Неймана (что формально правильно, поскольку автором текста был фон Нейман, но не вполне корректно в свете научных традиций, так как изложенные в документе идеи были результатом коллективной работы), и затем, размножив документ, разослал заинтересованным учёным несколько десятков копий. Именно этот документ намертво связал имя фон Неймана с соответствующими архитектурными принципами, хотя, судя по всему, фон Нейман не является автором (во всяком случае, единоличным) большинства изложенных там идей. Позже фон Нейман построил ещё одну машину, IAS, в которой воплотил архитектурные принципы, изложенные в «сообщении».

С вычислительной работой, проводившейся для проекта «Манхэттен», связано много интересных историй; некоторые из них описал другой участник проекта, знаменитый физик Ричард Фейнман, в своей книге «Вы, конечно, шутите, мистер Фейнман» [7]. Там есть, в частности, такой фрагмент:

> А что касается мистера Френкеля, который затеял всю эту деятельность, то он начал страдать от компьютерной болезни — о ней сегодня знает каждый, кто работал с компьютерами. Это очень серьёзная болезнь, и работать при ней невозможно. Беда с компьютерами состоит в том, что ты с ними играешь. Они так прекрасны, столько возможностей — если чётное число, делаешь это, если нечётное, делаешь то, и очень скоро на одной-единственной машине можно делать всё более и более изощрённые вещи, если только ты достаточно умён.
>
> Через некоторое время вся система развалилась. Френкель не обращал на неё никакого внимания, он больше никем не руководил. Система действовала очень-очень медленно, а он в это время сидел в комнате, прикидывая, как бы заставить один из табуляторов автоматически печатать арктангенс X. Потом табулятор включался,

[7] Американский проект создания атомной бомбы.

> печатал колонки, потом — бац, бац, бац — вычислял арктангенс автоматически путем интегрирования и составлял всю таблицу за одну операцию.
>
> Абсолютно бесполезное занятие. Ведь у нас уже были таблицы арктангенсов. Но если вы когда-нибудь работали с компьютерами, вы понимаете, что это за болезнь — восхищение от возможности увидеть, как много можно сделать.

К сожалению, наше время слишком сильно отличается от того, когда Фейнман работал в проекте «Манхэттен» и даже от того, когда он писал свою книгу. Далеко не все, кто сейчас имеет дело с компьютерами, знают о существовании этой вот «компьютерной болезни», компьютеры стали слишком привычным делом, а компьютерные игры большинство людей находит существенно более увлекательным занятием, чем «играть» с самим компьютером, с его возможностями. С другой стороны, Фейнман абсолютно прав в том, что об этой болезни «знал каждый, кто работал с компьютерами» — просто в те времена не существовало «конечных пользователей», каждый, кто работал с компьютером, был программистом. Как ни странно, именно эта «болезнь» превращает человека в программиста. Если хотите стать программистом — постарайтесь подхватить болезнь, описанную Фейнманом.

Так или иначе, принцип хранимой программы стал однозначным прорывом в области вычислительной техники. До этого машины программировались либо перфолентами, как машины Конрада Цузе, либо вообще перемычками и тумблерами, как ENIAC; на физическое задание программы — перестановку всех перемычек и переключение тумблеров — уходило несколько дней, а затем счёт проходил за час или два, после чего снова предстояло перепрограммировать машину. Программы в те времена не писали, а скорее изобретали, ведь по сути программа была не последовательностью инструкций, а схемой соединения узлов машины.

Хранение программы в памяти в виде инструкций позволило, во-первых, не тратить огромное количество времени на смену программы: её теперь можно было считать с внешнего носителя (перфоленты или колоды перфокарт), разместить в памяти и выполнить, и происходило это достаточно быстро; конечно, на подготовку программы — на то, чтобы её придумать и потом нанести на перфокарты или перфоленты — тоже уходило много времени, но при этом не расходовалось время самой машины, стоившее огромных денег. Во-вторых, использование одной и той же памяти как для кодов команд, так и для обрабатываемых данных позволило трактовать программу как данные и создавать программы, оперирующие другими программами. Такие привычные ныне явления, как компиляторы и операционные системы, были бы немыслимы на машинах, не отвечающих определению машины фон Неймана.

Если говорить точнее, под машиной фон Неймана обычно подразумевают компьютер, обладающий следующими свойствами:

- линейность и однородность памяти: оперативная память состоит из одинаковых ячеек, каждая из которых имеет свой *адрес*; в простейшем случае в качестве адреса выступает число — номер ячейки;
- произвольный доступ к памяти: центральный процессор может в любой момент времени обмениваться информацией с любой из ячеек памяти, указав её адрес, причём скорость работы никак не зависит от того, к каким именно ячейкам обращается процессор;
- программное управление: машина работает автоматически, выполняя одну за другой команды (инструкции), составляющие *программу*;
- хранение программы в памяти: программа хранится в ячейках оперативной памяти в виде последовательности числовых кодов отдельных операций;
- неразличимость команд и данных: ячейки памяти, хранящие коды команд программы, физически никак не отличаются от ячеек, хранящих исходные, промежуточные или результирующие данные, с которыми программа должна работать; обязанность помнить, где что лежит, возложена на программиста.

Часто в этот список включают также использование двоичной системы счисления, но этот аспект трудно считать определяющим, он скорее «несколько из другой оперы».

Между тем объёмы памяти компьютеров успели несколько подрасти; так, уже упоминавшийся IAS Джона фон Неймана имел 512 ячеек памяти по 40 бит каждая. Но пока американцы продолжали в построении компьютеров ориентироваться исключительно на научные и инженерные числовые расчёты, пусть даже и с использованием хранимой программы, в Великобритании в то же самое время нашлись люди, обратившие внимание на потенциал вычислительных машин для обработки информации за пределами узкой «расчётной» области. Первым или во всяком случае одним из первых компьютеров, исходно предназначавшихся для целей более широких, нежели числовые расчёты, считается LEO I, разработанный в британской компании J. Lyons & Co.; примечательно, что эта фирма, занимавшаяся поставками пищевых продуктов, ресторанным и отельным бизнесом, не имела никакого отношения к машиностроительной отрасли. В 1951 году только что построенный компьютер взял на себя изрядную часть функций бухгалтерии и финансового анализа компании, причём собственно вычисления как таковые составляли хотя и заметную, но отнюдь не самую большую долю выполняемых машиной операций. Принимая исходные данные с перфолент и выводя результаты на текстовое печатающее устойство, машина позволила автоматизировать подготовку зарплатных ведомостей и других подобных документов. Пророчество Ады Лавлейс начало потихоньку сбываться: объектом работы для компьютерной программы стала *информация*, при этом математические расчёты — это важный, но отнюдь не единственный случай её обработки.

Тем временем растущий уровень техники неотвратимо приближал революцию в компьютеростроении. Основным нововведением, обусло-

вившим революцию, стал ***полупроводниковый транзистор*** — электронный элемент, который со схемотехнической точки зрения очень похож на радиолампу-триод. Транзистор, как и триод, имеет три контакта, которые обычно называются «базой», «эмиттером» и «коллектором» (или «затвором», «истоком» и «стоком» для разновидности транзисторов, называемых ***полевыми***). При изменении напряжения на базе относительно эмиттера (на затворе относительно истока) изменяется ток между эмиттером и коллектором (между истоком и стоком). В аналоговой электронике оба устройства — и лампа-триод, и полупроводниковый транзистор — используются для усиления сигнала за счёт того, что пропускаемые токи между анодом и катодом триода, а равно и между эмиттером и коллектором транзистора могут быть гораздо больше, нежели сигналы, подаваемые для их «закрытия» соответственно на сетку или базу. В цифровых схемах мощности не важны, гораздо важнее эффект *управления* как таковой. В частности, как и два триода, два транзистора позволяют составить из них триггер, при этом ток, идущий через один транзистор, закрывает второй, и наоборот.

Считается, что первый работающий транзистор был создан в 1947 году в Bell Labs, а в качестве авторов изобретения называют Уильяма Шокли, Джона Бардина и Уолтера Браттейна; спустя несколько лет за это изобретение транзистора им была присуждена Нобелевская премия по физике. Ранние транзисторы были громоздки, ненадёжны и неудобны в работе, однако быстрое совершенствование технологии выращивания кристаллов позволило наладить серийный выпуск транзисторов, которые в сравнении с радиолампами были, во-первых, довольно миниатюрны; во-вторых, для транзисторов не требовался разогрев катода, так что электричества они потребляли тоже гораздо меньше; наконец, опять же в сравнении с лампами, транзисторы были практически безотказны: конечно, они тоже иногда выходят из строя, но это скорее чрезвычайное происшествие, тогда как выход из строя лампы — это просто штатное событие, а сами радиолампы рассматривались скорее как расходный материал, нежели как постоянные узлы конструкции.

Вторым серьёзным изобретением, определившим смену поколений компьютеров, стала память на магнитных сердечниках. Банк такой памяти (рис. 1.2, справа) представлял собой прямоугольную сетку из проводов, в узлах которой располагались ферритовые колечки; каждое колечко хранило один бит информации. В сравнении с громоздким устройством из трёх или четырёх радиоламп, применявшимся для хранения одного бита в ЭВМ первого поколения, это был настоящий прорыв. Компьютеры, построенные на *твердотельных электронных компонентах*, прежде всего на транзисторах, принято называть ***компьютерами второго поколения***. Если компьютеры первого поколения занимали целые здания, то машина второго поколения умещалась в одной комнате; потребление электроэнергии резко снизилось, а воз-

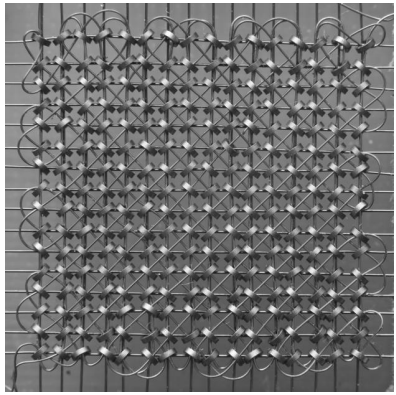

Рис. 1.2. Транзистор (слева); банк памяти на ферритовых сердечниках (справа)[8]

можности, и прежде всего объём оперативной памяти, существенно возросли. Также возросла надёжность машин, поскольку транзисторы выходят из строя гораздо реже, чем радиолампы. Существенно упала стоимость ЭВМ в денежном выражении. Первые полностью транзисторные вычислительные машины были построены в 1953 году, а уже в 1954 году компания IBM выпустила машину *IBM 608 Transistor Calculator*, которую называют первым коммерческим компьютером.

Следующий технический прорыв произошел с изобретением интегральных схем — полупроводниковых устройств, в которых на одном кристалле располагается несколько (в современных условиях — до нескольких миллиардов) элементов, таких как транзисторы, диоды, сопротивления и конденсаторы. Компьютеры, построенные на основе интегральных схем, принято относить к ***третьему поколению***; несмотря на всё ещё очень высокую стоимость, эти машины стало возможно выпускать массово — вплоть до десятков тысяч экземпляров. Центральный процессор такого компьютера представлял собой шкафчик или тумбочку, набитую электроникой. По мере совершенствования технологии микросхемы становились всё компактнее, а общее их количество в составе центрального процессора неуклонно снижалось. Наконец в 1971 году произошёл очередной переход количества в качество: были созданы микросхемы, заключающие в себе весь центральный процессор целиком. Какая из микросхем стала первым в истории ***микропроцессором***, доподлинно не известно; чаще всего таковым называют Intel 4004, про который можно, по крайней мере, точно сказать, что это был первый микропроцессор, доступный на рынке. Согласно некоторым источникам, первенство в этом деле следует отдать чипу MP944, который приме-

[8] Фото с сайта Викимедиа Коммонс; `http://commons.wikimedia.org/wiki/File:KL_CoreMemory_Macro.jpg`.

нялся в авионике истребителя F-14, однако широкая публика об этой разработке, как водится, ничего не знала вплоть до 1997 года.

Появление микропроцессоров позволило «упаковать» компьютер в настольный прибор, известный как «персональный компьютер». С этого момента принято отсчитывать историю ***четвёртого поколения ЭВМ***, каковая продолжается до наших дней[9].

Как можно заметить, в наше время компьютеры используются для обработки любой информации, которую можно записать и воспроизвести. Кроме традиционных баз данных и текстов, к которым сводилась электронная информатика в середине XX века, компьютеры успешно обрабатывают записанный звук, изображения, видеозаписи; имеются, пусть и в зачаточном состоянии, попытки обрабатывать тактильную информацию — в практическом применении это пока только брайлевские дисплеи для слепых, но инженеры не оставляют попыток создания всевозможных электронных перчаток и прочих подобных устройств. Гораздо хуже обстоят дела со вкусом и запахом, но вкусовая и обонятельная информация вообще на современном уровне техники не поддаётся обработке; можно не сомневаться, что если когда-нибудь будет найден способ записи и воспроизведения вкуса и запаха, компьютеры смогут работать и с этими видами информации.

Конечно, иногда компьютеры используются и для числовых расчётов; существует даже особая индустрия производства так называемых ***суперкомпьютеров***, исходно предназначенных исключительно для решения расчётных задач большого объёма. Современные суперкомпьютеры имеют в своём составе десятки тысяч процессоров и в большинстве случаев производятся в единичных экземплярах; в целом суперкомпьютеры — это скорее исключение из общего правила, тогда как большинство применений компьютеров имеет с численными расчётами весьма мало общего. Естественным образом может возникнуть вопрос — почему же в таком случае компьютеры до сих пор продолжают называться компьютерами? Не лучше ли использовать какой-нибудь другой термин, например, назвать их инфоанализаторами или инфопроцессорами? Как ни странно, это совершенно ни к чему; дело в том, что *вычислять* можно не только числа и не только по формулам. Если мы припомним понятие ***математической функции***, то немедленно обнаружим, что и область её определения, и область её значений могут быть *множествами произвольной природы.* Как известно, обрабатывать любую информацию можно лишь при условии, что она представлена в какой-то объективной форме; более того, цифровые компьютеры требу-

[9]Как ни странно, за минувшие сорок лет никаких качественно новых усовершенствований не предложено. Японский проект «компьютеров пятого поколения» существенных результатов не дал, тем более что опирался он не на технологическое развитие аппаратной базы, а на альтернативное направление развития программного обеспечения.

ют дискретного представления информации, а это уже не что иное, как *представление в форме цепочек символов в некотором алфавите*, или попросту *текстов*; отметим, что именно такое представление произвольной информации рассматривается в теории алгоритмов. При таком подходе любые преобразования информации оказываются функциями из множества текстов во множество текстов, а любая обработка информации становится *вычислением функции* Получается, что компьютеры по-прежнему занимаются именно *вычислениями* — пусть и не чисел, а произвольной информации.

1.2. История ОС Unix

В конце 1960-х годов консорциум в составе концерна General Electrics, Массачусетского технологического института (MIT) и исследовательской компании Bell Laboratories (на тот момент — подразделения AT&T) разрабатывали операционную систему MULTICS. О проекте MULTICS иногда говорят как о неудачном; так или иначе, Bell Labs в некий момент из проекта вышла. В число сотрудников Bell Labs, участвовавших в проекте MULTICS, входил Кен Томпсон. По одной из легенд, тогда его интересовала новая на тот момент область программирования — компьютерные игры. В силу дороговизны вычислительной техники того времени у Кена Томпсона были определенные сложности с использованием компьютеров для развлекательных целей, поэтому он заинтересовался имевшейся в Bell Labs машиной PDP-7; эта машина была уже морально устаревшей и, как следствие, претендентов на неё было не так много. Системное программное обеспечение, входившее в стандартный для той машины комплект, Томпсона не устроило, и, пользуясь опытом, полученным в проекте MULTICS, он написал для PDP-7 свою операционную систему. Первоначально система Томпсона была *двух*задачной, то есть позволяла запуск двух независимых процессов — по числу подключенных к PDP-7 терминалов [2].

Название UNICS (по аналогии с MULTICS) в шутку предложил Брайан Керниган. Название закрепилось, только последние буквы CS были позже заменены на одну X (произношение при этом не изменилось). К Кену Томпсону в его разработке присоединился Деннис Ритчи, и вдвоём они перенесли систему на более совершенный миникомпьютер PDP-11. Тогда же возникла идея переписать систему (по крайней мере, как можно бóльшую её часть) на языке высокого уровня. Томпсон попытался использовать для этого усечённый диалект языка BCPL, который он назвал «B» (читается «би»), но язык оказался для этого слишком примитивен: в нём не было даже структурных данных. Ритчи предложил расширить язык; для названия получившегося языка авторы

использовали следующую букву английского алфавита после «B» — букву «C», которую, как известно, по-английски называют «Си».

В 1973 году на Си удалось переписать созданную Томпсоном систему. Для того времени это был более чем сомнительный шаг: господствовала точка зрения, что высокоуровневое программирование с уровнем операционных систем принципиально несовместимо. Время показало, однако, что именно этот шаг определил на много лет тенденции развития индустрии. Язык программирования Си и операционная система Unix сохраняют популярность спустя более чем 40 лет после описываемых событий. По-видимому, причина в том, что Unix оказался первой операционной системой, переписанной на языке высокого уровня, а Си стал этим языком.

В 1974 году вышла статья Томпсона и Ритчи, в которой они рассказали о своих достижениях. PDP-11 на тот момент была машиной весьма популярной, установленной во многих университетах и других организациях, так что после выхода в свет статьи нашлось немало желающих попробовать воспользоваться новой системой. На этом историческом этапе важную роль сыграло особое положение компании AT&T: антимонопольные ограничения не позволяли ей участвовать в компьютерном бизнесе, как и вообще в любом бизнесе за пределами телефонии. В связи с этим копии Unix с исходными текстами предоставлялись всем желающим на некоммерческой основе. Согласно одной из легенд, Кен Томпсон подписывал каждый экземпляр, записанный на бобину магнитной ленты, словами «с любовью, Кен» (love, ken) [3]. Следующим серьёзным шагом стал перенос Unix на новую архитектуру. Эта идея была выдвинута Деннисом Ритчи и Стефаном Джонсоном и опробована на машине Interdata 8/32. В рамках этого проекта Джонсон разработал *переносимый компилятор языка Си*, ставший, кстати, едва ли не первым переносимым компилятором в истории. Перенос системы был завершён в 1977 году.

Важнейший вклад в развитие Unix внесли исследователи из университета Беркли. Одна из наиболее популярных веток Unix, BSD, представленная в настоящее время такими системами, как FreeBSD, NetBSD, OpenBSD и BSDi, была создана именно там; собственно говоря, акроним BSD означает Berkley Software Distribution. Исследования, связанные с Unix, начались здесь в 1974 году; определённую роль сыграли лекции Кена Томпсона, прочитанные в Беркли в 1975-1976 гг. Первая версия BSD была создана в 1977 году Биллом Джоем.

В 1984 году с AT&T после раздробления одного из её подразделений были сняты антимонопольные ограничения; менеджмент AT&T начал стремительную коммерциализацию Unix, свободное распространение исходных текстов Unix было прекращено. Последующие годы ознаменовались противостояниями и изнурительными судебными тяжбами между разработчиками Unix, в частности — между всё той же AT&T и

компанией BSDi, пытавшейся продолжать разработки на основе BSD. Неясности с юридическим статусом BSD затормозили развитие всего Unix-сообщества. Начиная с 1987 года в Беркли проводились работы по удалению кода, являющегося собственностью AT&T, из системы. Правовые споры были урегулированы лишь в 1993 году, когда AT&T продала своё подразделение, занимавшееся Unix (Unix Software Labs, USL) фирме Novell; юристы последней идентифицировали *три* из 18 000 (!) файлов, составлявшие действительный предмет спора, и заключили с университетом Беркли соглашение, устранившее разногласия.

Пока разработчики Unix были заняты междоусобицей, рынок оказался заполнен дешёвыми компьютерами на основе процессоров Intel и операционными системами от Microsoft. Появившийся ещё в 1986 году процессор Intel 80386 был пригоден для Unix; делались и попытки переноса BSD на платформу i386, однако (не в последнюю очередь из-за правовых проблем) до начала 1992 года об этих разработках ничего слышно не было.

Другая интересная линия событий прослеживается с 1984 года, когда Ричард Столлман основал Фонд свободного программного обеспечения и издал соответствующий идеологический манифест. Нарождающееся общественное движение для начала поставило себе целью создать свободную операционную систему. По некоторым сведениям, именно Столлман в 1987 году убедил исследователей из Беркли в необходимости очистки BSD от кода, находящегося в собственности AT&T. Сторонники Столлмана успели создать существенное количество свободных программных инструментов, но без полностью свободного ядра ОС цель оставалась всё же далека. Положение изменилось лишь в начале 1990-х. В 1991 году финский студент Линус Торвальдс начал работу над ядром Unix-подобной операционной системы для платформы i386, причём в этой работе код из других операционных систем не использовался вообще.

Как рассказывает сам Торвальдс, его творение сначала задумывалось как эмулятор терминала для удалённого доступа к университетскому компьютеру. Соответствующая программа под Minix его не удовлетворяла. Чтобы заодно разобраться в устройстве i386, Торвальдс решил написать свой эмулятор терминала в виде программы, не зависящей от операционной системы. Эмулятор терминала предполагает два встречных потока данных, для обработки которых Торвальдс сделал планировщик времени центрального процессора, фактически делающий то же самое, что делают планировщики в ядрах мультизадачных операционных систем. Позже автору потребовалась перекачка файлов, так что эмулятор терминала был снабжён драйвером дисковода; в итоге автор с удивлением обнаружил, что пишет операционную систему [4].

Свои промежуточные результаты Торвальдс открыто публиковал в Интернете, что позволило сначала десяткам, а затем и сотням добровольцев присоединиться к разработке. Нынешние исходные тексты ядра Linux включают код, написанный десятками тысяч людей. Одним из

последствий этого становится принципиальная невозможность «купить» Linux: у ядра как произведения, защищённого авторским правом, слишком много правообладателей, чтобы можно было всерьёз говорить о заключении каких-то соглашений с ними всеми. Единственный вариант лицензии, на условиях которой можно использовать ядро Linux — это лицензия GNU GPL v.2, изначально (с подачи Столлмана) принятая для исходных текстов ядра; одна из особенностей этой лицензии состоит в том, что каждый программист, вносящий свой авторский вклад в ядро, самим фактом такого вклада принимает условия GNU GPL, то есть соглашается предоставить результаты своей работы всем желающим на определяемых ею условиях.

Новая операционная система получила название Linux по имени своего создателя. Примечательно, что такое название дал системе один из сторонних участников проекта. Сам Торвальдс планировал назвать систему «Freax». Самый первый публично доступный код (версия 0.01) появился в 1991 году, первая официальная версия (1.0) — в 1994, вторая — в 1996. Следует отметить (и это также отмечает сам Линус Торвальдс), что немаловажную роль в стремительном взлёте Linux сыграла судебная война между AT&T и университетом Беркли, мешавшая распространению BSD на i386. Linux получил изрядную фору на старте, в итоге оставив BSD на вторых ролях: в наше время BSD-системы распространены меньше, хотя по-прежнему активно используются. Созданное Торвальдсом ядро решило главную проблему возглавляемого Ричардом Столлманом общественного движения: полностью свободная операционная система наконец появилась. Более того, Торвальдс принял решение использовать для ядра предложенную Столлманом лицензию GNU GPL, так что Столлману и его единомышленникам осталось только заявить о достижении поставленной цели.

Эпоха конца 1980-х — начала 1990-х породила рассогласованность в Unix-сообществе, снижавшую переносимость программ по причине различий в интерфейсах. Отчасти проблему решает появившийся стандарт POSIX 1003.1, описывающий основные системные вызовы Unix и созданный по принципу пересечения основных ветвей развития Unix (на тот момент — BSD и System V). POSIX представляет собой редкий пример технического стандарта, от которого имеется ощутимая польза; к сожалению, о большинстве существующих технических стандартов этого сказать нельзя.

Сейчас торговая марка «Unix» не используется для именования конкретных операционных систем. Вместо этого речь идёт о ***Unix-подобных операционных системах***, составляющих целое семейство. По популярности лидируют Linux, представленный несколькими сотнями вариантов дистрибутивов различных производителей и (с некоторым отрывом) FreeBSD. Обе системы распространяются свободно.

Кроме того, нельзя не отметить коммерческие системы семейства Unix, среди которых наиболее известны SunOS/Solaris и AIX.

Выдержав более чем тридцатилетнюю историю, Unix (уже не как конкретная операционная система, а как общий подход к их построению) совершенно не выглядит устаревшим, хотя при этом практически не претерпевал революционных изменений с середины 1970-х годов. Даже создание графической надстройки X Window не внесло существенных изменений в основы Unix. Отметим, что небезызвестный Android — это ничто иное, как Linux, снабжённый своей (т. е. созданной компанией Google специально для Android) графической оболочкой; то же самое обнаруживается на компьютерах от Apple: MacOS X представляет собой систему-потомка BSD, точнее, потомком BSD является система Darwin, а MacOS X — это Darwin со специально созданной графической подсистемой.

1.3. Как компьютер работает

1.3.1. Процессор, память, шина

В нашем зале есть машина,
Сквозь неё проходит шина,
А по шине взад-вперёд
Информация идёт.

Внутреннее устройство практически всех современных компьютеров базируется на одних и тех же принципах. Основой компьютера служит ***общая шина***, которая представляет собой, грубо говоря, много (несколько десятков) параллельных проводов, называемых *дорожками*. К шине подключаются ***центральный процессор*** (ЦП, англ. CPU от *central processing unit*), ***оперативное запоминающее устройство*** (ОЗУ, англ. RAM от *random access memory*, то есть *память случайного доступа*) и ***контроллеры***, через которые производится управление остальными устройствами компьютера. Через шину центральный процессор осуществляет взаимодействие с остальными компонентами компьютера; оперативная память и контроллеры устроены так, чтобы игнорировать любую информацию, проходящую по шине, кроме той, которая адресована конкретно данному банку памяти или данному контроллеру. Для этого часть дорожек шины выделяется под *адрес*; эту часть дорожек шины называют ***шиной адресов***. Дорожки, по которым производится передача информации, называют ***шиной данных***, а те дорожки, по которым передаются управляющие сигналы — ***шиной управления***. Со схемотехнической точки зрения каждая дорожка может находиться в состоянии логической единицы (дорожка «подтянута»

к напряжению питания схемы) или нуля (дорожка соединена с «землёй», то есть нулевым уровнем напряжения); определённая комбинация нулей и единиц на шине адресов как раз и составляет адрес, и все устройства, кроме центрального процессора, включаются в работу с шиной только тогда, когда состояние шины адресов соответствует их адресу, а всё остальное время они не обращают на шину никакого внимания и ничего в неё не передают, чтобы не мешать работе процессора с другими устройствами.

Оперативное запоминающее устройство, или просто ***память***, состоит из одинаковых ***ячеек памяти***, каждая из которых имеет свой уникальный ***адрес***, отличающий эту ячейку от других. С ячейкой памяти можно проделать всего две операции: записать в неё значение и прочитать из неё значение. Для выполнения этих операций центральный процессор устанавливает на адресной шине адрес нужной ему ячейки, а затем, используя шину управления, передаёт электрический импульс, который заставляет выбранную ячейку — то есть ту, чей адрес выставлен на шине, и никакую другую — передать в шину данных своё содержимое (операция чтения) или, наоборот, установить своё новое содержимое в соответствии с состоянием шины данных (операция записи). Старое содержимое ячейки при этом теряется. По шине данных информация передаётся параллельно: например, если ячейка содержит, как это обычно бывает, восемь разрядов (нулей и единиц), то для передачи данных при её чтении и записи используется восемь дорожек; при выполнении операции чтения ячейка памяти должна установить на этих дорожках логические уровни, соответстветствующее хранящимся в ней цифрам, а при операции записи, наоборот, установить хранящиеся в ячейке цифры в соответствии с логическими уровнями дорожек данных. Для хранения значений часто используется несколько ячеек памяти, идущих подряд, то есть имеющих соседние адреса, а разрядности шины данных обычно хватает на одновременную передачу информации нескольких ячеек.

Следует отметить, что оперативная память — это электронное устройство, функционирование которого требует подачи электропитания. **При выключении питания информация, хранящаяся в ячейках памяти, немедленно и безвозвратно теряется**.

Память компьютера ни в коем случае не следует путать с дисковыми запоминающими устройствами, где хранятся файлы. С памятью центральный процессор может непосредственно взаимодействовать через шину; работать с дисками и другими устройствами процессор сам по себе не может, для этого на процессоре нужно выполнять специальные довольно сложные программы, которые называются ***драйверами***. Драйверы организуют работу с диском и другими внешними устройствами через контроллеры путём передачи определённой управляющей информации.

Некоторые блоки памяти могут физически представлять собой не оперативную, а *постоянную* память. Такая память не поддерживает операцию записи, то есть изменить её содержимое невозможно, во всяком случае, выполнением операций процессора; с другой стороны, информация, записанная в такую память, не стирается при выключении питания. Центральный процессор никак не различает между собой ячейки оперативной и постоянной памяти, поскольку при выполнении операции чтения они работают абсолютно одинаково. Обычно в постоянную память при изготовлении компьютера записывается некая программа, предназначенная для тестирования аппаратуры компьютера и подготовки его к работе. Эта программа начинает выполняться, когда вы включаете компьютер; её задача — найти, откуда можно загрузить ***операционную систему***, затем загрузить её и отдать ей управление; всё остальное, в том числе запуск пользовательских программ — забота операционной системы. Подробный разговор об операционных системах нам предстоит позднее; пока отметим только, что **операционная система — это такая программа**, а не что-то иное, то есть её тоже *написали программисты*; от всех остальных программ операционная система отличается лишь тем, что, запустившись на компьютере раньше других, она получает доступ ко всем его возможностям, при этом все остальные программы запускаются уже силами операционной системы, а она все остальные программы запускает так, чтобы им никакого доступа к возможностям компьютера не дать; пользовательские программы могут лишь преобразовывать информацию в отведённой им памяти, а для всего остального — даже для того, чтобы просто закончить работу — вынуждены обращаться к операционной системе.

1.3.2. Принципы работы центрального процессора

Сам центральный процессор представляет собой электронную схему, обычно выполненную в виде микропроцессора (то есть в виде одной микросхемы); его основное и единственное назначение — выполнять простейшие действия, заданные командами, из которых состоит программа. В составе центрального процессора обычно присутствуют так называемые ***регистры*** — запоминающие устройства, способные хранить от нескольких до нескольких десятков двоичных разрядов; основную работу процессор производит над информацией, хранящейся в регистрах. В число операций, которые может выполнять процессор, обязательно входят *чтение и запись ячеек памяти*, при которых информация передаётся через шину из оперативной памяти в процессор или обратно.

Среди операций, выполняемых центральным процессором, обязательно имеется арифметика — как минимум сложение и вычитание, хотя на всех современных процессорах присутствует также умножение и деление; примером арифметического действия может быть предписа-

ние «возьми значения из второго и четвёртого регистров, сложи их, а результат запиши обратно во второй регистр». Некоторые процессоры могут производить такие действия не только над регистрами, но и над группами ячеек памяти. Есть и другие действия, такие как копирование информации из одного регистра в другой, логические операции, всевозможные служебные действия вроде переходов к выполнению команд из другого места памяти; все вместе они образуют ***систему команд*** конкретного процессора. Подробное знакомство с системой команд реально существующего процессора нам предстоит позднее, во втором томе нашей книги.

Каждое элементарное действие, выполняемое центральным процессором (машинная команда), обозначается ***кодом операции*** или, как его часто называют, ***машинным кодом***. Программа, состоящая из таких кодов, располагается в ячейках памяти; один из регистров процессора, так называемый ***счётчик команд***[10], содержит адрес той ячейки памяти, в которой располагается следующая инструкция, предназначенная к выполнению.

Процессор работает, раз за разом выполняя ***цикл обработки команд***. В начале этого цикла из ячеек памяти, на которые указывает[11] счётчик команд, считывается код очередной команды. Сразу после этого счётчик команд меняет своё значение так, чтобы указывать на следующую команду в памяти; например, если только что прочитанная команда занимала три ячейки памяти, то счётчик команд увеличивается на три. Схемы процессора дешифруют код и выполняют действия, предписанные этим кодом: например, это может быть уже знакомое нам предписание «взять содержимое двух регистров, сложить, а результат поместить обратно в один из регистров» или «скопировать число из одного регистра в другой» и т. п. Когда действия, предписанные командой, будут исполнены, процессор возвращается к началу цикла обработки команд, так что следующий проход этого цикла выполняет уже следующую команду, и так далее до бесконечности (точнее, пока процессор не выключат).

Некоторые машинные команды могут изменить последовательность выполнения команд, предписав процессору перейти в другое место программы (то есть, попросту говоря, в явном виде изменить текущее значение счётчика команд). Такие команды называются ***командами перехода***. Различают ***условные*** и ***безусловные*** переходы; команда условного перехода сначала проверяет истинность некоторого условия и производит переход только если условие выполнено, тогда как команда

[10] Английское название этого регистра — *instruction pointer*, то есть «указатель на инструкцию»; устоявшийся в русскоязычной литературе термин «счётчик команд», как и оригинальный термин *program counter*, использовавшийся, например, на PDP, не столь удачен, ведь этот «счётчик» на самом деле ничего не считает.

[11] Выражение вида «нечто указывает на ячейку памяти» является синонимом выражения «нечто содержит адрес ячейки памяти».

безусловного перехода просто заставляет процессор продолжить выполнение команд с заданного адреса без всяких проверок. Процессоры обычно поддерживают также переходы с запоминанием точки возврата, которые используются для вызова подпрограмм.

1.3.3. Внешние устройства

Центральный процессор и память (оперативная и постоянная) по сути как раз и образуют то, что называется компьютером, но если бы у нас были только они, толку от такого компьютера было бы на удивление мало. Во-первых, компьютеру как-то нужно получать программы и данные из внешнего мира, а результаты работы каким-то образом отдавать обратно. Во-вторых, содержимое оперативной памяти мгновенно теряется при выключении питания, так что для сколько-нибудь долговременного и надёжного хранения информации она не годится. Ясно, что компьютеру нужны ещё какие-то устройства помимо оперативной памяти и процессора. В ранних компьютерных системах все устройства подключались непосредственно к центральному процессору, но очень быстро стало ясно, что это неудобно: для каждого нового устройства центральный процессор приходилось переделывать; между тем внешние устройства обычно гораздо проще, чем центральный процессор, и появляются, естественно, быстрее, чем новые процессоры. Кроме того, разнообразных внешних устройств слишком много. В центральном процессоре попросту технически невозможно организовать поддержку всех или хотя бы значительного количества таких устройств. Всё это привело к возникновению идеи ***общей шины***, описанной выше.

Подключение внешних устройств к общей шине производится через ***контроллер*** — электронную схему, которая может взаимодействовать через шину с центральным процессором; все контроллеры делают это одинаково, вне зависимости от того, какими устройствами они управляют, то есть с точки зрения процессора все контроллеры оказываются «на одно лицо». При этом «другим концом» каждый контроллер подключается непосредственно к своему внешнему устройству и осуществляет управление его работой. Взаимодействие центрального процессора с контроллером строится на уже знакомых нам операциях чтения и записи; более того, на некоторых архитектурах ячейки памяти и контроллеры находятся в одном общем пространстве адресов и оказываются взаимозаменяемы, то есть центральный процессор «не знает», с чем он имеет дело — с настоящей памятью или с контроллером; впрочем, более популярен подход, когда контроллеры имеют отдельное пространство адресов[12]; в этом случае говорят об ***адресах портов ввода-вывода***. Один контроллер может поддерживать один или больше таких «портов»,

[12]Фактически при этом в компьютере имеются две разные шины — одна для ячеек памяти, вторая для контроллеров.

то есть при работе с шиной отзываться на несколько разных адресов. Операции чтения из порта ввода-вывода и записи в такой порт с точки зрения центрального процессора выглядят совершенно так же, как и операции чтения-записи ячеек памяти, но контроллеры, в отличие от ячеек памяти, присланные им в ходе «записи» значения не запоминают, а воспринимают их как предписания что-то сделать; в ходе «чтения» порта контроллеры выдают не какое-то заранее сохранённое значение, а такое значение, которое как-то связано с состоянием самого контроллера, то есть позволяет узнать, например, завершена ли очередная операция, готов ли контроллер к выполнению следующей операции, не обнаружены ли какие-либо неисправности и т. п.

Сколько портов поддерживает конкретный контроллер, какие ему можно отдавать предписания и каковы коды этих предписаний, что означают значения, которые прочитываются из его портов — всего этого центральный процессор не знает, потому что всё это зависит от конкретного контроллера, а для другого контроллера всё может быть совсем не так. Для работы с каждым конкретным контроллером нужна специальная *программа*, которая, как мы уже говорили, называется ***драйвером***; выполняясь на центральном процессоре, программа-драйвер даёт команды записи и чтения портов ввода-вывода «своего» контроллера, решая поставленные ей задачи. Обычно драйвер является частью операционной системы или становится такой частью после его загрузки. Например, если пользователь запустил некую программу, а этой программе потребовалась запись в файл на диске, то для этого программа обратится к операционной системе с просьбой записать такие-то данные в такой-то файл, операционная система вычислит, в каком месте диска находится или должен находиться соответствующий файл, и, обратившись, в свою очередь, к драйверу (то есть фактически к своей обособленной части), потребует записать определённые данные в определённое место диска; после этого драйвер, вооружась имеющимся у него знанием возможностей контроллера диска, сначала выполнит несколько операций записи в порты ввода-вывода, что заставит контроллер начать операцию записи; затем, выполнив операции чтения из портов, узнает о результатах операции и сообщит о них операционной системе[13].

1.3.4. Иерархия запоминающих устройств

Информация в вычислительной системе может запоминаться и храниться устройствами различного типа в зависимости от того, насколько оперативным должен быть доступ к данной информации, насколько долговременным должно быть её хранение и каков её объём. Иерархия

[13] На самом деле в случае с диском всё будет несколько сложнее; более подробное обсуждение происходящего оставим до части нашей книги, посвящённой операционным системам.

Рис. 1.3. Иерархия запоминающих устройств

запоминающих устройств схематически показана на рис. 1.3. Наиболее оперативно доступна информация в регистрах центрального процессора. Однако объём регистровой памяти задаётся раз и навсегда при проектировании процессора и увеличен быть не может; этот объём ограничен, т. к. каждый новый регистр ЦП увеличивает сложность схемы ЦП, требует введения дополнительных инструкций и в целом может существенно повысить стоимость процессора.

Кеш-память предназначена для увеличения скорости доступа к данным, находящимся в оперативной памяти. Различают ***кеш первого уровня***, физически реализованный на одной микросхеме с ЦП, и ***кеш второго уровня***, представляющий собой отдельную конструкцию, связанную с процессором непосредственно (без использования общей шины). В кеш-памяти дублируются данные из оперативной памяти, наиболее часто используемые выполняющейся программой. Скорость доступа к кешу существенно выше, чем к оперативной памяти, поскольку для взаимодействия процессора с кешем не нужно задействовать шину — а шина как раз работает довольно медленно из-за её сравнительно большой длины, причём её работа не может быть существенно ускорена, так как ограничена скоростью света и другими физическими причинами. При этом сам кеш имеет достаточно сложное устройство, а объём его сравнительно невелик.

Оперативная память[14] является основным хранилищем программ и данных, находящихся в непосредственной обработке. Объём оперативной памяти может быть сравнительно большим, а её стоимость в последние годы снизилась. Тем не менее, её объема может не хватить. Кроме того, содержимое оперативной памяти, кеша и регистров теряется с выключением компьютера, так что для долговременного хранения данных эти виды запоминающих устройств непригодны.

[14]Можно также встретить термины «основная память» и «оперативное запоминающее устройство» (ОЗУ). В англоязычной литературе используется термин RAM (Random Access Memory), который можно перевести как «память произвольного доступа».

На следующем уровне иерархии находятся магнитные диски или, говоря в общем, устройства долговременного хранения, позволяющие производить доступ к данным в произвольном порядке. Кроме собственно магнитных дисков, к устройствам такого класса относятся, например, накопители на flash-картах. Ныне вышедшие из употребления магнитные барабаны также относились к этому классу. Объём таких устройств может быть на порядки больше, чем объём ОЗУ, а стоимость — существенно ниже. Кроме того, сохранённая на дисках информация не теряется при выключении питания и может храниться долгое время. С другой стороны, для доступа к дискам требуются медленные (в сравнении со скоростью процессора и ОЗУ) операции ввода-вывода; более того, процессор не в состоянии непосредственно обращаться к дискам, так что данные для их использования должны быть предварительно скопированы в оперативную память.

Срок хранения информации на дисках может составлять годы, но он всё же ограничен. Для нужд архивирования применяют накопители на магнитных лентах (стримеры). Ленты представляют собой самый надёжный, долговременный и дешёвый способ хранения данных. Недостаток лент состоит в невозможности доступа к блокам данных в произвольном порядке; как правило, данные с лент перед использованием копируют на диски. В последние годы с ростом объёмов жёстких дисков ленты стали использоваться сравнительно редко, сейчас стримеры можно встретить только в организациях, имеющих дело с большими архивами данных. Отметим, что современная кассета для стримера может хранить несколько *терабайт* данных; сам стример, то есть устройство для работы с такими кассетами, стоит заметно дороже обычного жёсткого диска аналогичной ёмкости, но кассеты, естественно, стоят дешевле; при использовании сравнительно большого числа кассет удельная стоимость[15] хранения архивных данных может быть ниже в десятки раз в сравнении с использованием жёстких дисков.

1.3.5. Резюме

Итак, мы можем подвести некоторые итоги: компьютер основан на общей шине, к которой подсоединены оперативная память и центральный процессор; внешние устройства, включая жёсткие диски и дисководы, а также клавиатуру, монитор, звуковые устройства и вообще всё, что вы привыкли видеть в составе компьютера, но что не является ни процессором, ни памятью, тоже подключается к общей шине, только не напрямую, а через специальные схемы, называемые контроллерами. С памятью процессор может работать сам по себе, для работы со всеми остальными устройствами необходимы специальные программы, называемые драйверами. Для долговременного хранения информации

[15] То есть общая стоимость хранения, поделенная на объём хранимой информации.

используются дисковые запоминающие устройства, где информация обычно организуется в виде хорошо знакомых вам ***файлов***; в файлах могут храниться как данные, так и компьютерные программы, но чтобы запустить программу или обработать данные, и то, и другое должно быть сначала загружено в оперативную память.

Среди всех программ особое место занимает программа, называемая ***операционной системой***; она запускается первой и получает полный доступ ко всем возможностям аппаратуры компьютера, а все остальные программы запускаются уже под управлением (и под контролем) операционной системы, и непосредственного доступа к аппаратуре не имеют; для выполнения действий, не сводящихся к преобразованию информации в отведённой памяти, программы вынуждены обращаться к операционной системе.

Понимание этих основных принципов построения компьютерных систем жизненно необходимо программисту в его работе, а тем, кто решил изучать программирование — в процессе обучения; так что если что-то оказалось непонятным, попробуйте перечитать эту главу снова, если же и это не поможет, попросите кого-нибудь более опытного объяснить вам всё, что вы не поняли.

1.4. Как правильно использовать компьютер

1.4.1. Операционные системы и виды пользовательского интерфейса

Установивший Windows сначала месяц
радуется, потом всю жизнь мучается.
Установивший Unix сначала месяц
мучается, потом всю жизнь радуется.

Из подслушанного у студентов

В предыдущем параграфе мы упоминали особую программу, которая называется *операционной системой*. В её основные задачи входит, во-первых, управление запуском и завершением других программ; во-вторых, операционная система берёт на себя управление периферийными устройствами во всём их многообразии, а всем остальным программам предоставляет упрощённые возможности для доступа к периферии: так, пользовательская программа может обратиться к операционной системе с просьбой открыть какой-нибудь файл на чтение (указав при этом только имя файла), прочитать из него информацию, разместить эту информацию в указанной области оперативной памяти, после чего закрыть файл, то есть прекратить работу с ним. Программу при этом совершенно «не волнует», на диске какого типа располагается

файл: на жестком диске, встроенном в компьютер (которых, кстати, тоже очень много и очень разных), на оптическом CD или DVD, на старотипной дискете, на flash-брелке или вообще на диске другого компьютера, который подключён в качестве сетевого ресурса. Все заботы о том, какие конкретно технические операции следует выполнить, чтобы найти файл с нужным именем и извлечь из него информацию, операционная система берёт на себя.

Первые операционные системы появились ещё в 1960-х годах, и за прошедшие с тех пор полвека, разумеется, их было создано очень много; но, как ни странно, к нынешнему времени количество принципиально разных операционных систем изрядно сократилось. Всем известно слово «Windows»; так называет свои системы компания Microsoft. При этом фактически все существующие ныне операционные системы, не имеющие отношения к Microsoft (и, соответственно, слова «Windows» в названии), оказываются представителями семейства систем под общим названием Unix. Именно таковы свободно распространяемые системы Linux во всём многообразии его ***дистрибуций***, таких как Debian, Ubuntu, Fedora, Slackware, Gentoo и многие другие, и FreeBSD, вариаций которой гораздо меньше, но тоже есть. Кроме того, к семейству Unix относятся Android, который основан на ядре Linux, а также Mac OS X и iOS, которые, как и FreeBSD, ведут своё начало от версии Unix, созданной в университете Беркли (как мы помним, аббревиатура BSD означает *Berkeley Software Distribution*).

Роль, задачи и принципы работы операционных систем — это тема для долгого обсуждения, которому мы посвятим в этой книге отдельную часть — пятую; пока отметим, что в задачи операционных систем, вопреки распространённому заблуждению, никоим образом не входит организация взаимодействия с пользователем, т. е. человеком, который работает с компьютером. Дело тут в том, что операционные системы сами по себе достаточно сложны — по большому счёту, это едва ли не самые сложные программы из существующих в мире; поэтому их создатели обычно стараются всё, что может быть сделано вне операционной системы, именно вне её и делать. При этом средства общения компьютера с пользователем — так называемый ***пользовательский интерфейс*** — не обязан быть частью операционной системы, его может поддерживать обычная программа. Именно так делается во всех вариантах Unix; за рисование окошек, переключение между ними и всё такое прочее отвечают разнообразные *надстройки*, написанные в виде обычных программ, запускающихся под управлением операционной системы, но при этом не являющихся её частью. В операционных системах Microsoft, напротив, поддержка графического интерфейса включена в ядро системы, что приводит, в частности, к невозможности для пользователя самостоятельно выбрать интерфейс; работать приходится с тем единственным интерфейсом, который предоставлен системой.

Коль скоро речь пошла о графическом пользовательском интерфейсе, следует отметить, что в некоторых случаях он в системе вообще не нужен. Например, серверные компьютеры, обслуживающие запросы пользователей по компьютерной сети, чаще всего размещают в специальных стойках для аппаратуры, причём в зависимости от конкретной задачи в одну стойку может войти от десятка до двух-трёх сотен компьютеров, каждый со своим процессором, памятью и периферийными устройствами. Процессоры и блоки питания компьютеров нуждаются в охлаждении — как правило, воздушном, то есть с помощью обычных вентиляторов; эти вентиляторы при таком количестве компьютеров, размещённых в одном месте, могут производить изрядный шум. Поэтому для размещения серверов обычно предусматривают специальное помещение, в котором человеку находиться некомфортно из-за шума, из-за пониженной температуры воздуха, которая специально поддерживается кондиционерами для обеспечения более надёжной работы компьютеров, из-за сквозняков, создаваемых вентиляторами. Более того, физическое отсутствие людей в серверной улучшает условия для работы компьютеров — в помещение никто не приносит пыль и грязь, не выделяет в воздух лишнюю влагу, не спотыкается о провода. Поэтому люди в такое помещение заходят, только когда требуется что-то сделать с аппаратурой — отремонтировать её, установить новую, заменить старую; бывает, что ни один живой человек не появляется в серверной в течение нескольких месяцев. Все настройки и управление работой серверных машин производятся удалённо, из других помещений, где находятся рабочие места программистов и системных администраторов. Многие серверные компьютеры вообще не имеют в своём составе видеокарты, то есть монитор к ним подключить нельзя; до начала массового распространения USB к таким компьютерам невозможно было подключить также и клавиатуру. Зачем, спрашивается, на таких машинах поддерживать графический интерфейс, которого никто и никогда не увидит?

Большинство конечных пользователей[16] компьютеров в наши дни не понимает, как вообще можно использовать компьютер, если на нём нет графического интерфейса, но это, по большому счёту, лишь следствие проводимой некоторыми корпорациями пропаганды. Вплоть до середины 1990-х годов графические пользовательские интерфейсы не имели такого всеобъемлющего распространения, как сейчас, что совершенно

[16] Под ***конечным пользователем*** обычно подразумевают человека, который с помощью компьютеров решает какие-то свои задачи, никак не связанные с дальнейшим использованием компьютеров; например, секретарь или дизайнер, используя компьютер, являются при этом конечными пользователями, а программист — нет, поскольку задачи, которые он решает, нацелены на организацию работы других пользователей (или даже его самого) с компьютером. Вполне возможно, что конечными пользователями окажутся те, кто будет использовать программу, которую сейчас пишет программист.

не мешало людям пользоваться компьютерами, да и сейчас многие пользователи предпочитают копировать файлы и просматривать содержимое дисков с помощью двухпанельных файловых мониторов, таких как Far Manager или Total Commander, идеологическим предшественником которых был Norton Commander, работавший в текстовом режиме. Любопытно, что даже традиционный оконный интерфейс, в котором подразумевается возможность менять размеры окон, перемещать их по экрану, частично накладывать друг на друга и т. п., первоначально был реализован без всякой графики, на экране алфавитно-цифрового монитора.

Впрочем, и Norton Commander со всеми его поздними клонами, и оконные интерфейсы, использовавшие текстовый режим (а во времена MS-DOS они были весьма популярны), хотя и не используют графику как таковую, всё же основаны на том же базовом принципе, что и привычные ныне «иконочно-менюшечные» интерфейсы: пространство экрана они используют для размещения так называемых *элементов интерфейса*, или «виджетов», которые обычно включают меню, кнопки, флажки и переключатели (англ. *checkboxes* и *radiobuttons*), поля для ввода текстовой информации, а также статические поясняющие надписи; использование графического режима несколько расширяет репертуар виджетов, включая в него окна с пиктограммами («иконками»), всевозможные ползунки, индикаторы и другие элементы, на которые хватило фантазии у разработчика. Между тем, понаблюдав за работой профессионалов — программистов и системных администраторов, в особенности тех из них, кто использует системы семейства Unix, можно заметить ещё один подход к взаимодействию человека с компьютером: ***командную строку***. В этом режиме пользователь вводит с клавиатуры ***команды***, предписывающие выполнение тех или иных действий, а компьютер эти команды исполняет и выводит на экран результаты; когда-то давно именно это называлось ***диалоговым режимом*** работы с компьютером, в отличие от ***пакетного режима***, когда операторы заранее формировали пакеты заданий, полученных от программистов, а компьютер эти задания обрабатывал по мере готовности.

Первоначально диалоговый режим работы с компьютерами строился с помощью так называемых *телетайпов*[17], которые представляли собой электромеханическую пишущую машинку, подключаемую к линии связи. Исходное предназначение телетайпов состояло в передаче текстовых сообщений на расстоянии; ещё совсем недавно для срочных сообщений использовали *телеграммы*, которые почтальон доставлял на дом адресату, причём полученная телеграмма представляла собой полоски печатного текста, выданного телетайпом, вырезанные

[17]Интересно отметить, что в русском языке слово «телетайп» прочно закрепилось в качестве нарицательного обозначения соответствующих устройств, тогда как в англоязычных источниках чаще используется термин «телепринтер» (teleprinter); дело в том, что слово *teletype* являлось зарегистрированной торговой маркой одного из производителей такого оборудования.

Рис. 1.4. Телетайп ASR-33 с перфоратором и устройством чтения перфолент[18]

.

ножницами и наклеенные на плотное основание. Телеграммы были практически полностью вытеснены из обихода развитием цифровых сетей связи — мобильной телефонии и Интернета.

Подключить телетайп к компьютеру догадались ещё в эпоху первого поколения ЭВМ; свою роль сыграло то обстоятельство, что телетайпы к тому времени серийно производились для нужд телеграфии и были доступны на рынке, то есть их не надо было разрабатывать, а компьютерным инженерам того времени хватало других забот. Работая с компьютером в диалоговом режиме с помощью телетайпа, оператор набирал команду на клавиатуре, а ответ компьютера печатался на бумажной ленте. Интересно, что такой режим работы «продержался» на удивление долго: он был полностью вытеснен из практики лишь к концу 1970-х годов.

Использование телетайпа в качестве устройства доступа к компьютеру имело очевидный недостаток: расходовалось очень много бумаги. Первоначально именно это стало причиной массового перехода от традиционных телетайпов к алфавитно-цифровым ***терминалам***, которые были оснащены клавиатурой и устройством отображения (экраном) на основе электронно-лучевой трубки (кинескопа); всё, что оператор набирал на клавиатуре, передавалось в линию связи, а информация, полученная оттуда, отображалась на экране. Например, если два терминала соединить друг с другом, операторы смогут между собой «поговорить».

Экран вместо бумажной ленты существенно расширил имеющиеся возможности; в частности, для терминалов практически сразу были введены управляющие цепочки символов, известные как ***escape-последовательности*** (от названия спецсимвола Escape, имеющего код 27), при получении которых терминал перемещал курсор в указанную позицию на экране, менял цвет выводимого текста и т. п.

Сейчас алфавитно-цифровые терминалы больше не выпускаются; при необходимости с этой ролью может справиться любой ноутбук, оснащённый последовательным портом или переходником USB-serial, если на нём запустить

[18] Фото с сайта Википедии, см. http://en.wikipedia.org/wiki/File:ASR-33_at_CHM.agr.jpg

соответствующее программное обеспечение; кстати, первоначальная настройка упоминавшихся выше серверных машин, не имеющих видеокарты, производится именно так: системный администратор подключает свой рабочий компьютер через COM-порт к настраиваемой серверной машине и запускает у себя эмулятор терминала. Это позволяет произвести загрузку операционной системы с внешнего носителя, установить её на серверную машину, настроить связь с локальной сетью и средства удалённого доступа; дальнейшая настройка, а также управление в ходе эксплуатации обычно производятся удалённо по сети, поскольку это удобнее — настраиваемый компьютер уже не нужно связывать шнуром напрямую с машиной администратора.

Иногда можно услышать, что системы семейства Unix якобы *неудобны для пользователя, потому что там приходится работать с командной строкой*. Это, разумеется, миф, истоками которого служат перепутанные причины и следствия. Опровергнуть этот миф проще простого: вряд ли кто-нибудь, кто хотя бы раз видел ноутбук от Apple, заявит, что в MacOS X отсутствует графический интерфейс пользователя; наоборот, он там едва ли не самый «развесистый». Большинство «обычных» пользователей этим вполне удовлетворяется, но когда очередной макбук оказывается в руках профессионала, среди всего великолепия графического интерфейса *внезапно* обнаруживается эмулятор терминала с приглашением командной строки.

Практически то же самое происходит на современных дистрибуциях свободно распространяемых Unix-систем, ориентированных на конечного пользователя. Бросается в глаза разнообразие графических оболочек, используемых там. Как уже говорилось, графический интерфейс пользователя здесь не является частью операционной системы; больше того, в отличие от коммерческих систем, включающих ту же MacOS X, внешний вид пользовательского интерфейса в традиционных системах не зашит намертво в графическую надстройку, а реализуется отдельной программой, которая называется ***оконным менеджером***. Пользователь может выбрать тот внешний вид и функциональность оконной системы, которые ему удобнее, а при определённой сноровке — менять вид и поведение оконной системы, например, в зависимости от настроения, причём прямо во время работы, даже не закрывая окна запущенных приложений.

Разумеется, для Linux и FreeBSD давно существуют, кроме прочего, и «иконочные» файловые менеджеры, причём их написано довольно много — Nautilus, Dolphin, Konqueror, PCManFM, Thunar, Nemo, SpaceFM, ROX Desktop, Xfe и другие; ещё шире представлены *двухпанельные файловые менеджеры*, продолжающие традиции знаменитого Norton Commander: это текстовый Midnight Commander, а также графические gentoo (его не следует путать с одноимённым дистрибутивом Linux), Krusader, emelFM2, Sunflower, GNOME Commander, Double Commander, muCommander и т. п. Тем не менее многие профессионалы предпочитают работать с файлами — копировать их, переименовывать, сортировать

Рис. 1.5. Терминал vt100[20]

.
по отдельным каталогам[19], перебрасывать с диска на диск, удалять — с помощью команд командной строки. Это объясняется одним очень простым фактом: *так действительно удобнее и быстрее.*

Интересно, что средства командной строки присутствуют и в системах семейства Windows; получить окно терминала с соответствующим приглашением там можно, если нажать пресловутую кнопку «Пуск» (Start), выбрать в меню пункт «Выполнить» (Run) и в качестве имени команды ввести три буквы «`cmd`»; но стандартный интерпретатор командной строки под Windows очень примитивен, использовать его неудобно, а большинство пользователей даже не подозревают о его существовании. Профессионалам он тоже не подходит, так что в мире Windows даже они вынуждены обходиться графическими интерфейсами, задействуя командную строку только в редких случаях, как правило, связанных с обслуживанием системы. Программисты, привыкшие к Unix-системам и по тем или иным причинам вынужденные работать с Windows, часто устанавливают там перенесённые из-под Unix интерпретаторы командной строки; например, такой интерпретатор входит в пакет MinGW.

Конечно, командная строка требует некоторого *запоминания*, но команд, которые необходимо запомнить, не так много; между тем гра-

[20] Там же, см. http://en.wikipedia.org/wiki/File:DEC_VT100_terminal.jpg

[19] В наше время в ходу термин «папка»; этот термин, на самом деле означающий элемент графического интерфейса — то самое «окошко с иконками» — **неприемлем** для именования объекта файловой системы, содержащего имена файлов. В частности, *папки* совершенно не обязательно как-либо представлены на диске, а иконки в них не обязаны соответствовать файлам; в то же время с файлами и каталогами можно работать без «папок» — ни двухпанельные файловые менеджеры, ни командная строка никаких «папок» не подразумевают. В этой книге мы используем корректную терминологию; термины «каталог» и «директория» мы считаем равноправными, а слово «папка», кроме этой сноски, вам больше нигде в тексте не встретится.

фические интерфейсы, несмотря на все заявления об их «интуитивной понятности», *тоже вынуждают много чего запомнить*: чего стоит одно только использование клавиш Ctrl и Shift в сочетании с «мышкой» при выделении элементов (это ещё довольно просто, поскольку результат сразу же виден) и при копировании файлов, их перемещении и создании «ярлыков». Обучение работе с графическими интерфейсами «с нуля», то есть когда обучаемый вообще не имеет *никакого* опыта работы с компьютером, оказывается на поверку тяжелее, чем обучение работе со средствами командной строки; широкая публика потихоньку перестаёт замечать это просто потому, что сейчас к графическим интерфейсам люди начинают привыкать с дошкольного возраста в силу их широкого распространения — которое, в свою очередь, является скорее результатом усилий PR-подразделений определённых коммерческих корпораций, нежели следствием весьма сомнительного «удобства» графических интерфейсов. Зачастую пользователь привыкает не к графическому интерфейсу в принципе, а к конкретной версии такового и оказывается совершенно беспомощен, например, при переходе на другую версию операционной системы.

Конечно, прежде чем средства работы с командной строкой стали действительно удобными, им пришлось пройти долгий путь совершенствования. Современные интерпретаторы командной строки «помнят» несколько сотен последних введённых пользователем команд и позволяют быстро и без усилий найти нужную команду среди запомненных; кроме того, они позволяют редактировать вводимую команду с помощью клавиш «стрелок», «угадывать» имя файла по первым введённым буквам, некоторые варианты интерфейса командной строки выдают пользователю контекстные подсказки относительно того, что ещё можно написать в этой части вводимой команды, и т. п. Работа с такой командной строкой при условии хорошего уровня владения ею может происходить *в разы и даже в десятки раз быстрее*, чем выполнение тех же действий с помощью сколь угодно «навороченного» графического интерфейса. Представьте себе, к примеру, что вы вернулись из поездки в Париж и хотите скопировать на свой компьютер фотографии с карточки фотоаппарата. Команды

```
cd Photoalbum/2015
mkdir Paris
cd Paris
mount /mnt/flash
cp /mnt/flash/dcim/* .
umount /mnt/flash
```

с учётом автодополнения имён файлов и использования истории команд можно, не особенно торопясь, набрать за шесть-семь секунд, ведь бóльшую часть текста вообще не придётся набирать: скорее всего, у

вас в домашнем каталоге только имя подкаталога `Photoalbum` начинается с `Ph`, так что достаточно будет набрать только эти две буквы, нажать клавишу Tab, и командный интерпретатор допишет имя `Photoalbum`, услужливо поставив после него косую черту; то же самое можно сделать при наборе команды «`mount /mnt/flash`» (каталог `mnt` — скорее всего, единственный в корневом каталоге начинается с `m`, а его подкаталог `flash` — скорее всего, единственный, который начинается с `f`); вместо «`cp /mnt/flash/dcim/* .`» опытный пользователь наберёт «`cp !:1/dcim/* .`», а интерпретатор вместо «`!:1`» подставит первый аргумент предыдущей команды, то есть «`/mnt/flash`»; команду «`umount /mnt/flash`» набирать не нужно, достаточно будет набрать «`u!m`» (вместо `!m` будет подставлен текст последней команды, начинавшейся с `m`), или просто нажать два раза стрелку вверх и в появившейся на экране команде `mount /mnt/flash` добавить в начало букву `u`.

Если те же действия выполнять через интерфейс с пиктограммами, то вам понадобится сначала щелчками «мышки» добраться до содержимого карточки, затем, используя «мышку» в сочетании с клавишей Shift, пометить весь список файлов, правой кнопкой «мышки» вызвать контекстное меню, выбрать в нём действие «копировать», затем найти (всё теми же щелчками «мышки») каталог `Photoalbum/2014`, снова вызвать контекстное меню, создать подкаталог `Paris`, двойным щелчком зайти в него и, наконец, вызвав контекстное меню в третий раз, выбрать в нём пункт «вставить». Даже если всё делать быстро, у вас на эту процедуру уйдёт никак не меньше двадцати-тридцати секунд, а то и больше. Но это, как ни странно, не главное. Если вы, к примеру, очень часто копируете фотографии к себе на диск, то с использованием командной строки эту процедуру можно автоматизировать, написав так называемый ***скрипт*** — обычный текстовый файл, состоящий из команд. Для нашего примера скрипт может выглядеть так:

```
#!/bin/bash
cd Photoalbum/2015
mkdir $1
cd $1
mount /mnt/flash
cp /mnt/flash/dcim/* .
umount /mnt/flash
```

но опытный пользователь, скорее всего, напишет скрипт более гибко:

```
#!/bin/bash
DIR=Photoalbum/2015
[ "x$1" = x ] && { echo "No dir name"; exit 1 }
mkdir $DIR/$1
mount /mnt/flash
cp /mnt/flash/dcim/* $DIR/$1
```

```
umount /mnt/flash
```

Если теперь назвать любой из этих двух скриптов, например, `getphotos`, то в следующий раз, когда потребуется скопировать новые фотографии (например, по возвращении из Милана), достаточно будет дать команду

```
./getphotos Milan
```

С графическими интерфейсами такой фокус не проходит: в отличие от команд, движения и щелчки «мышки» не поддаются формальному описанию, во всяком случае, достаточно простому для практического использования.

Отметим, что запуск графических/оконных программ тоже гораздо правильнее осуществлять из командной строки, нежели использовать для этого всевозможные меню. Например, если вы знаете адрес сайта, на который хотите попасть, то для запуска браузера проще всего дать команду:

```
firefox http://www.stolyarov.info &
```

Само имя запускаемой программы (в данном случае `firefox`) не настолько длинное, чтобы с его набором могли быть какие-то проблемы, особенно с учётом автодополнения — например, автору на его компьютере оказалось достаточно набрать только буквы `fir` и нажать Tab; ну а адрес сайта вам всё равно пришлось бы набрать на клавиатуре, только, возможно, не в командной строке, а в соответствущем окошке браузера.

Примечательна выразительная мощь современных командных интерпретаторов: например, в них можно использовать текст, выведенный одной командой, в качестве части другой команды, не говоря уже о том, что результат работы одной программы можно направить на вход другой программы, и таким образом построить целую цепочку преобразований информации, называемую *конвейером*. Каждая программа, появившаяся в вашей системе, в том числе и написанная лично вами, может быть использована в бесконечном количестве сочетаний с другими программами и встроенными средствами самого интерпретатора командной строки; при этом вы можете воспользоваться программами других авторов для таких целей, о которых их авторы даже не подозревали. Вообще, **если возможности графического пользовательского интерфейса ограничены фантазией его разработчика, то возможности правильно организованной командной строки ограничены только возможностями компьютера**.

Эти возможности, во всяком случае, заведомо стоят того, чтобы их изучить. Конечно, преодолеть влияние пропаганды «софтверных» монстров и убедить всех пользователей компьютеров перейти на командную строку — задача в современных условиях нереальная; но,

коль скоро вы читаете эту книгу, по-видимому, вы не совсем обычный пользователь. Так вот, **для профессионала в сфере информационных технологий свободное владение средствами командной строки практически обязательно**; отсутствие этих навыков резко снижает вашу ценность как специалиста. Наконец, очень важно и то, что интерфейс командной строки оказывается чрезвычайно полезен в ходе начального обучения программированию, если угодно, в качестве учебного пособия. Причины этого подробно изложены в «методическом предисловии», но оно может оказаться непонятным для неспециалиста; в таком случае автору остаётся только просить читателя на некоторое время принять важность командной строки на веру; это ненадолго, вскоре вам всё станет понятно.

Вся наша книга написана в предположении, что на вашем компьютере установлена та или иная система семейства Unix и что вы используете для работы с компьютером интерфейс командной строки; тому, как это делается, посвящён остаток этой главы. Хотя мы уже говорили об этом в предисловии, но сочтём уместным повторить: если вы хотите чему-то научиться с помощью этой книги, интерфейс командной строки должен стать для вас основным способом повседневной работы с компьютером, причём это должно произойти как можно раньше.

1.4.2. Unix на домашней машине

Среди современных версий систем семейства Unix многие ориентированы на конечного пользователя, так что установить их можно без каких-либо проблем, следуя пошаговым инструкциям, которые в изобилии присутствуют в Интернете. Там же можно скачать и сами системы, и программы, которые в этих системах работают, практически для любых целей: одно из ключевых отличий мира Unix от мира «коммерческих» систем состоит в том, что никто не потребует от вас платить деньги за массово используемое программное обеспечение; деньги обычно приходится платить только в том случае, если программа изготовлена специально для вас по заказу (деньги при этом платятся за работу программистов, а не за программу как таковую), но такие вещи, как правило, касаются корпоративных пользователей, а не частных лиц.

Основных вариантов[21] выбора системы, собственно, всего два: это Linux и FreeBSD, причём из доступных вариантов FreeBSD конечному пользователю с некоторой натяжкой подходит разве что PC-BSD. С вариантами Linux проблема, можно сказать, противоположна: их так

[21] Эта информация была актуальна на момент написания книги в начале 2016 года. Вполне возможно, что к тому времени, когда вы будете читать этот текст, ситуация изменится. Так или иначе, в Интернете на форумах всегда найдутся люди, готовые помочь очередному неофиту.

много, что оказывается затруднительно сделать выбор. В любом случае, мы возьмём на себя смелость в качестве первой системы порекомендовать именно Linux; при желании вы можете позже, когда уже будете представлять, что делаете, попробовать поставить FreeBSD.

Сразу же может возникнуть вопрос, *куда* поставить Unix-систему. Конечно, если вы привыкли использовать другие системы, одномоментно отказаться от них будет сложно, но это и не требуется; переход на Unix можно сделать плавным и постепенным.

Самый простой способ решения проблемы — установить Unix на отдельный компьютер; этот способ подойдёт вам, если у вас найдётся «устаревший» компьютер, который вы перестали использовать после покупки более новой модели, но продать не успели или по каким-то причинам не захотели. Вас может удивить, насколько нетребовательны Unix-системы к характеристикам аппаратуры: компьютер, которому исполнилось уже лет десять и который вы считали ни на что не годным, под управлением того же Linux будет просто «летать». Профессионалы легко устанавливают Linux на машины класса Pentium-1 производства середины 1990-х годов, хотя так делать, пожалуй, мы вам не посоветуем: многие современные программы на такое не рассчитаны, так что приходится пользоваться их старыми, неподдерживаемыми версиями. Впрочем, вряд ли у вас найдётся исправный Pentium-1; ну а всё, что новее, вам вполне подойдёт. Между прочим, даже если нужного компьютера у вас в запасах не оказалось, вы можете попробовать приобрести такую машинку с рук на какой-нибудь барахолке в Интернете; как правило, денег за них хотят совсем немного, килограмм колбасы может стоить дороже.

Если отдельного компьютера нет, остаётся вариант с установкой Linux на один из дисков имеющейся машины. В принципе, Linux может быть установлен на любой логический диск, но займёт этот диск, безусловно, целиком, поскольку формат файловой системы у Linux совершенно не такой, как у Windows. Если жёсткий диск вашего компьютера разбит на логические разделы («`C:`», «`D:`» и так далее), вы можете выбрать любой из них, сделать копии всех нужных файлов, *удалить* этот логический диск (проще сделать это средствами Windows, хотя можно и в процессе установки Linux), а затем поставить Linux на освободившееся пространство. Если у вас есть программное обеспечение, способное изменять размеры существующих логических дисков, то проще будет сократить ваш логический диск в размерах, чтобы на жёстком диске образовалось незанятое пространство. Примерно 10-15 Gb будет более чем достаточно.

С самого начала лучше ориентироваться на «облегчённые» оконные менеджеры, такие как IceWM, Xfce, LXDE и другие. Автор этих строк на всех своих компьютерах использует древний и довольно аскетичный `fvwm2`, с которого начал при переходе на Linux в 1994 году; но

рекомендовать его читателю, пожалуй, всё же не станет. Чего следует по возможности избегать — так это использования «тяжёлых» окружений, прежде всего KDE и GNOME; в конце концов, операционная среда нужна, чтобы в ней запускать программы, а создатели «тяжёлых» оконников об этом, похоже, забывают: их изделия сами благополучно съедают львиную долю ресурсов системы.

Независимо от того, сами вы устанавливаете себе систему или вам в этом кто-то помогает, необходимо сразу же найти, как запускать эмулятор терминала; это может быть одна из программ `xterm`, `konsole`, `Terminal` и других. Обычно окно эмулятора терминала по умолчанию имеет такой размер, чтобы в нём помещалось 24 строки по 80 символов; изменять количество символов не следует, оно практически оптимально, но вот размер одного символа (и, как следствие, размер всего терминального окошка) практически всегда стоит подрегулировать так, чтобы окно, по-прежнему вмещая 24x80 символов, при этом накрыло бóльшую часть вашего экрана. Рекомендуется сразу же настроить чёрный цвет фона и серые (не белые, а именно серые) буквы; так глаза меньше устают.

Обязательно сделайте так, чтобы окно терминала можно было получить без усилий — нажатием комбинации клавиш или щелчком мыши по пиктограмме, расположенной где-то на видном месте; вызывать окно терминала через иерархические меню оказывается слишком долго. При этом обязательно настройте свою операционную среду так, чтобы терминальное окно сразу же открывалось с нужным размером шрифта. Если вы не знаете, как это сделать — обязательно разберитесь, поищите в Интернете соответствующие инструкции (они наверняка есть), найдите какой-нибудь форум, где вам помогут. Комфортное существование в операционной среде очень важно; ваш процесс самообучения может провалиться только из-за того, что вы поленитесь с самого начала правильно настроить среду.

Обязательно установите себе программы для повседневного использования — браузер (он, впрочем, скорее всего установится в процессе инсталляции системы), LibreOffice для работы с файлами «офисных» форматов, `evince` для чтения PDF-файлов, `eog` для просмотра изображений, `mplayer` для воспроизведения аудио и видео. Подумайте, что ещё вы обычно делаете на своём компьютере под управлением Windows и узнайте, как то же самое делается в Linux; можете не сомневаться, всё это возможно.

Наконец, в процессе изучения материала, приведённого в этой книге, вам также потребуются программистские редакторы текстов (попробуйте сразу же установить `vim`, `joe` и `nano`, чтобы они всегда были «под рукой»), компилятор Free Pascal (соответствующий пакет может называться fpc или fp-compiler; интегрированную среду устанавливать не надо), ассемблер NASM (пакет обычно так и называется «nasm»),

компилятор Си и Си++, который называется `gcc` (убедитесь, что при этом часть, имеющая отношение к Си++, тоже установилась, иногда они бывают в разных пакетах; впрочем, до Си++ дело дойдёт ещё не скоро), система сборки `make`, отладчик `gdb`. После установки всего этого вы готовы к дальнейшей работе.

С самого начала обязательно создайте для обычной работы непривилегированную пользовательскую учётную запись и всю работу проводите под ней. **Повседневная работа с правами пользователя root (то есть системного администратора) категорически недопустима**, причём некоторые программы даже откажутся запускаться. Входить в систему с правами администратора нужно только в случае необходимости (например, чтобы установить в системе дополнительные программы, скорректировать системные настройки и т. п.), причём лучше это делать с текстовой консоли, а не из графической оболочки; для переключения на текстовую консоль нажмите Ctrl-Alt-F1, для переключения обратно в оболочку X Window — Alt-F7, Alt-F8 или Alt-F9, в зависимости от конфигурации вашей системы. Отметим, что использовать команду `sudo` для повышения своих полномочий с пользовательских до администраторских тоже крайне нежелательно, несмотря на то, что в разнообразных руководствах (особенно ориентированных на дистрибутив Ubuntu) ровно так и предлагается делать. Опытные пользователи Unix вообще не допускают наличия этой команды в системе.

1.4.3. Первый сеанс в компьютерном классе

Если на домашнем компьютере проще всего установить для работы один из вариантов дистрибутива Linux, то в разнообразных компьютерных классах вам, возможно, придется столкнуться с той же ОС Linux, а возможно, что и с другой Unix-системой, такой как FreeBSD, причём в некоторых случаях нужная вам операционная система может функционировать либо непосредственно на той машине, с которой вы работаете, либо на общем сервере, к которому вам потребуется удалённый доступ. С точки зрения пользователя различия между этими вариантами не слишком велики.

Если речь идёт о компьютерном классе, то краткую инструкцию о том, как войти в систему, вы, скорее всего, получите от преподавателя или от системного администратора компьютерного класса вместе с вашим входным именем (login) и паролем (password). Итак, введите входное имя и пароль. Если вы сделали ошибку, система выдаст сообщение `Login incorrect`, которое может означать как опечатку во входном имени, так и неправильный пароль. Учтите, что регистр букв важен в обоих случаях, так что причиной неприятия системой пароля может быть, например, случайно нажатая клавиша CapsLock.

Для работы с системой вам необходимо получить приглашение командной строки. При использовании удаленного терминального доступа (например, с помощью программы `putty`) командная строка — это единственное средство работы с системой, которое вам доступно. Приглашение появится сразу после того, как вы введете верные имя и пароль. Если вы работаете в терминальном Unix-классе и вход в систему выполняете с помощью текстовой консоли, после ввода верных имени и пароля вы также немедленно получаете приглашение командной строки, однако в этом случае у вас есть возможность запустить один из возможных графических оконных интерфейсов. Это удобнее хотя бы тем, что можно открыть несколько окон одновременно. Для запуска графической оболочки X Window необходимо дать команду `startx`[22]; после этого нужно запустить один или несколько экземпляров программы `xterm`, выглядящей как графическое окно, в котором запускается интерпретатор команд.

Возможно также, что вход в систему выполняется сразу с помощью графического интерфейса; этот вариант бывает возможен как при работе с локальной машиной, так и при использовании удалённого доступа. В этом случае также необходимо получить доступ к интерпретатору командной строки, запустив программу `xterm` или какую-то другую программу эмуляции терминала.

Первым вашим действием в системе, если только это не ваш личный компьютер, должна стать смена пароля. В зависимости от конфигурации системы это может потребовать команды `passwd` или (в редких случаях) какой-то другой; об этом вам, скорее всего, скажет системный администратор. Введите эту команду (без параметров). Система спросит у вас сначала старый пароль, затем (дважды) новый. Учтите, что при вводе пароля на экране ничего не отображается. Придуманный вами пароль должен содержать не менее восьми символов, причём в нём должны присутствовать латинские буквы верхнего и нижнего регистров, цифры и знаки препинания. Пароль не должен основываться на слове естественного языка или на вашем входном имени. Вместе с тем следует придумать такой пароль, который вы легко запомните. Проще всего взять какую-либо запоминающуюся фразу, содержащую знаки препинания и числительные, и построить пароль на ее основе (числительные передаются цифрами, от остальных слов берутся первые буквы, причём буквы, соответствующие существительным, берутся заглавными, остальные — строчными). Например, из пословицы «Один с сошкой, семеро с ложкой» можно «сделать» пароль `1sS,7sL.`. И последнее: **не сообщайте свой пароль никому и никогда**, а также никому и никогда не позволяйте работать в системе под вашим именем. Фразы вроде «мне не жалко», «я доверяю своим друзьям» или «у меня

[22] В некоторых системах может потребоваться другая команда; за информацией обращайтесь к вашему системному администратору.

там всё равно нет ничего секретного» суть проявление дилетантства и легкомыслия в самом худшем смысле слова, и по мере набора опыта вы сами это поймёте.

1.4.4. Дерево каталогов. Работа с файлами

Система каталогов в ОС Unix существенно отличается от привычной пользователям MS-DOS и Windows, и наиболее заметные на первый взгляд отличия — это отсутствие букв, обозначающих устройства (что-то вроде `A:`, `C:` и т. п.), а также то обстоятельство, что имена каталогов разделяются в ОС Unix не обратной, а прямой косой чертой (`/`).

После входа в систему вы окажетесь в вашем *домашнем каталоге*. Домашний каталог — это место для хранения ваших личных файлов. Чтобы узнать имя (*путь*) текущего каталога, введите команду `pwd`[23]:

```
$ pwd
/home/lizzie
```

Узнать, какие файлы находятся в текущем каталоге, можно с помощью команды `ls`:

```
$ ls
Desktop    tmp
```

Имена файлов в ОС Unix могут содержать любое количество точек в любых позициях, т. е., например, `a.b..c...d....e` является вполне допустимым именем файла. При этом действует соглашение, что имена, начинающиеся с точки, соответствуют «невидимым» файлам. Чтобы увидеть все файлы, включая невидимые, можно воспользоваться командой `ls -a`:

```
$ ls -a
.  ..  .bash_history   Desktop    tmp
```

Некоторые из показанных имен могут соответствовать подкаталогам текущего каталога, другие могут иметь специальные значения. Чтобы было проще различать файлы по типам, можно воспользоваться флажком `-F`:

```
$ ls -aF
./  ../  .bash_history   Desktop/    tmp/
```

[23] Здесь и далее в примерах команд, когда необходимо показать и саму команду, и её вывод, мы будем перед командой добавлять символ приглашения командной строки «$». Настоящее приглашение командной строки обычно длиннее, оно может содержать имя пользователя, название машины, на который вы работаете, и текущий каталог, что очень удобно; в некоторых примерах мы для наглядности будем использовать более сложное приглашение, но пока для краткости ограничимся одним символом.

Таблица 1.1. Команды для работы с файлами

`cp`	копирование файла
`mv`	переименование или перемещение файла
`rm`	удаление файла
`mkdir`	создание директории
`rmdir`	удаление директории
`touch`	создание файла или установка нового времени модификации
`less`	просмотр содержимого файла с пейджингом

Теперь мы видим, что все имена, кроме `.bash_history`, соответствуют каталогам. Заметим, что «`.`» — это ссылка на сам текущий каталог, а «`..`» — ссылка на каталог, содержащий текущий каталог (в нашем примере это `/home/lizzie`).

Перейти в другой каталог можно командой `cd`:

```
$ pwd
/home/lizzie
$ cd tmp
$ pwd
/home/lizzie/tmp
$ cd ..
$ pwd
/home/lizzie
$ cd /usr/include
$ pwd
/usr/include
$ cd /
$ pwd
/
$ cd
$ pwd
/home/lizzie
```

Последний пример показывает, что команда `cd` без указания каталога делает текущим домашний каталог пользователя, как это было сразу после входа в систему.

Основные команды работы с файлами перечислены в табл. 1.1. Например, команда `cp file1.txt file2.txt` создаст копию файла `file1.txt` под именем `file2.txt`; команда `rm oldfile` удалит файл с именем `oldfile`. Большинство команд принимает дополнительные флажки-опции, начинающиеся со знака '`-`'. Так, команда `rm -r the_dir` позволяет удалить каталог `the_dir` вместе со всем его содержимым.

1.4.5. Шаблоны имён файлов

Во многих случаях бывает удобно произвести ту или иную операцию сразу над несколькими файлами. Для этого командный интерпретатор поддерживает ***подстановку имён файлов по заданному шаблону***. В качестве шаблона интерпретатор рассматривает любое слово, не заключённое в кавычки или апострофы, содержащее хотя бы один символ «`*`» или «`?`», квадратные или фигурные скобки. Знак вопроса в шаблоне считается соответствующим одному произвольному символу, а знак звёздочки — произвольной цепочке символов (в том числе, возможно, пустой); смысл скобок мы поясним чуть позже. Остальные символы в шаблоне обозначают сами себя. Встретив в командной строке такой шаблон, интерпретатор заменяет его списком из всех имён файлов, соответствующих шаблону, то есть в общем случае слово-шаблон может быть заменено на последовательность слов, имеющую произвольную длину: одно слово, десять, сто, тысяча, сколько угодно — в зависимости от того, сколько файлов соответствуют шаблону. Например, вместо шаблона, состоящего из одной звёздочки (или из произвольного количества звёздочек), интерпретатор подставит список всех файлов из текущей директории; вместо шаблона «`???*`» будут подставлены все имена файлов, состоящие не менее чем из трёх символов, а вместо «`???`» — имена файлов, состоящие **ровно** из трёх символов. Шаблон «`*.txt`» заменится на список всех имён файлов, имеющих суффикс[24] `.txt`, а шаблону `img_????.jpg` будут соответствовать имена вроде `img_2578.jpg`, `img_cool.jpg` и прочее в таком духе.

Применять шаблоны можно в любых командах, предполагающих списки имён файлов в качестве аргументов. Например, команда `rm *~` удалит в текущей директории все файлы, в имени которых последним символом стоит тильда, команда «`ls /etc/*`conf.» покажет список файлов с суффиксом `.conf`, находящихся в директории `/etc`, команда «`cp files/* /mnt/flash`» скопирует в директорию `/mnt/flash` все файлы из поддиректории `files`, находящейся в текущей директории, и так далее. В принципе команда, которую мы заставляем работать с шаблонами, не обязана предполагать именно файлы; так, команда «`echo *`» напечатает список файлов в текущей директории; между тем, команда `echo` сама по себе не имеет никакого отношения к файлам и не работает с файлами: она *печатает свои аргументы командной строки*.

Квадратные скобки в шаблоне позволяют обозначить *любой символ из заданного множества*; например, шаблон

[24] Читатель, привыкший к традиционной терминологии мира Windows, может удивиться использованию термина «суффикс» вместо «расширения». Дело здесь в том, что в MS-DOS и ранних версиях Windows «расширение» файла было намертво связано с его типом и рассматривалось обособленно от имени, тогда как в системах семейства Unix окончание имени файла никогда не играло такой роли и всегда было просто частью имени.

«`img_27[234][0123456789].jpg`» соответствует именам `img_2720.jpg`, `img_2721.jpg`, ..., `img_2734.jpg`, ..., `img_2749.jpg` и более никаким. Конечно, в реальной жизни можно воспользоваться тем обстоятельством, что, скорее всего, в директории нет ни одного файла, у которого в имени вместо четвёртой цифры стояло бы что-то другое, и воспользоваться более коротким шаблоном «`img_27[234]?.jpg`». Символ восклицательного знака позволяет, напротив, обозначить любой символ *кроме перечисленных*; например, «`[!_]*.c`» соответствует любым именам файлов, имеющим суффикс «`.c`», за исключением начинающихся с символа подчёркивания.

Фигурные скобки в шаблонах обозначают любую *цепочку* символов из явно перечисленных, сами цепочки при этом разделяются запятой. Например, шаблон «`*.{jpg,png,gif}`» соответствует всем файлам из текущей директории, имеющим суффиксы `.jpg`, `.png` или `.gif`.

Если шаблону не соответствует ни одно имя файла, интерпретатор оставляет шаблон без изменений, то есть передаёт слово вызываемой команде, как если бы это слово вообще не было шаблоном. Использовать эту возможность следует с осторожностью; большинство командных интерпретаторов, отличных от `bash`, такой особенности не имеют.

1.4.6. История команд и автодописывание имён файлов

Начинающие пользователи командной строки часто полагают, что каждую команду нужно написать вручную, буква за буквой; в командных интерпретаторах, использовавшихся лет сорок назад, всё так и было, но те времена давно прошли.

Для начала отметим, что современные командные интерпретаторы умеют *дописывать имена файлов*; эта возможность задействуется нажатием клавиши Tab. Начав писать имя файла, вы можете нажать Tab, и если на диске есть только один файл, имя которого начинается с букв, уже введённых вами, то его имя интерпретатор допишет за вас. Если же подходящих файлов больше одного, при нажатии на клавишу Tab ничего видимого не произойдёт, но вы можете тут же нажать на неё второй раз, и интерпретатор выдаст вам полный список файлов, подходящих к вашему случаю; бросив взгляд на этот список, вы в большинстве случаев поймёте, сколько букв вам ещё нужно набрать, прежде чем снова нажать Tab. Некоторые интерпретаторы умеют дописывать не только имена файлов, но и другие параметры в зависимости от команды, которую вы собрались дать. В целом эта возможность, называемая *автодополнением* (англ. *autocompletion*), способна сэкономить вам больше половины нажатий на клавиши.

Вторая удачная возможность интерпретатора, изрядно облегчающая жизнь пользователю, состоит в том, что интерпретатор *помнит историю введённых вами команд*, причём при завершении сеанса работы он эту историю сохраняет в специальном файле, так что вы можете воспользоваться своими командами и на следующий день, и через неделю. Если нужная команда была дана недавно, вы можете вернуть её на экран, последовательно нажимая клавишу «стрелка вверх»; случайно проскочив нужную команду при таком «движении вверх», можно вернуться обратно, нажав, что вполне естественно, «стрелку вниз». Любую команду из сохранённой истории вы можете отредактировать, используя привычные «стрелки» влево и вправо, клавиши Home, End и Backspace в их обычной роли.

Всю сохранённую историю можно просмотреть, используя команду `history`, в большинстве случаев её удобнее будет сочетать с пейджером `less`, то есть дать команду `history | less`. Здесь вы увидите, что каждая из запомненных интерпретатором команд снабжена номером; вы можете повторить любую из старых команд, зная её номер, используя восклицательный знак; например, `!137` выполнит команду, сохранённую в истории под номером `137`. Отметим, что «`!!`» обозначает *последнюю введённую команду*, а «`!:0`», «`!:1`» и т. д. — отдельные слова из неё; «`!abc`» обозначает последнюю команду, начинавшуюся со строки `abc`.

Наконец, в истории можно выполнить поиск по подстроке. Для этого нажмите `Ctrl-R` (от слова *reverse*) и начинайте вводить вашу подстроку. По мере того как вы будете набирать буквы, интерпретатор будет находить всё более и более старые команды, содержащие набранную подстроку. Если нажать Ctrl-R снова, вы получите *следующую* (то есть ещё более старую) команду, содержащую *ту же самую* подстроку.

Если при редактировании команды, поиске в истории и т. п. вы запутались, можно в любой момент сбросить вводимую строку, нажав `Ctrl-C`; это гораздо быстрее, чем, например, удалять все введённые символы, настойчиво нажимая на Backspace.

Начав активно использовать перечисленные здесь средства, вы вскоре убедитесь, что ваши трудозатраты на набор команд сократились по меньшей мере раз в двадцать. Ни в коем случае не пренебрегайте этими возможностями! На их освоение у вас уйдёт минут пять, а сэкономленное время будет исчисляться сотнями часов.

1.4.7. Управление выполнением задач

Далеко не все команды исполняются мгновенно, как это делали в наших примерах `pwd`, `cd` и `ls`; часто требуется сообщить запущенной программе, что пора заканчивать работу, а если она не понимает по-хорошему — прекратить её выполнение принудительно.

Первое, что нужно помнить при работе в командной строке Unix — это что многие программы подразумевают чтение данных с клавиатуры (говоря строже, «из потока стандартного ввода») до тех пор, пока там не возникнет ***ситуация конца файла***. С этим моментом у начинающих часто возникают определённые сложности: как может кончиться файл, более-менее понятно, но как то же самое может произойти с *клавиатурой*?! Но на самом деле ничего сложного тут нет. В ОС Unix используется обобщённый термин «поток данных», который может означать как чтение из файла, так и ввод данных с клавиатуры, или ещё откуда-нибудь; подробный разговор об этом у нас впереди. Одним из фундаментальных свойств потока данных является его способность *заканчиваться*.

Естественно, клавиатура сама по себе «кончиться» не может, но пользователь, вводящий данные, имеет полное право решить, что он уже ввёл всё, что хотел. Чтобы сообщить об этом активной программе (то есть той программе, которая в данный момент читает информацию, вводимую с клавиатуры), нужно нажать комбинацию клавиш `Ctrl-D`; при этом операционная система (если быть точным, драйвер терминала) устроит в соответствующем потоке ввода ситуацию «конец файла», и хотя наша клавиатура вроде бы никуда не делась, активная программа будет точно знать, что её входной поток информации иссяк. Между прочим, командный интерпретатор, который ведёт с нами диалог, тоже корректно обрабатывает ситуацию конца файла, так что если вы хотите завершить сеанс работы в одном из окон с командной строкой, наиболее корректный способ сделать это — нажать `Ctrl-D`; отметим заодно, что закрывать окно терминала средствами оконного менеджера (всякими дабл-кликами или через менюшки) — напротив, способ самый *некорректный*, поступать так с терминалами ни в коем случае не следует.

Умение имитировать на клавиатуре ситуацию «конец файла» ещё не раз потребуется нам в дальнейшем, так что **запомните: «конец файла» на клавиатуре имитируется нажатием `Ctrl-D`.**

Конечно, активная программа совершенно не обязана завершаться при обнаружении конца файла; больше того, она может вообще не читать ничего из своего стандартного потока ввода, так что о наступлении в нём ситуации конца файла просто не узнает. В конце концов, программа может просто «зависнуть» из-за какой-нибудь ошибки. Во всех подобных случаях нужно знать, как прекратить выполнение программы, не дожидаясь, когда она закончится сама.

Простейший и наиболее общепринятый способ принудительного завершения активной программы — это нажатие комбинации `Ctrl-C`; в большинстве случаев это поможет. К сожалению, бывает и так, что от `Ctrl-C` не наблюдается никакого эффекта; в этом случае можно попро-

бовать нажать `Ctrl-\`, в некоторых случаях активную программу это всё-таки остановит, но после этого (в зависимости от настроек системы) в текущем каталоге может появиться файл с именем `core` (в системе FreeBSD — файл с суффиксом `.core`), который стоит удалить сразу же, как только вы его увидели — он занимает довольно много места и при этом для вас пока совершенно бесполезен.

Чего точно **не следует** делать — это использовать комбинацию `Ctrl-Z` до тех пор, пока вы не поймёте, что она **в действительности** делает в ОС Unix. Этот вопрос мы подробно обсудим чуть позже.

Прежде чем переходить к использованию «тяжёлой артиллерии», следует ещё вспомнить, что терминалы обычно позволяют временно останавливать вывод (чтобы, например, успеть прочитать нужный фрагмент текста и не дать ему «прокрутиться» за пределы экрана). Такая «пауза» включается комбинацией `Ctrl-S`, а выключается — нажатием `Ctrl-Q`. Начинающие операторы Unix, привыкшие к комбинациях клавиш в графических интерфейсах, часто нажимают `Ctrl-S` случайно (пытаясь, например, сохранить файл в редакторе текстов и забыв, что здесь не Windows). Если вам кажется, что ваш терминал безнадёжно завис, на всякий случай попробуйте нажать `Ctrl-Q`: это поможет, если причиной «зависания» стала случайно нажатая «пауза», а в остальных случаях ничего плохого всё равно не случится.

Начинающие пользователи Unix часто делают ещё одну характерную ошибку — не зная, как справиться с программой, запущенной в окне терминала, они попросту закрывают само окно, например, дважды щёлкнув мышкой «где следует». Такая стратегия сродни засовыванию головы в песок: зависшей программы больше не видно, но это не значит, что она исчезла. Напротив, если не помогли штатные методы, то уж закрытие терминального окна не поможет тем более: запущенная задача продолжает работать, при этом она может впустую расходовать процессорное время, память, а в некоторых случаях и ещё что-нибудь натворить.

Если ни `Ctrl-C`, ни `Ctrl-\`, ни `Ctrl-Q` не помогли, то для принудительного завершения задачи придётся разобраться (хотя бы кратко) с понятием ***процесса*** и тем, как с процессами обращаться. В самом первом приближении «процесс» — это программа, которая запущена и в настоящий момент выполняется в системе; иначе говоря, когда вы запускаете любую программу, в системе появляется процесс, а когда программа заканчивает выполнение, соответствующий процесс исчезает (завершается). На самом деле всё несколько сложнее, например, запущенная вами программа может по своему усмотрению породить ещё несколько процессов и т. д.; всё это будет обсуждаться во втором томе нашей книги, в части, посвящённой операционным системам. Пока нас волнует вопрос сугубо прагматический: если мы запустили программу, в результате чего в системе возник процесс, то как этот процесс найти и уничтожить?

Отметим сразу же, что все процессы имеют в системе свои уникальные номера, благодаря которым их можно различать между собой. Список процессов, выполняющихся в настоящий момент, можно получить командой `ps`:

```
$ ps
  PID TTY          TIME CMD
 2199 pts/5    00:00:00 bash
 2241 pts/5    00:00:00 ps
```

Как видно, команда по умолчанию выдаёт только список процессов, запущенных в данном конкретном сеансе работы. К сожалению, флаги команды `ps` очень сильно отличаются в зависимости от версии (в частности, для FreeBSD и Linux). За подробной информацией следует обращаться к документации по данной конкретной ОС; здесь мы ограничимся замечанием, что команда «`ps ax`» выдаст список всех существующих процессов, а команда «`ps axu`» дополнительно покажет информацию о владельцах процессов[25].

В некоторых случаях может оказаться полезной программа `top`, работающая интерактивно. Она выдаёт на экран список наиболее активных процессов, обновляя его один раз в секунду. Чтобы выйти из программы `top`, нужно ввести букву `q`.

Снять процесс можно с помощью так называемого ***сигнала***; заметим, именно это происходит при нажатиии упоминавшихся выше комбинаций `Ctrl-C` и `Ctrl-\`, сигналы процессу при этом посылает драйвер терминала. Каждый сигнал имеет свой номер, название и некую предопределённую роль; больше про сигналы ничего толком сказать нельзя, понятие «отправки процессу сигнала» невозможно пояснить, не влезая в дебри, но нам это сейчас не требуется. Достаточно знать, во-первых, что процессу можно отправить сигнал с заданным номером (или именем); во-вторых, что процесс может сам решить, как реагировать на большинство сигналов, в том числе не реагировать на них вообще никак; и в-третьих, что существуют такие сигналы, над которыми процессы не властны; это позволяет убить процесс наверняка.

Комбинации `Ctrl-C` и `Ctrl-\` отправляют активному процессу соответственно сигналы `SIGINT` и `SIGQUIT` (для наглядности отметим, что они имеют номера 2 и 3, но помнить это не нужно). Обычно оба этих сигнала приводят к немедленному завершению процесса; если этого не произошло — скорее всего, ваш процесс их «перехватил» и для его снятия придётся применить неперехватываемый сигнал `SIGKILL` (№ 9). Отправить процессу произвольный сигнал позволяет команда `kill`, но прежде чем её применять, нужно узнать номер процесса, который мы хотим уничтожить. Для этого обычно открывают ещё одно окно терми-

[25] Это верно для ОС Linux и FreeBSD. В других ОС, например в SunOS/Solaris, ключи команды ps имеют совершенно иной смысл.

нала и в нём дают команду `ps ax`; в появившемся списке будут показаны как номера процессов, так и их командные строки, что обычно позволяет узнать номер нужного нам процесса. Например, если вы написали программу `prog`, запустили её, а она так качественно зависла, что не помогают никакие комбинации, то в выдаче команды `ps ax` ближе к концу вы, скорее всего, найдёте примерно такую строку:

```
2763 pts/6    R+     0:06 ./prog
```

Опознать нужную строку следует по имени программы (в данном случае `./prog`), а номер процесса посмотреть в начале строки (здесь это 2763). Зная этот номер, мы можем применить команду `kill`, но следует помнить, что по умолчанию она отправляет указанному процессу сигнал `SIGTERM`, (№ 15), который тоже может быть перехвачен процессом. Задать другой сигнал можно либо по номеру, либо по названию (`TERM`, `KILL`, `INT` и т. п.). Следующие две команды эквивалентны; обе передают процессу 2763 сигнал `SIGKILL`:

```
kill -9 2763
kill -KILL 2763
```

Очень редко процесс не исчезает даже после этого. Так может произойти только в двух случаях. Во-первых, это может быть так называемый процесс-зомби, который на самом деле уже завершился, но остаётся в системе, поскольку его непосредственный предок — тот, кто его запустил — почему-то не торопится затребовать у операционной системы информацию об обстоятельствах завершения своего потомка. Зомби убить нельзя — он уже и так мёртвый, помочь тут может разве что уничтожение его предка, тогда исчезнет и сам зомби. Впрочем, зомби не потребляет ресурсов системы, он только занимает место в таблице процессов, что неприятно, но не очень страшно.

Вторая ситуация гораздо хуже. Процесс мог выполнить системный вызов, то есть обратиться к операционной системе за какой-то услугой, в ходе выполнения которой система перевела его в состояние «непрерываемого сна», в котором он и остался. Обычно система переводит процессы в такое состояние на доли секунды; если процесс остался в таком виде надолго — в большинстве случаев это означает серьёзные проблемы с вашим компьютером, например, испорченный (физически!) диск. Здесь, к сожалению, не поможет вообще ничего; принудительно вывести процесс из такого состояния невозможно. Впрочем, если у вас начал «сыпаться» диск, скорее всего вам должно быть уже не до процессов.

Узнать, какая из двух ситуаций имеет место, можно из выдачи всё той же команды `ps ax`. Процесс-зомби отмечается буквой `Z` в столбце `STAT` и словом «defunct» в столбце командной строки, примерно так:

```
3159 pts/6    Z+     0:00 [prog] <defunct>
```

Процесс в состоянии «непрерываемого сна» можно отличить по букве `D` в поле `STAT`:

```
4711 pts/6    D      0:01 badblocks /dev/sdc1
```

Если какой-нибудь процесс находится в таком состоянии хотя бы несколько секунд — это повод проверить, всё ли хорошо с вашим компьютером.

1.4.8. Выполнение в фоновом режиме

Некоторые программы выполняются в течение длительного времени, при этом не требуя взаимодействия с пользователем через стандартные потоки ввода/вывода. Во время выполнения таких программ удобно иметь возможность продолжать давать команды командному интерпретатору, чтобы не тратить время.

Допустим, нам потребовалось обновить базу данных для команды `locate`, которая позволяет находить файлы в системе по части имени. В норме эта база данных обновляется автоматически, но обычно это происходит в ночное время, а если мы привыкли выключать наш компьютер на ночь, то данные, используемые программой `locate`, могут изрядно устареть. Обновление делается командой `updatedb`, выполнение которой может занять несколько минут. Ждать её окончания нам бы не хотелось, поскольку эти несколько минут мы могли бы, например, использовать для набора текста в редакторе. Чтобы запустить команду в фоновом режиме, к ней следует в конце приписать символ `&`, например:

```
$ updatedb &
[1] 2437
```

В ответ на нашу команду система сообщает, что задание запущено в фоновом режиме в качестве фоновой задачи №1, причём номер запущенного процесса — 2437. Текущий список выполняемых фоновых задач можно узнать командой `jobs`:

```
$ jobs
[1]+  Running   updatedb &
```

После завершения задачи командный интерпретатор нам об этом сообщит. В случае успешного завершения сообщение будет выглядеть так:

```
[1]+  Done      updatedb &
```

Если же программа при завершении сообщила операционной системе, что не считает своё выполнение успешным (с `updatedb` такое случается редко, с другими программами — гораздо чаще), сообщение будет иметь другой вид:

```
[1]+  Exit 1      updatedb &
```

Наконец, если фоновый процесс снять сигналом, сообщение будет примерно таким (для сигнала SIGTERM):

```
[1]+  Terminated  updatedb &
```

При отправлении сигналов процессам, являющимся фоновыми задачами данного конкретного экземпляра командного интерпретатора, можно ссылаться на номера процессов по номерам фоновых задач, добавляя к номеру символ «%». Так, команда «`kill %2`» отправит сигнал `SIGTERM` второй фоновой задаче. Символ «%» без указания номера обозначает последнюю из фоновых задач.

Если задача уже запущена не в фоновом режиме и нам не хочется ждать её завершения, мы можем сделать обычную задачу фоновой. Для этого следует нажать `Ctrl-Z`, в результате чего выполнение текущей задачи будет приостановлено. Затем с помощью команды `bg`[26] приостановленную задачу можно снова поставить на выполнение, но уже в фоновом режиме. Также возможно сделать текущей (т. е. такой, окончания которой ожидает командный интерпретатор) любую из фоновых и приостановленных задач. Это делается с помощью команды `fg`[27].

Помните: комбинация `Ctrl-Z` не убивает активную задачу, а лишь временно приостанавливает её выполнение. Это особенно важно для тех, кто привык к работе с «консольными» программами в Windows; там эта комбинация имеет совершенно иной смысл. Если вы привыкли к ней — значит, пора отвыкать.

Отметим, что возможности фонового исполнения особенно полезны при запуске оконных приложений — web-браузера, редактора текстов, работающего в отдельном окне (например, `geany`) или просто другого экземпляра `xterm`. Запустив на исполнение такую программу, мы, как правило, совершенно не хотим, чтобы наш командный интерпретатор ждал её завершения, не принимая от нас новых команд.

1.4.9. Перенаправление потоков ввода-вывода

В системах семейства Unix запущенные программы общаются с внешним миром через так называемые ***потоки ввода-вывода***; каждый такой поток позволяет получать извне (вводить) или, наоборот, передавать вовне (выводить) последовательность байтов, причём эти байты могут поступать с клавиатуры, из файла, из канала связи с другой программой, от аппаратного устройства или от партнёра по взаимодействию через компьютерную сеть; точно так же они могут выводиться на экран, в файл на диске, в канал связи, передаваться аппаратному устройству или уходить через компьютерную сеть на другую

[26] От английского background — «фон».
[27] От английского foreground — «передний план».

Таблица 1.2. Примеры перенаправлений ввода-вывода

`cmd1 > file1`	запустить программу `cmd1`, направив её вывод в файл `file1`; если файл существует, он будет перезаписан с нуля, если не существует — будет создан
`cmd1 >> file1`	запустить программу `cmd1`, дописав её вывод в конец файла `file1`; если файла не существует, он будет создан
`cmd2 < file2`	запустить программу `cmd2`, подав ей содержимое файла `file2` в качестве стандартного ввода; если файла не существует, произойдёт ошибка
`cmd3 > file1 < file2`	запустить программу `cmd3`, перенаправив как ввод, так и вывод
`cmd1 | cmd2`	запустить одновременно программы `cmd1` и `cmd2`, подав данные со стандартного вывода первой на стандартный ввод второй (так называемый ***конвейер***)
`cmd4 2> errfile`	направить поток сообщений об ошибках в файл `errfile`
`cmd5 2>&1 | cmd6`	объединить потоки стандартного вывода и вывода ошибок программы `cmd5` и направить на стандартный ввод программы `cmd6`

машину. Программа может одновременно работать с несколькими потоками ввода-вывода, различая их по номерам; эти номера называются ***дескрипторами***.

Практически все программы в ОС Unix следуют соглашению, по которому поток ввода-вывода с дескриптором 0 объявляется потоком стандартного ввода, поток с дескриптором 1 — потоком стандартного вывода и поток с дескриптором 2 — потоком для вывода сообщений об ошибках. Осуществляя обмен данными через стандартные потоки, большинство программ не делает предположений о том, с чем на самом деле связан тот или иной поток. Это позволяет использовать одни и те же программы как для работы с терминалом, так и для чтения из файла и/или записи в файл. Командные интерпретаторы, в том числе классический Bourne Shell, предоставляют возможности для управления вводом-выводом запускаемых программ. Для этого используются символы `<`, `>`, `>>`, `>&` и `|` (см. табл. 1.2).

Обычно в ОС Unix присутствует программа `less`, позволяющая постранично просматривать содержимое файлов, пользуясь клавишами «стрелка вверх», «стрелка вниз», PgUp, PgDn и др. для прокрутки. Эта же программа позволяет постранично просматривать текст, поданный ей на стандартный ввод. Использование программы `less` полезно в

случае, если информация, выдаваемая какой-либо из запускаемых вами программ, не умещается на экран. Например, команда

```
ls -lR | less
```

позволит вам просмотреть список всех файлов, находящихся в текущей директории и всех её поддиректориях.

Учтите, что многие программы выдают все сообщения об ошибках и предупреждения в стандартный поток ошибок. Чтобы просмотреть постранично сообщения такой программы (например, компилятора `gcc`), следует дать команду, объединяющую поток ошибок со стандартным потоком вывода и направляющую на вход программы `less` результат объединения:

```
gcc -Wall -g myprog.c -o myprog 2>&1 | less
```

Отметим ещё один момент. Если по какой-то причине вам не интересен поток информации, выдаваемый какой-нибудь программой, вы можете перенаправить его в псевдоустройство `/dev/null`: всё, что туда направлено, просто исчезает. Например, следующая команда сформирует список всех файлов в вашей системе, за исключением тех каталогов, на чтение которых у неё не хватит прав; при этом все сообщения об ошибках будут проигнорированы:

```
ls -l -R / > list.txt 2> /dev/null
```

1.4.10. Редакторы текстов

Различных редакторов текстов в операционных системах семейства Unix существует несколько сотен. Ниже приводятся основные сведения о трёх из них.

Выбирая для работы редактор текстов, следует обратить внимание на то, подходит ли он для написания программ. Для этого редактор текстов должен, во-первых, работать с файлами в обычном текстовом формате; во-вторых, редактор не должен выполнять автоматического форматирования абзацев текста (например, MSWord для этой цели непригоден); и, в-третьих, редактор обязан использовать моноширинный шрифт, то есть шрифт, в котором все символы имеют одинаковую ширину. Выяснить, удовлетворяет ли редактор этому свойству, проще всего, набрав в этом редакторе строку из десяти латинских букв `m` и под ней — строку из десяти латинских букв `i`. В редакторе, использующем моноширинный шрифт, полученный текст будет выглядеть так:

```
mmmmmmmmmm
iiiiiiiiii
```

Таблица 1.3. Команды редактора vim

`^`	перейти в начало строки
`$`	перейти в конец строки
`x`	удалить символ под курсором
`dw`	удалить слово (от курсора до пробела или конца строки)
`dd`	удалить текущую строку
`d$`	удалить символы от курсора до конца строки
`J`	слить следующую строку с текущей (удалить перевод строки)
`i`	начать ввод текста с позиции перед текущим символом (insert)
`a`	то же, но после текущего символа (append)
`o`	вставить пустую строку после текущей и начать ввод текста
`O`	то же, но строка вставляется перед текущей
`.`	повторить последнюю операцию
`u`	отменить последнюю операцию (undo)
`U`	отменить все изменения, внесенные в текущую строку

тогда как в редакторе, использующем пропорциональный шрифт (и непригодном, вследствие этого, для программирования), вид будет примерно таков:

mmmmmmmmmm
iiiiiiiiii

Редактор vim

Редактор vim (Vi Improved) является клоном классического редактора текстов для Unix-подобных операционных систем VI. Работа в редакторах этого семейства может показаться начинающему пользователю несколько неудобной, т. к. по построению интерфейса они коренным образом отличается от привычных большинству пользователей экранных редакторов текстов с системами меню. В то же время многие программисты, работающие под Unix-системами, предпочитают использовать именно эти редакторы, поскольку для человека, умеющего использовать основные функции этих редакторов, именно этот вариант интерфейса оказывается наиболее удобным для работы над текстом программы. Больше того, как показывает опыт автора этих строк, всё это относится не только к программам; текст книги, которую вы читаете, набран в редакторе vim, как и тексты всех остальных книг автора.

В любом случае, если освоение редактора vim покажется вам чрезмерно сложной задачей, к вашим услугам другие редакторы текстов, два из которых описаны ниже. Для читателей, решивших обойтись без изучения vim, приведем для справки последовательность нажатия

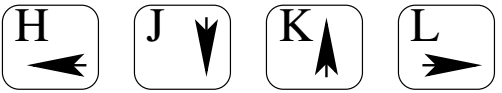

Рис. 1.6. Перемещение курсора в vim с помощью алфавитных клавиш

клавиш для выхода из этого редактора: если вы случайно запустили vim, практически в любой ситуации вы можете нажать `Escape`, затем набрать `:qa!`, и это приведёт к выходу из редактора без сохранения изменений.

Чтобы запустить редактор vim, достаточно дать команду `vim myfile.c`. Если файла `myfile.c` не существует, он будет создан при первом сохранении изменений. Первое, что необходимо уяснить, работая с vim — это **наличие у него двух режимов работы: режима ввода текста и режима команд**. Сразу после запуска вы оказываетесь в режиме команд. В этом режиме любые нажатия клавиш будут восприняты как команды редактору, т. е. если вы попытаетесь ввести текст, результат такой попытки вам не понравится.

Перемещение по тексту в режиме команд возможно с помощью стрелочных клавиш, однако более опытные пользователи vim предпочитают пользоваться для этой цели символами j, k, h и l для перемещения, соответственно, вниз, вверх, влево и вправо (см. рис. 1.6).

Причина такого выбора в том, что в ОС UNIX стрелочные клавиши генерируют последовательность байтов, начинающуюся с кода `Esc` (27); любая такая последовательность может быть воспринята редактором как требование на переход в командный режим и несколько команд-символов, причём единственный способ отличить Esc-последовательность, порождённую нажатием клавиши, от такой же последовательности, введённой пользователем — это измерение времени между приходом кода `Esc` и следующего за ним. При работе на медленной линии связи (например, при удалённом редактировании файла в условиях медленной или неустойчивой работы сети) этот способ может давать неприятные сбои.

Несколько наиболее часто употребляемых команд приведены в табл. 1.3. Команды `i`, `a`, `o`, и `O` переводят вас в режим ввода текста. Теперь всё вводимое с клавиатуры воспринимается как текст, подлежащий вставке. Естественно, возможно использование клавиши Backspace в её обычной роли. В большинстве случаев возможно также использование стрелочных клавиш, но в некоторых версиях vim, при некоторых особенностях настройки, а также при работе через медленный канал связи возможна неправильная реакция редактора на стрелки. В этом случае для навигации по тексту необходимо выйти из режима ввода. Выход из режима ввода и возврат в режим команд осуществляется нажатием клавиши Escape.

При необходимости найти в тексте то или иное ключевое слово следует использовать (в командном режиме) последовательность `/<word>`,

Таблица 1.4. Файловые команды редактора vim

`:w`	сохранить редактируемый файл
`:w <name>`	записать файл под новым именем
`:w!`	сохранить, игнорируя (по возможности) флаг readonly
`:wq`	сохранить файл и выйти
`:q`	выйти из редактора (если файл не был изменён с момента последнего сохранения)
`:q!`	выйти без сохранения, сбросив сделанные изменения
`:r <name>`	прочитать содержимое файла <name> и вставить его в редактируемый текст
`:e <name>`	начать редактирование еще одного файла
`:ls`	показать список редактируемых файлов (активных буферов)
`:b <N>`	перейти к буферу номер N

завершая её нажатием Enter. Так, `/myfun` установит курсор на ближайшее вхождение строки `myfun` в вашем тексте. Повторить поиск можно, введя символ `/` и сразу же нажав Enter.

Переместиться на строку с заданным номером (например, на строку, для которой компилятор выдал сообщение об ошибке) можно, набрав двоеточие, номер строки и нажав Enter. Также через двоеточие доступны команды сохранения, загрузки файлов, выхода и т. п. (см. табл. 1.4).

При одновременной работе с несколькими файлами переход между двумя последними редактируемыми файлами, кроме команды `:b`, также можно осуществить комбинацией клавиш `Ctrl-^`.

Отдельного упоминания заслуживают команды выделения блоков и работы с блоками. Начать выделение фрагмента, состоящего исключительно из целых строк, можно командой `V`; выделить фрагмент, состоящий из произвольного количества символов, можно с помощью команды `v`. Граница выделения устанавливается стрелками или соответствующими командами h, j, k и l.

Удалить выделенный блок можно командой d, скопировать — командой y. В обоих случаях выделение снимается, а фрагмент текста, находившийся под выделением, помещается в специальный буфер. Содержимое буфера можно вставить в текст командами `p` (после курсора) и `P` (перед курсором). Текст может попасть в буфер и без выделения. Так, все команды, удаляющие те или иные фрагменты текста (`x`, `dd`, `dw`, `d$` и др.), помещают удалённый текст в буфер. Команды `yy`, `yw`, `y$` помещают в буфер, соответственно, текущую строку, текущее слово и символы от курсора до конца строки.

Таблица 1.5. Наиболее употребительные команды редактора joe

`Ctrl-K D`	сохранить файл
`Ctrl-K X`	сохранить и выйти
`Ctrl-C`	выйти без сохранения
`Ctrl-Y`	удалить текущую строку
`Ctrl-K B`	отметить начало блока
`Ctrl-K K`	отметить конец блока
`Ctrl-K C`	скопировать выделенный блок в новое место
`Ctrl-K M`	переместить выделенный блок в новое место
`Ctrl-K Y`	удалить выделенный фрагмент
`Ctrl-K L`	найти строку по номеру
`Ctrl-Shift-'-'`	отменить последнее действие (undo)
`Ctrl-^`	снова выполнить отмененное действие (redo)
`Ctrl-K F`	поиск ключевого слова
`Ctrl-L`	повторный поиск

Редактор joe

Другой популярный в среде Unix редактор текстов называется joe от слов Jonathan's Own Editor. Чтобы запустить его, достаточно дать команду `joe myfile.c`. Если файла `myfile.c` не существует, он будет создан при первом сохранении изменений. В отличие от редактора vim, интерфейс редактора joe более похож на привычные для большинства пользователей редакторы текстов. Стрелочные клавиши, Enter, Backspace и другие работают в своей обычной роли, в большинстве случаев также доступна клавиша Delete. Команды редактору даются с помощью комбинаций клавиш, большинство из которых начинается с `Ctrl-K`. В частности, `Ctrl-K h` покажет в верхней части экрана памятку по наиболее употребительным командам редактора (см. табл. 1.5).

Встроенный редактор оболочки Midnight Commander

Оболочка (файловый монитор) Midnight Commander представляет собой клон некогда популярного файлового менеджера под MS-DOS, известного как Norton Commander. Запуск оболочки производится командой `mc`. Вызов встроенного редактора текстов для редактирования выбранного файла производится клавишей `F4`; если вы хотите создать новый файл, используйте комбинацию `Shift-F4`.

Интерфейс этого редактора достаточно понятен на интуитивном уровне, поэтому подробное описание мы опускаем. Ограничимся одной рекомендацией. Если не предпринять специальных мер, редактор будет вставлять в текст символ табуляции вместо групп из восьми пробелов, что может оказаться неудобным при использовании других редакторов. Единственный способ отключить такой стиль заполнения — установить опцию «Fill tabs with spaces». Чтобы добраться до диалога с настрой-

ками, нажмите F9, выберите пункт меню «Options», в нём — пункт «General». Чтобы настройки не потерялись при выходе из Midnight Commander, сохраните их. Для этого, выйдя из редактора, нажмите F9, выберите пункт меню «Options», а в нём — пункт «Save Setup».

1.4.11. Права доступа к файлам

С каждым файлом в ОС Unix связано 12-битное слово, называемое «правами доступа» к файлу[28]. Младшие 9 бит этого слова объединены в три группы по три бита; каждая группа задаёт права доступа для владельца файла, для его группы и для всех остальных пользователей. Три бита в каждой группе отвечают за право чтения файла, право записи в файл и право исполнения файла. Чтобы узнать права доступа к тому или иному файлу, можно воспользоваться командой `ls -l`, например:

```
$ ls -l /bin/cat
-rwxr-xr-x  1 root  root   14232  Feb 4 2013 /bin/cat
```

Расположенная в начале строки группа символов `-rwxr-xr-x` показывает тип файла (минус в самом начале означает, что мы имеем дело с обыкновенным файлом, буква `d` означала бы каталог и т. п.) и права доступа, соответственно, для владельца (в данном случае `rwx`, т. е. чтение, запись и исполнение), группы и всех остальных (в данном случае `r-x`, т. е. права на запись отсутствуют). Как мы видим, файл `/bin/cat` доступен любому пользователю на чтение и исполнение, но модифицировать его может только пользователь `root` (администратор системы).

Поскольку группа из трёх бит соответствует ровно одной цифре восьмеричной системы счисления[29], общепринятой является практика записи слова прав доступа к файлу в виде восьмеричного числа, обычно трёхзначного. При этом младший разряд (последняя цифра) соответствует правам для всех пользователей, средняя — правам для группы и старшая (обычно она идёт самой первой) цифра обозначает права для владельца. Права на чтение соответствуют 1, права на запись — 2, права на чтение — 4; соответствующие значения суммируются, т. е., например, права на чтение и запись обозначаются цифрой 6 $(4 + 2)$, а права на чтение и исполнение — цифрой 5 $(4 + 1)$. Права доступа к файлу `/bin/cat` из нашего примера можно закодировать восьмеричным числом 0755[30].

[28] В английском оригинале — permissions.

[29] Подробно о системах счисления будет рассказано в § 1.5.2; в принципе, для работы с правами доступа к файлам не обязательно понимать, как в действительности устроена восьмеричная система счисления, достаточно просто помнить, какой вид прав соответствует какой цифре (4, 2, 1) и что итоговое обозначение режима доступа составляет их сумму, которая, естественно, оказывается цифрой от 0 до 7.

[30] Обратите внимание, что число записано с нулём впереди; согласно правилам языка Си это означает, что число записано в восьмеричной системе, а поскольку

Для каталогов интерпретация битов прав доступа несколько отличается. Права на чтение каталога дают возможность просмотреть его содержимое. Права на запись позволяют модифицировать каталог, т. е. создавать и уничтожать в нём файлы (причём удалить можно и чужой файл, а также такой, на который прав доступа нет, т. к. достаточно иметь права доступа на запись в сам каталог). Что касается бита прав «на исполнение», для каталога этот бит означает возможность каким-либо образом использовать содержимое каталога, в том числе, например, открывать файлы, находящиеся в каталоге. Таким образом, если на каталог установлены права чтения, но нет прав исполнения, мы можем его просмотреть, но воспользоваться увиденным нам не удастся. Напротив, если есть права исполнения, но нет прав чтения, мы можем открыть файл из этого каталога только в том случае, если точно знаем имя файла. Узнать имя мы никак не можем, т. к. возможности просмотреть каталог у нас нет.

Оставшиеся три (старших) разряда сло́ва прав доступа называются `SetUid Bit` (04000), `SetGid Bit` (02000) и `Sticky Bit` (01000). Если для исполняемого файла установить `SetUid Bit`, этот файл будет при исполнении иметь права своего владельца (чаще всего — пользователя root) вне зависимости от того, кто из пользователей соответствующий файл запустил. `SetGid Bit` работает похожим образом, устанавливая исполнение от имени группы владельца файла вместо группы запустившего программу пользователя. Примером suid-программы является `passwd`. `Sticky Bit`, установленный на исполняемом файле, в старых версиях ОС Unix означал, что сегмент кода программы следует оставить в памяти даже после того, как программа будет завершена; это позволяло экономить время на загрузке в память программ, исполняемых чаще других. Современные системы сами решают, какие файлы следует оставить в памяти, и делают это в среднем эффективнее, чем мог бы сделать администратор; `sticky bit` на исполняемых файлах современными системами игнорируется.

Для каталогов `SetUid Bit` и `SetGid Bit` означают, что какой бы пользователь ни создал в этом каталоге файл, в качестве, соответственно, «владельца» и «группы владельца» для этого файла будут установлены те же владелец и группа, что и у самого каталога. `Sticky Bit` означает, что даже если пользователь имеет право на запись в данный каталог, удалить он сможет только свои (принадлежащие ему) файлы.

Для изменения прав доступа к файлам используется команда `chmod`[31]. Эта команда позволяет задать новые права доступа в виде восьмеричного числа, например

```
chmod 644 myfile.c
```

устанавливает для файла `myfile.c` права записи только для владельца, а права чтения — для всех.

профессиональные пользователи Unix очень любят этот язык, они обычно записывают восьмеричные и шестнадцатеричные числа, следуя соглашениям Си и никак это не оговаривая, то есть предполагая, что их и так поймут.

[31] Сокращение английских слов *change mode*, то есть «изменить режим».

Права доступа также можно задать в виде мнемонической строки вида `[ugoa][+-=][rwxsXtugo]` Буквы u, g, o и a в начале означают, соответственно, владельца (user), группу (group), всех остальных (others) и всех сразу (all). «+» означает добавление новых прав, «-» — снятие старых прав, «=» — установку указанных прав и снятие всех остальных. После знака буквы r, w, x означают, как можно догадаться, права на чтение, запись и исполнение, буква s — установку/снятие `Set`-битов (имеет смысл для владельца и группы), t обозначает `Sticky Bit`. Буква `X` (заглавная) означает установку/снятие бита исполнения только для каталогов, а также для тех файлов, на которые хотя бы у кого-нибудь есть права исполнения. Если команду `chmod` использовать с флажком `-R`, она проведёт смену прав доступа ко всем файлам во всех поддиректориях заданной директории. Например, команда `chmod a+x myscript` сделает файл `myscript` исполняемым; команда `chmod go-rwx *` снимет со всех файлов в текущем каталоге все права, кроме прав владельца. Очень полезной может оказаться команда

```
chmod -R u+rwX,go=rX ~
```

на случай, если вы случайно испортите права доступа в своей домашней директории; эта команда, скорее всего, приведёт всё в удовлетворительное состояние. Поясним, что эта команда устанавливает в вашей домашней директории и всех её поддиректориях для всех файлов права для владельца на чтение и запись; для директорий, а также файлов, для которых исполнение разрешено хоть кому-то, владельцу назначаются также права на исполнение. Для группы и остальных пользователей устанавливаются права на чтение, для исполняемых файлов и директорий — также права на исполнение, а все остальные права убираются.

1.4.12. Электронная документация

Дистрибутивы ОС Unix обычно содержат большое количество документации, доступ к которой можно получить непосредственно в процессе работы. Большая часть такой документации оформлена в виде файлов, отображаемых с помощью команд `man` (от слова *manual*) и `info`.

Команда `man`

Команда `man` служит для поиска и отображения установленной в системе справочной информации. Справочник охватывает команды ОС Unix, системные вызовы, библиотечные функции языка Си (и других установленных в системе языков), форматы файлов, некоторые общие понятия и т. д. К примеру, если вы хотите узнать все опции команды `ls`, следует дать команду «`man ls`», а если вы, допустим, забыли, в каком порядке идут аргументы системного вызова `waitpid()`, вам поможет

команда «`man waitpid`». Программа `man` найдёт соответствующий документ в системном справочнике и запустит программу его отображения. Появившийся на экране документ можно листать с помощью клавиш «стрелка вверх» и «стрелка вниз», можно использовать клавишу «пробел», чтобы пропустить сразу страницу текста. Выход из просмотра справочного документа осуществляется клавишей `q` (от слова quit).

Если нужный вам справочный документ имеет большой объём, а вам необходимо найти в нём определённое место, может оказаться удобным поиск подстроки. Это делается вводом символа `/`, после которого следует набрать строку для поиска и нажать Enter. Повторный поиск той же строки осуществляется вводом `/` и нажатием Enter (то есть саму строку можно опустить). Чтобы выполнить поиск в обратном направлении, можно воспользоваться символом `?` вместо `/`.

В некоторых случаях системный справочник может содержать более одного документа с данным именем. Так, в системе существует команда `write` и системный вызов `write`. Команда `write` вам вряд ли понадобится, так что, если вы набрали `man write`, скорее всего, вы имели в виду системный вызов; к сожалению, система этого не знает и выдаст вам совсем не тот документ, который вам нужен. Эту проблему можно решить указанием *номера секции* системного справочника. Так, в нашем примере команда

```
man 2 write
```

выдаст именно документ, посвящённый системному вызову `write`, поскольку секция № 2 содержит справочные документы по системным вызовам. Перечислим другие секции системного справочника:

- 1 — пользовательские команды ОС Unix (такие команды, как `ls`, `rm`, `mv` и т. п. описываются в этой секции);
- 2 — системные вызовы ядра ОС Unix;
- 3 — библиотечные функции языка Си (к этой секции можно обратиться, например, за информацией о функции `sprintf`);
- 4 — описания файлов устройств;
- 5 — описания форматов системных конфигурационных файлов;
- 6 — игровые программы;
- 7 — общие понятия (например, `man 7 ip` выдаст полезную информацию о программировании с использованием TCP/IP);
- 8 — команды системного администрирования ОС Unix (например, в этой секции вы найдёте описание команды `mount`, предназначенной для монтирования файловых систем).

Справочник может содержать и другие секции, причём не обязательно обозначающиеся цифрой; так, при установке в системе интерпретатора языка Tcl его справочные страницы обычно оформляются в отдельную секцию, которая может называться “n”, “3n” и т. п.

Команда `info`

Если команда `man` служит для отображения отдельных документов, то команда `info` позволяет осуществлять просмотр целых наборов страниц, связанных между собой гиперссылками; иначе говоря, документы в формате `info` представляют собой гипертекст, а сама команда — браузер (просмотрщик) гипертекста, работающий в текстовом режиме. Система `info` часто содержит более подробную информацию о том же предмете, чем `man`. Так, `man make` выдаст довольно скудную информацию об опциях командной строки команды `make`, в то время как `info make` позволит основательно изучить работу с программой `make`, научиться составлять её входные файлы и т. д.

При работе с программой `info` можно пролистывать текст с помощью клавиш стрелок и пробела. Клавиши `n` и `p` позволяют перейти, соответственно, к следующей и предыдущей странице гипертекста. Гиперссылки в тексте заключены между знаком `*` и `::`, например:

```
* Reading Makefiles::    How makefiles are parsed
```

Если позиционировать курсор на словах `Reading Makefiles` и нажать Enter, произойдёт переход по данной гиперссылке. Если необходимо перейти назад, можно воспользоваться клавишей `l` (от слова last). Выход из программы `info` производится клавишей `q` (от слова quit) или нажатием `Ctrl-C`.

1.4.13. Командные файлы в Bourne Shell

Интерпретатор Bourne Shell может не только работать в режиме диалога с пользователем, но и выполнять программы, которые называются командными файлами (скриптами). Файл с программой, предназначенной для исполнения интерпретатором Bourne Shell, должен начинаться со строки

```
#!/bin/sh
```

Переменные в языке Bourne Shell имеют имена, состоящие из латинских букв, цифр, знака подчёркивания и начинающиеся всегда с буквы. Значением переменной может быть любая строка символов. Чтобы присвоить переменной значение, нужно написать оператор присваивания, например:

```
I=10
MYFILE=/tmp/the_file_name
MYSTRING="Here are several words"
```

Обратите внимание, что в имени переменной, а также вокруг знака равенства (символа присваивания) не должно быть пробелов, в противном случае команда будет расценена не как присваивание, а как обычная команда, в которой знак присваивания — один из параметров. Для обращения к переменной используется знак `$`, например:

```
echo $I $MYFILE $MYSTRING
```

В результате выполнения этой команды будет напечатано:

```
10 /tmp/the_file_name Here are several words
```

При необходимости скомпоновать слитный текст из значений переменных можно имена переменных заключать в фигурные скобки; например, команда «`echo ${I}abc`» напечатает «`10abc`».

Для выполнения арифметических действий используется конструкция `$(( ))`. Например, команда «`I=$(( $I + 7 ))`» увеличит значение переменной `I` на семь.

С помощью встроенной в интерпретатор Bourne Shell команды `test` можно осуществлять проверку выполнения различных условий. Если заданное условие выполнено, команда завершится с нулевым (успешным) кодом возврата, в противном случае — с единичным (неуспешным). Синонимом команды test является символ открывающей квадратной скобки, причём сама команда в этом случае воспринимает символ закрывающей квадратной скобки в качестве своего параметра (как знак окончания выражения), что позволяет наглядно записывать проверяемое выражение, заключая его в квадратные скобки. Приведём несколько примеров.

```
[ -f "file.txt" ]
            # существует ли файл с именем file.txt
[ "$I" -lt 25 ]
            # значение переменной I меньше 25
[ "$A" = "abc" ]
            # значение переменной A является строкой abc
[ "$A" != "abc" ]
            # значение переменной A не является строкой abc
```

Это можно, например, использовать в операторе ветвления:

```
if [ -f "file.txt" ]; then
   cat "file.txt"
else
   echo "Файл file.txt не найден"
fi
```

Заметим, что то же самое можно было написать и иначе, без использования квадратных скобок, но это менее наглядно:

```
if test -f "file.txt" ; then
   cat "file.txt"
else
   echo "Файл file.txt не найден"
fi
```

В качестве команды, проверяющей условие, может фигурировать не только test, но и любая другая команда. Например:

```
if gcc -Wall -g myprog.c -o myprog; then
   echo "Compiled successfully"
else
   echo "Failed to compile"
fi
```

Кроме оператора ветвления, язык Bourne Shell поддерживает и более сложные конструкции, в том числе циклы. Например, следующий фрагмент напечатает все числа от 1 до 100:

```
I=0
while [ $I -le 101 ]; do
   echo $I
   I=$(( I + 1 ))
done
```

Использовать информацию об успешности выполнения команды можно также с помощью так называемых логических связок `&&` и `||`, означающих, соответственно, логические операции «и» и «или». При этом логическому значению «истина» соответствует успешное завершение команды, а значению «ложь» — неуспешное. Командная строка

```
cmd1 && cmd2
```

заставит интерпретатор выполнить сначала команду `cmd1`; а команда `cmd2` будет выполнена только в случае, если `cmd1` завершилась **успешно**. Наоборот, командная строка

```
cmd1 || cmd2
```

подразумевает запуск `cmd2` в случае **неуспешного** завершения `cmd1`.

Приоритет логических связок между собой — традиционный (то есть «и» приоритетнее, чем «или»). В то же время приоритет операций «конвейер» и перенаправлений ввода-вывода выше, чем приоритет логических связок; так, команда

```
cmd1 && cmd2 | cmd3
```

представляет собой связку между командой cmd1 и конвейером cmd2 | cmd3 как целым. Значение «истинности» конвейера определяется успешностью или неуспешностью выполнения последней из составляющих его команд. Очерёдность применения операций, как обычно, можно изменить использованием круглых скобок, например:

```
(cmd1 && cmd2) | cmd3
```

В этом примере стандартный вывод команд cmd1 и cmd2 (если, конечно, она вообще будет выполняться) будет направлен на стандартный ввод cmd3.

Для более подробной информации о программировании на языке Bourne Shell следует обратиться к специальной литературе [1].

1.4.14. Переменные окружения

Выполняющиеся в ОС Unix программы имеют возможность получать некоторую (обычно связанную с глобальными настройками) информацию из так называемых ***переменных окружения***. ***Окружение*** фактически представляет собой множество текстовых строк вида `VAR=VALUE`, где `VAR` — имя переменной, а `VALUE` — её значение.

На самом деле окружение своё у каждой запущенной программы, так называемого ***процесса***. Процесс имеет возможность изменить своё окружение: добавить новые переменные, удалить уже имеющиеся или изменить их значения. Когда один процесс запускает другой процесс, то порождённый процесс обычно наследует окружение процесса-родителя.

Одной из наиболее важных является переменная с именем `PATH`. Эта переменная содержит список каталогов, в которых следует искать исполняемый файл, если пользователь дал команду, не указав каталог. Стоит также упомянуть переменную `HOME`, содержащую путь к домашнему каталогу пользователя; переменную `LANG`, по которой многоязычные приложения определяют, на каком языке следует выдавать сообщения; переменную `EDITOR`, в которую можно занести имя предпочитаемого редактора текстов. Разумеется, список переменных окружения этим не исчерпывается. Весь набор имеющихся в вашем окружении переменных можно увидеть, дав команду `set` без параметров.

Интерпретатор командной строки предоставляет возможности по управлению переменными окружения. Во-первых, при старте интерпретатор копирует всё окружение в свои собственные переменные (заметим, что внутренние переменные интерпретатора организованы так же, как переменные окружения, а именно — в виде набора строк вида `VAR=VALUE`), так что к ним можно обратиться:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin
$ echo $HOME
/home/stud/s2003324
```

```
$ echo $LANG
ru_RU.KOI8-R
```

Кроме того, интерпретатор предоставляет возможность копировать значения переменных обратно в окружение с помощью команды `export`:

```
PATH=$PATH:/sbin:/usr/sbin
export PATH
```

или просто

```
export PATH=$PATH:/sbin:/usr/sbin
```

Отметим, что сами по себе присваивания внутренних переменных, подобные тем, что мы использовали в командных файлах в предыдущем параграфе, на окружение никак не влияют.

Переменную можно убрать из окружения с помощью команд `unset` и `export`:

```
unset MYVAR
export MYVAR
```

Модификация окружения влияет на выполнение всех команд, которые мы даем интерпретатору, поскольку запускаемые интерпретатором процессы наследуют уже модифицированный набор переменных окружения. Кроме того, при необходимости можно отдельно взятую команду запустить с модифицированным только для неё окружением. Это делается примерно так:

```
VAR=value command
```

Так, в ОС FreeBSD можно сменить информацию о пользователе, включая используемый командный интерпретатор, с помощью команды `chfn`, которая предлагает к редактированию определённый текст, из которого затем извлекает нужные значения. Эта команда по умолчанию запускает редактор текстов `vim`, что не для всех пользователей удобно. Выйти из положения можно, например, так:

```
EDITOR=joe chfn
```

В этом случае будет запущен редактор `joe`.

1.4.15. Протоколирование сеанса работы

При выполнении заданий практикума часто требуется представить протокол сеанса работы с программой, т. е. текст, включающий как информацию, вводимую пользователем, так и информацию, выдаваемую программой. Это легко сделать с помощью команды `script`. Чтобы начать протоколирование, запустите команду `script` с одним параметром, задающим имя файла протокола. Для окончания протоколирования нажмите `Ctrl-D` («конец файла»). Например:

```
$ script my_protocol.txt
Script started, file is my_protocol.txt
$ ls
a1.c   Documents   my_protocol.txt tmp
$ echo "abc"
abc
$ [Ctrl-D]
Script done, file is my_protocol.txt
```

Файл `my_protocol.txt` теперь содержит протокол сеанса работы:

```
Script started on Wed May 17 16:31:54 2015
$ ls
a1.c   Documents   my_protocol.txt tmp
$ echo "abc"
abc
$
Script done on Wed May 17 16:32:14 2015
```

1.4.16. Графическая подсистема в ОС Unix

Общие сведения

В отличие от некоторых других систем, ОС Unix сама по себе не включает никаких средств поддержки графического интерфейса. Для работы в графическом режиме в ОС Unix используется программный комплекс под общим названием X Window System, который состоит из обычных пользовательских программ и не является, вообще говоря, частью операционной системы.

Иногда можно встретить в литературе и разговорах наименование «XWindows». Такое наименование является категорически неправильным, что создатели системы X Window настойчиво подчеркивают. Слово «window» (окно) в наименовании этой системы должно стоять в единственном числе.

Центральным компонентом X Window System является программа, отвечающая за отображение графической информации на дисплее пользователя. Эта программа называется *X-сервером*. Все приложения, использующие графику, обращаются к X-серверу с запросами на вывод

того или иного изображения; таким образом, X-сервер предоставляет приложениям *услугу* (сервис) отображения графической информации, откуда и происходит название «X-сервер». Программы, обращающиеся к X-серверу (то есть все программы, работающие в ОС Unix и использующие графику), называются, соответственно, X-клиентами.

Отдельного упоминания заслуживают X-клиенты специального типа, называемые *оконными менеджерами*. Оконный менеджер отвечает за обрамление появляющихся на экране окон — отрисовывает рамки и заголовки окон, позволяет перемещать окна по экрану и менять их размеры. Авторам других графических программ, как следствие, не нужно думать о декоре окна; обычно X-приложение отвечает только за отрисовку прямоугольной области экрана, не имеющей ни рамки, ни заголовка, ни прочих стандартных элементов декора окна. С другой стороны, и пользователь в этой ситуации может выбрать из нескольких оконных менеджеров тот, который лучше отвечает его индивидуальным склонностям и потребностям.

Автор в свое время любил демонстрировать «непосвященным» простенький фокус, состоявший в замене «на лету» оконного менеджера с аскетично выглядящего `fvwm2` на `fvwm95`, в мельчайших подробностях копирующий внешний вид MS Windows-95. Особенно почему-то впечатляет зрителей тот факт, что открытые приложения при этом никуда не деваются.

Одной из самых популярных X-клиентских программ является `xterm` — эмулятор алфавитно-цифрового дисплея для X Window. В работе может быть удобно завести одновременно несколько экземпляров процесса `xterm`, каждый из которых порождает своё окно, в котором запускает копию интерпретатора командной строки. В одном окне мы можем запускать редактор vim, в другом —- выполнять трансляцию и отладку, в третьем — запускать тестовые программы и т. д.

Запуск X Window и выбор оконного менеджера

В зависимости от конфигурации конкретной машины система X Window может оказаться уже запущена, либо вам будет нужно запустить её самостоятельно. Обычно это делается с помощью команды `startx`, которая, в свою очередь, запускает программу `xinit`.

Возможно, в вашей локальной сети присутствует машина, выполняющая роль сервера приложений на основе `xdm`; к такой машине можно подключиться с помощью штатных средств X Window таким образом, что все ваши программы будут выполняться на этой (удалённой) машине, а на вашем локальном рабочем месте будет осуществляться только отображение их графических окон, т. е. ваш компьютер будет играть роль X-терминала. Чтобы проверить, есть ли в вашей сети xdm-сервера, попробуйте дать команду «`X -broadcast`» Если xdm-сервер в вашей сети действительно есть, после переключения в графический режим вы увидите приглашение к вводу имени и пароля для входа на этот сервер. Если xdm-серверов в сети больше одного, сначала вы увидите их список с предложением

выбрать, услугами какого из них вы желаете воспользоваться. Если в течение 15–20 секунд после перехода в графический режим ничего подобного не произошло — скорее всего, xdm-сервер в вашей сети отсутствует.

Команда `startx` вместе с X-сервером запустит для вас тот или иной оконный менеджер. В некоторых системах оконный менеджер можно выбрать из меню, появляющегося сразу после запуска X Window, в других системах выбор конкретного оконного менеджера определяется конфигурацией.

Если конфигурация системы вас по тем или иным причинам не устраивает, вы можете настроить её по своему вкусу путём создания в домашней директории файла с именем `.xinitrc` (либо его редактирования, если такой файл уже есть). Обратите внимание на точку перед именем файла, она важна. По сути `.xinitrc` представляет собой командный файл, из которого запускаются прикладные программы, включая и сам оконный менеджер. Программа `xinit` запускает X-сервер, а затем, соответствующим образом установив переменные окружения, начинает выполнение команд из `.xinitrc`. Завершение выполнения `.xinitrc` означает завершение сеанса работы с X Window, при этом процесс X-сервера завершается. Простейший пример `.xinitrc` может выглядеть примерно так:

```
xterm &
twm
```

В этом случае сначала будет запущен `xterm` (его мы запускаем на всякий случай, чтобы можно было работать, даже если оконный менеджер имеет неудобную конфигурацию), после чего — стандартный оконный менеджер `twm`. Обратите внимание, что `xterm` запускается в фоновом режиме (для этого в конце первой строки поставлен знак `&`). Это сделано, чтобы не ожидать его завершения для запуска `twm`.

Оконный менеджер `twm` достаточно примитивен. Хотя некоторые пользователи Unix используют именно его[32], вы, возможно, захотите попробовать другие оконные менеджеры. В вашей системе могут быть установлены, кроме `twm`, такие оконные менеджеры, как FVWM, IceWM, WindowMaker, Fluxbox и другие; кроме того, вы можете обнаружить и более развитые ОМ, такие как KDE и GNOME[33]. Чтобы воспользоваться IceWM, измените ваш `.xinitrc`:

```
xterm &
icewm
```

Если вы предпочитаете KDE, ваш файл может выглядеть так:

[32] Надо сказать, что `twm` в системе присутствует практически всегда, чего о других оконных менеджерах не скажешь.

[33] Всего существует несколько десятков оконных менеджеров, так что приведённый список не может претендовать на полноту.

```
xterm &
startkde
```

Выбор конкретного оконного менеджера среди множества доступных зависит в основном от ваших личных предпочтений. Так, стандартный ОМ `twm` выглядит достаточно аскетично, тогда как KDE по вычурности своего графического дизайна, пожалуй, превосходит системы семейства Windows. В то же время KDE весьма требователен к ресурсам и ощутимо «тормозит» даже на сравнительно быстрых компьютерах, тогда как аскетичные `twm`, `fvwm` и даже достаточно развитый в дизайнерском отношении `icewm` работают с приемлемой скоростью даже на компьютерах с процессорами Pentium-1 (KDE на таких машинах лучше и не пробовать запускать).

Отметим, что KDE предназначен в основном для пользователей, не умеющих и не желающих работать в командной строке. Профессиональные программисты и системные администраторы обычно предпочитают оконные менеджеры попроще.

Работа с классическими оконными менеджерами

К классическим ОМ мы относим `twm`, `fvwm` и некоторые другие. Следует отметить, что любой оконный менеджер имеет весьма развитые средства настройки, позволяющие существенно изменить его поведение, так что дать исчерпывающую инструкцию по работе с каким-либо из оконных менеджеров на уровне «нажмите такую-то клавишу для получения такого-то результата» было бы затруднительно. В этом параграфе мы ограничимся общими рекомендациями, которые позволят вам быстро освоить работу с вашим оконным менеджером в той конфигурации, которая установлена в вашей системе. При желании вы можете изменить любые настройки оконного менеджера; за инструкциями по этому поводу следует обратиться к технической документации.

Итак, первое, что можно посоветовать после запуска оконного менеджера — это попытаться понять, каким образом в нем высветить главное меню. Во всех классических ОМ главное меню выдаётся, если щёлкнуть левой кнопкой мыши *в любом свободном месте экрана* (то есть в таком месте, которое не закрыто никакими окнами). Отметим, что в классических ОМ не поддерживается ничего похожего на «пиктограммы на рабочем столе», так что запуск программ можно осуществить либо через меню, либо с помощью командной строки, доступ к которой можно получить, запустив программу `xterm` или аналогичную. Обычно `xterm` или какой-то его аналог имеется либо в самом главном меню, либо в его подменю, называемом «terminals», «shells» и т. п. Если у вас уже есть одно окошко с командной строкой, можно запустить новый экземпляр `xterm`, дав команду

```
xterm &
```

Обратите внимание на символ «&». Программу `xterm` мы запускаем в фоновом режиме, чтобы старый экземпляр `xterm` (с помощью которого мы даём команду) не ждал его завершения: в противном случае запуск нового `xterm` теряет смысл, ведь у нас во время его работы не будет возможности пользоваться старым экземпляром.

Программа `xterm` имеет развитую систему опций. Например,

```
xterm -bg black -fg gray &
```

запустит эмулятор терминала на черном фоне с серыми буквами (тот же набор цветов обычно используется в текстовой консоли).

Классическим в системе X Window считается поведение, при котором фокус ввода находится в том окне, в котором расположен курсор мыши[34]. Такое соглашение позволяет, например, работать с окном, которое частично скрыто другими окнами, не вытаскивая его на верхний уровень. Такое поведение (именуемое FocusFollowsMouse), однако, может оказаться непривычным; при необходимости его можно сменить на более привычное, называемое ClickToFocus, с помощью настроек вашего оконного менеджера. В любом случае следите внимательно за положением курсора мыши.

Показать полностью (поднять на верхний уровень) окно, которое частично скрыто, можно, щёлкнув мышью по его *заголовку* (а не по любому месту окна). Ваши настройки могут предусматривать и обратную операцию («утопить» окно, показав то, что под ним) — обычно это делается «щелчком» *правой* кнопки мыши по заголовку. Для перемещения окна по экрану также служит его заголовок: достаточно навести на заголовок курсор мыши, нажать (и не отпускать) левую кнопку, выбрать новое положение окна и отпустить кнопку. Если заголовка окна не видно (например, он скрыт под другими окнами), ту же операцию можно проделать с помощью вертикальных и горизонтальных частей рамки окна, кроме выделенных участков в уголках рамки; эти угловые участки служат для изменения размеров окна, то есть при протягивании их мышкой перемещается не всё окно, а только тот его уголок, который вы захватили.

Если вы потеряли нужное вам окно, обычно его можно легко отыскать, щёлкнув *правой* кнопкой мыши в свободном месте экрана. Обычно при этом выдаётся меню, состоящее из списка существующих окон.

В большинстве случаев оконные менеджеры поддерживают так называемый виртуальный рабочий стол (virtual desktop), состоящий из нескольких «виртуальных экранов», на каждом из которых можно разместить свои окна. Это бывает удобно, если вы работаете одновременно

[34] Это отличается от поведения в Windows, где для перемещения фокуса ввода необходимо выбрать окно, щёлкнув по нему мышкой, при этом окно обязательно показывается полностью.

с большим количеством окон. Карта виртуального рабочего стола, на которой изображены виртуальные экраны, обычно находится в правом нижнем углу экрана; чтобы переключиться на нужный вам виртуальный экран, достаточно щёлкнуть мышью в соответствующем месте карты.

Из окон, в которых отображается некий текст, обычно можно скопировать этот текст в другие окна. Для этого достаточно выделить текст мышью; в большинстве программ, работающих под управлением X Window, нет специальной операции «copy»: копируется тот текст, который выделен. Вставка выделенного текста осуществляется *третьей* (средней) кнопкой мыши. Скорее всего, ваша мышь оснащена «колесом» для скроллинга; обратите внимание на то, что на это колесо можно нажать сверху вниз, не прокручивая его, и тогда оно сработает как обычная (третья) кнопка, что вам, собственно, и требуется.

1.5. Теперь немного математики

В этой главе мы рассмотрим очень краткие сведения из области математики, без знания и понимания которых в ходе дальнейшего чтения этой книги (и изучения программирования) у вас совершенно однозначно возникнут проблемы. В основном эти сведения относятся к так называемой *дискретной математике*, которая совершенно игнорируется в школьной программе по математике, но в последние годы вошла в программу школьной информатики. К сожалению, то, как эти вещи обычно излагаются в школе, не оставляет нам иного выбора, кроме как рассказать их самим.

1.5.1. Элементы комбинаторики

Комбинаторикой называется раздел математики, охватывающий задачи вроде «сколькими способами можно...», «сколько существует различных вариантов...» и прочее в таком духе. В комбинаторике всегда рассматриваются *конечные множества*, с элементами которых всё время что-то происходит: их переставляют в разном порядке, часть из них отбрасывают, потом возвращают обратно, объединяют в разные группы, сортируют и снова перемешивают и вообще всячески над ними измываются. Мы начнём с одной из самых простых задач комбинаторики, которую, во избежание лишнего занудства, сформулируем языком учебника для младшего школьного возраста.

> *Вася и Петя решили поиграть в шпионов. Для этого Вася не поленился прикрутить к окошку своей комнаты три цветные лампочки, которые хорошо видно снаружи, если их зажечь, причём зажигать каждую из лампочек можно независимо*

от других. Вася прикрутил лампочки на совесть, так что менять их местами нельзя. Сколько разных сигналов может передать Вася Пете с помощью своих лампочек, если вариант «ни одна не горит» тоже рассматривать как сигнал? Вася точно не знает, когда Петя решит посмотреть на его окно, так что всяческие варианты с азбукой Морзе и другими подобными сигнальными системами не подходит: Васе нужно привести лампочки в положение, соответствующее передаваемому сигналу, и в таком виде оставить надолго, чтобы Петя точно успел заметить сигнал.

Многие читатели, возможно, дадут правильный ответ без лишних раздумий: восемь; однако интерес здесь представляет скорее не то, *как* вычислить ответ (возвести двойку в нужную степень), а то, *почему* ответ вычисляется именно так. Чтобы выяснить это, мы начнём с тривиального случая: когда лампочка всего одна. Очевидно, что передать здесь можно всего два разных сигнала: один из них будет обозначаться включённой лампочкой, а второй — выключенной.

Добавим теперь ещё одну лампочку. Если, например, эта вторая лампочка будет всё время включена, то можно будет, как и прежде, передать всего два сигнала: «первая лампочка включена» и «первая лампочка выключена». Но никто не мешает вторую лампочку выключить; в этом положении у нас тоже будет два разных сигнала: «первая включена» и «первая выключена». Однако тот, кому предназначены сигналы, в нашей задаче Петя, может смотреть на *обе* лампочки, то есть учитывать состояние их обеих. Первые два сигнала (при включённой второй лампочке) будут отличаться для него от вторых двух сигналов (при выключенной второй лампочке). Всего, следовательно, мы получим возможность передачи четырёх различных сигналов: *выключена-выключена, выключена-включена, включена-выключена* и *включена-включена.*

Снабдим эти четыре сигнала номерами от 1 до 4 и добавим ещё одну лампочку, третью. Если её включить, то можно будет передать четыре различных сигнала (с помощью включения и выключения первых двух лампочек). Если её выключить, мы получим ещё четыре сигнала, которые будут отличаться от первых четырёх; всего разных сигналов получится восемь. Останавливаться на этом нас никто не заставляет; пронумеровав имеющиеся восемь сигналов числами от 1 до 8 и добавив четвёртую лампочку, мы получим $8 + 8 = 16$ сигналов. Рассуждения можно обобщить: если с помощью n лампочек мы можем передать N сигналов, то добавление лампочки с номером $n+1$ удваивает количество возможных сигналов (то есть их получается $2N$), поскольку первые N у нас получаются с помощью исходно имевшихся лампочек при вы-

ключенной новой, а вторые N получается (всё с теми же имевшимися лампочками), если новую включить.

Полезно будет рассмотреть *вырожденный* случай: ни одной лампочки, то есть $n = 0$. Конечно, в шпионов так не поиграешь, но случай, тем не менее, с математической точки зрения важный. На вопрос «сколько сигналов можно передать, имея 0 лампочек», большинство обывателей ответит «нисколько», но, как ни странно, этот ответ неудачный. В самом деле, наши «сигналы» отличают одну ситуацию от другой, или, точнее, они *соответствуют* каким-то разным ситуациям. Если говорить ещё точнее, можно заметить, что «ситуаций» на самом деле бесконечно много, просто при передаче сигнала мы игнорируем некоторые факторы, тем самым объединяя множество ситуаций в одну. Например, наш юный шпион Вася мог обозначить сигналом «все лампочки погашены» ситуацию «меня нет дома»; остальные семь комбинаций в сигнализации наших друзей могли бы означать «я делаю уроки», «я читаю книжку», «я ем», «я смотрю телевизор», «я сплю», «я занят чем-то ещё» и, наконец, «я ничем не занят, мне вообще нечего делать, так что приходи в гости». Если внимательно посмотреть на этот список, можно заметить, что в любой из ситуаций возможны дальнейшие уточнения: сигнал «я ем» может одинаково обозначать ситуации «я обедаю», «я ужинаю», «я ем вкусный торт», «я пытаюсь одолеть невкусную и противную перловку» и т. п.; «я делаю уроки» может с равным успехом означать «я решаю задачи по математике», «я раскрашиваю карты, которые задали по географии» или «я сдуваю упражнения по русскому у соседки Кати». Возможны варианты «я делаю уроки и хорошо себя чувствую, так что скоро всё сделаю» и «я делаю уроки, но у меня при этом болит живот, так что домашняя работа сегодня затянется». Каждый из возможных сигналов несколько снижает общую неопределённость, но, разумеется, не устраняет её.

Вернёмся к нашему вырожденному примеру. Не имея ни одной лампочки, мы вообще не можем отличать ситуации друг от друга, но значит ли это, что у нас нет вообще никаких ситуаций? Очевидно, что это не так: наши юные шпионы по-прежнему чем-то заняты или, наоборот, не заняты, просто наш вырожденный вариант сигнализации не позволяет эти ситуации различать. Попросту говоря, мы *объединили все возможные ситуации в одну*, полностью убрав какую-либо определённость; но ведь мы объединили их в *одну* ситуацию, а не в *ноль* таковых.

Теперь уже всё становится на свои места: при нуле лампочек у нас один возможный сигнал, а добавление каждой новой лампочки увеличивает количество сигналов вдвое, так что $N = 2^n$, где n — количество лампочек, а N — количество возможных сигналов. Попутно отметим, что иногда приведённые рассуждения позволяют лучше понять, почему считается, что $k^0 \overset{\mathrm{Df}}{=\!=} 1$ для любого $k > 0$.

Задача про количество сигналов, передаваемых с помощью n лампочек, каждая из которых может гореть или не гореть, эквивалентна многим другим задачам; прежде чем мы перейдём к сухой математической сути, приведём ещё одну формулировку:

> *У Маши есть брошка, цепочка, серёжки, колечко и браслет. Каждый раз, выходя из дома, Маша долго раздумывает, какие из украшений в этот раз надеть, а какие оставить дома. Сколько у неё есть вариантов выбора?*

Чтобы понять, что это *та же самая задача*, давайте введём произвольное допущение, не относящееся к сути задачи и никак эту суть не затрагивающее: пусть наш юный шпион Вася из предыдущей задачи оказался младшим братом Маши и решил от нечего делать сообщать своему приятелю Пете, какие из украшений его сестра нацепила в этот раз. Для этого ему придётся добавить к трём уже имеющимся ещё две лампочки, чтобы их стало столько же, сколько у Маши украшений. Первая лампочка из пяти будет обозначать, надела ли Маша брошку, вторая — надела ли Маша цепочку, и так далее, по одной лампочке на каждое украшение из имеющихся у Маши. Мы уже знаем, что количество сигналов, передаваемых пятью лампочками, равно $2^5 = 32$; очевидно, что это количество в точности равно количеству комбинаций из украшений, которые надела Маша.

На «высушенном» математическом языке та же самая задача, полностью лишённая «шелухи», не влияющей на результат, и сведённая к чистым абстракциям, существенным с точки зрения решения, формулируется так:

> *Дано множество из n элементов. Сколько существует различных подмножеств этого множества?*

Ответ 2^n несложно запомнить, и, к сожалению, в школе обычно именно так и делают; результатом такого «дешёвого и сердитого» подхода к изучению математики становится лёгкость, с которой ученика можно совершенно запутать сколько-нибудь нестандартными формулировками условий задачи. Вот вам пример:

> *У Димы есть четыре разноцветных кубика. Ставя их один на другой, Дима строит «башни», причём один кубик, на котором ничего не стоит, Дима тоже считает «башней»; иначе говоря, башня у Димы имеет высоту от 1 до 4 кубиков. Сколько различных «башен» можно построить из имеющихся кубиков, по одной за раз?*

Кабы вы только знали, уважаемый читатель, сколько старшеклассников, совершенно не задумываясь, выдают на эту задачу ответ 2^4! Между

прочим, некоторые из них, заметив, что пустая башня условиями задачи исключена, «совершенствуют» свой ответ, вычтя «запрещённый» вариант, и получают $2^4 - 1$. Правильнее он от этого не становится ни на йоту, потому что это попросту *совершенно не та задача*, в которой двойку возводят в степень n; но чтобы это понять, необходимо для начала понимать, *почему* двойку возводят в степень n в «той» задаче, а вот с этим у школьников, зазубривших «магическое» 2^n, обнаруживаются фатальные проблемы.

Кстати, правильный ответ этой задачи — 64, но решение не имеет ничего общего с возведением двойки в шестую степень; если бы кубиков было три, ответ был бы 15, а для пяти кубиков правильный ответ будет 325. Всё дело тут, разумеется, в том, что в этой задаче важно не только *из каких кубиков состоит башня*, но и *в каком порядке* расположены кубики, составляющие башню. Поскольку для башен, состоящих более чем из одного кубика, можно получать разные варианты, просто меняя кубики местами, итоговых комбинаций оказывается гораздо больше, чем если бы мы рассматривали возможные наборы кубиков без учёта их порядка.

Прежде чем перейти к рассмотрению задач, в которых существенны перестановки, рассмотрим ещё пару задач на количество вариантов без перестановок элементов. Первую из них мы «состряпаем» из исходной задачи про юных шпионов:

> *Папа принёс Васе дешёвый китайский светильник, который может либо быть выключен, либо просто светиться, либо мигать. Вася немедленно прикрутил светильник к своему окну, где уже есть три обычные лампочки. Сколько разных сигналов Вася может передать Пете теперь, после усовершенствования своего шпионского оборудования?*

Задача, конечно, совершенно элементарная, но она представляет интерес вот в каком плане. Если человек понимает, *как* (и *почему именно так*) решается задача с тремя обыкновенными лампочками, то никаких проблем с «заковыристой» лампочкой, имеющей три разных состояния, у него не возникнет; но если задачу пытается решать среднестатистический школьник, которому на уроке вдолбили формулу $N = 2^n$, не объяснив, откуда она взялась, то с хорошей степенью достоверности на этой новой задаче он «сядет». А ведь решается она ничуть не сложнее, чем предыдущая, причём ровно теми же рассуждениями: мы можем передать восемь разных сигналов, если китайский светильник погашен; столько же мы можем передать, если он включён; и ещё столько же — если он мигает. Итого $8 + 8 + 8 = 3 \cdot 8 = 24$. Этот случай показывает, насколько *схема вывода формулы* ценнее самой формулы, и сейчас самое время отметить, что в комбинаторике *всегда так*; больше того, комбинаторные формулы попросту *вредно* помнить, их лучше каждый

раз выводить, благо все они настолько простые, что вывести их можно в уме. Запомнив любую формулу из области комбинаторики, вы рискуете применить её не по делу, как это делают вышеупомянутые школьники, пытаясь решать задачу про башни из кубиков.

Ещё одна задача на ту же тему выглядит так:

> *У Оли есть заготовки для флажков в форме обыкновенного прямоугольника, треугольника и прямоугольника с вырезом; ещё у Оли есть нашивки другого цвета в форме кружочка, квадратика, треугольничка и звёздочки. Оля решила сделать много флажков для праздника; сколько различных флажков она может сделать, пришивая к одной из имеющихся заготовок одну из имеющихся нашивок?*

Эта задача тоже, можно сказать, стандартно-хрестоматийная, так что большинство людей, хотя бы приблизительно знающих, о чём идёт речь, просто перемножат два числа и получат совершенно правильный ответ — 12. Но гораздо интереснее здесь не то, как решить конкретную задачу, а то, какими вообще путями можно это сделать. Для начала отметим, что наше рассуждение с лампочками замечательно проходит и здесь: в самом деле, если бы у Оли были только заготовки прямоугольной формы, то она смогла бы сделать столько разных флажков, сколько у неё есть разных нашивок, то есть четыре. Если бы у неё были только заготовки треугольной формы, она бы тоже смогла сделать четыре разных флажка, и то же самое — если бы все её заготовки имели форму прямоугольника с вырезом. Но ведь первые четыре варианта отличаются от вторых четырёх вариантов, а третьи четыре варианта отличаются и от первых, и от вторых — формой заготовки; следовательно, всего вариантов $3 \cdot 4 = 12$.

Однако больший интерес может представлять другой вариант рассуждений. Составим таблицу, в которой будет три столбца по числу разных заготовок и четыре строки по числу разных нашивок. В каждом столбце расположим флажки, сделанные с использованием соответствующей заготовки, а в каждой строке — флажки, сделанные с использованием соответствующей нашивки (см. рис. 1.7). Для каждого, у кого в мозгах сформирована абстракция *умножения*, немедленно становится очевидно, что клеток с флажками получилось $3 \cdot 4 = 12$; интересно, что это понимание сродни понятию *площади прямоугольника*, только для дискретного случая.

Наконец, рассмотрим ещё одну задачу, похожую на ту, решение которой мы оставили на потом:

> *У Димы есть ящик с кубиками четырёх разных цветов. Ставя кубики один на другой, Дима строит «башни», причём один кубик, на котором ничего не стоит, Дима тоже считает*

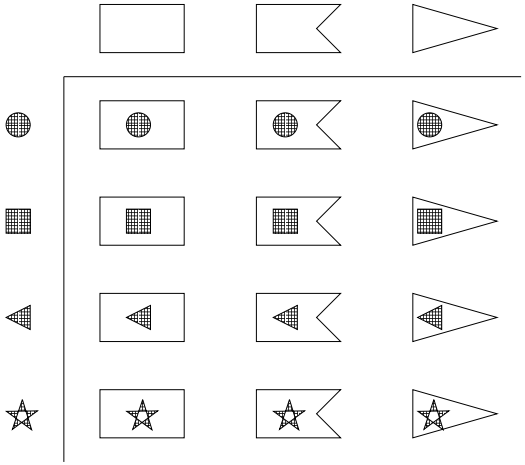

Рис. 1.7. Задача о флажках

> *«башней»; иначе говоря, башня у Димы имеет высоту от 1 до 4 кубиков. Сколько различных «башен» можно построить из имеющихся кубиков? Считается, что кубиков каждого цвета у Димы имеется сколько угодно.*

Несмотря на кажущуюся схожесть (здесь совпадают примерно три четверти текста), эта задача гораздо проще своей предыдущей версии, где кубиков было всего четыре. Впрочем, опять-таки, если не понимать, *как* получаются комбинаторные результаты, эту задачу решить невозможно, поскольку стандартные формулы для неё не срабатывают. Правильный ответ здесь — 340; предлагаем читателю самостоятельно продемонстрировать, как этот ответ получился.

Пока все задачи, которые мы рассматривали, решались без учёта перестановок; единственную задачу, в которой перестановки оказались существенны, мы решать не стали. Разговор о задачах с перестановками мы начнём, собственно говоря, с *канонической задачи о количестве возможных перестановок*. Как обычно, для начала сформулируем её по-школьному:

> *У Коли в мешке лежат семь шариков для американского бильярда с разными номерами (например, «сплошные» от единички до семёрки). Сколькими разными способами Коля может поставить их на полке в ряд?*

Прийти к правильному ответу можно двумя способами, и мы рассмотрим их оба. Начнём с тривиального варианта: есть всего один шарик, сколькими «способами» можно поставить его на полку? Очевидно, что

2	2 3 6	2 6 3
3	3 2 6	3 6 2
6	6 2 3	6 3 2

Рис. 1.8. Перестановки трёх шариков

способ тут только один. Возьмём теперь два шарика; в принципе, неважно, какие у них номера, результат от этого не меняется, но пусть для определённости это будут шарики с номерами 2 и 6. Очевидно, есть два способа расставить их на полке: «двойка слева, шестёрка справа» и «шестёрка слева, двойка справа». Первый способ назовём прямым, второй — обратным, поскольку номера шаров слева направо в этом случае не возрастают, а, наоборот, убывают.

Добавим теперь третий шарик (например, пусть это будет номер 3) и посмотрим, сколько стало способов. Крайний левый шарик мы можем выбрать тремя способами: поставить слева двойку, поставить слева тройку или поставить слева шестёрку. Какой бы шарик мы ни выбрали, оставшиеся два шарика на оставшиеся две позиции можно поставить двумя уже известными нам способами — прямым и обратным; иначе говоря, на каждый вариант выбора крайнего левого шарика имеются два варианта расстановки остальных, то есть всего вариантов будет шесть (рис. 1.8). Отметим, что обычно перестановки нумеруют именно в таком порядке: сначала их сортируют по возрастанию первого элемента (то есть сначала идут перестановки, в которых первый элемент имеет наименьший номер, а в конце — перестановки, где номер первого элемента наибольший), затем все перестановки, имеющие одинаковый первый элемент, сортируют по возрастанию второго, и так далее.

Если теперь добавить *четвёртый* шарик (пусть это будет шар с номером 5), мы получим *четыре* способа выбора крайнего левого из них, и при каждом таком способе остальные шарики можно будет расставить уже известными нам шестью способами; всего перестановок для четырёх шариков получится, соответственно, 24. Теперь мы, можно надеяться, готовы к обобщению: если для $k-1$ шаров существует M_{k-1} возможных перестановок, то, добавляя k-й шар, мы количество перестановок увеличим в k раз, то есть $M_k = k \cdot M_{k-1}$. В самом деле, при добавлении k-го шара всего шаров становится как раз k, то есть самый первый (например, крайний левый) шар мы можем выбрать k способами, а остальные, согласно сделанному предположению, можно расставить (при фиксированном крайнем левом) M_{k-1} способами. Поскольку начали мы с того, что для *одного* шара существует *одна* возможная перестановка, то есть $M_1 = 1$, общее число перестановок

для k шаров получится равным *произведению всех натуральных чисел от* 1 *до* k:

$$M_k = k \cdot M_{k-1} = k \cdot (k-1) \cdot M_{k-2} = \ldots = k \cdot (k-1) \cdot \ldots 2 \cdot 1$$

Как известно, это число называется ***факториалом*** k и обозначается «$k!$»; на самом деле определением факториала натурального числа k считается «число перестановок из k элементов», а то, что факториал равен произведению чисел от 1 до k — это следствие.

Для сформулированной выше задачи ответ составит, таким образом, $7! = 5040$ комбинаций.

К этому результату можно прийти и другим путём. Рассмотрим мешок, в котором лежат семь шаров, и семь пустых позиций на полке. Выбрать шар для заполнения первой пустой позиции мы можем семью способами; какой бы шар мы ни выбрали, в мешке их останется шесть. Иначе говоря, когда одна позиция уже заполнена, для заполнения второй позиции у нас имеется шесть вариантов, вне зависимости от того, каким из семи возможных способов была заполнена первая пустая позиция. Таким образом, на каждый из семи способов заполнения первой позиции у нас есть шесть способов заполнения второй позиции, а всего способов заполнения первых двух позиций получается $7 \cdot 6 = 42$. В мешке при этом остаётся пять шаров, то есть на каждую из 42 комбинаций первых двух шаров есть пять вариантов выбора третьего шара; всего вариантов для первых трёх шаров получается $42 \cdot 5 = 210$. Но на каждую такую комбинацию у нас есть четыре способа выбора очередного шара, ведь в мешке их осталось четыре; и так далее. Предпоследний шар из оставшихся в мешке мы сможем выбрать двумя способами, последний — одним. Получаем, что всего вариантов расстановки семи шаров у нас

$$7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 7! = 5040$$

Повторив это же рассуждение для случая k шаров, мы придём к уже знакомому нам выражению

$$k \cdot (k-1) \cdot (k-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 = k!$$

только в этот раз мы к нему пришли, двигаясь от бóльших чисел к меньшим, а не наоборот, как в предыдущем рассуждении. Отметим, что для понимания дальнейших выкладок нам будут полезны *оба* рассуждения.

Рассмотрим теперь промежуточную задачу, начав, как обычно, с частного случая:

> *У Коли в мешке по-прежнему лежат семь шариков для американского бильярда с номерами от единички до семёрки.*

Вася показал Коле небольшую полочку, на которой могут поместиться только три шара. Сколькими разными способами Коля может заполнить эту полочку шарами?

Несомненно, читатель без труда найдёт ответ на этот вопрос, повторив три первых шага из приведённого выше рассуждения: первый из трёх шаров мы можем выбрать семью способами, второй — шестью, третий — пятью; ответ составит $7 \cdot 6 \cdot 5 = 210$ вариантов. Это число можно записать, используя символ факториала:

$$7 \cdot 6 \cdot 5 = \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{4 \cdot 3 \cdot 2 \cdot 1} = \frac{7!}{4!}$$

В общем случае, когда у нас есть n предметов (элементов множества) и нужно составить упорядоченный набор (кортеж) длиной k, имеем:

$$n \cdot (n-1) \cdot \ldots \cdot (n-k+1) =$$
$$= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1) \cdot (n-k) \cdot \ldots \cdot 2 \cdot 1}{(n-k) \cdot \ldots \cdot 2 \cdot 1} =$$
$$= \frac{n!}{(n-k)!}$$

Эту величину, называемую *количеством размещений из n по k*, иногда в русскоязычной литературе обозначают A_n^k (читается *А из эн по ка*). В англоязычной литературе, а также и в некоторых русскоязычных источниках используется обозначение $(n)_k$, которое называют *убывающим факториалом*.

Сейчас, пожалуй, самое время решить задачу, которую мы сформулировали на стр. 109, но не стали решать. Напомним её условие:

У Димы есть четыре разноцветных кубика. Ставя их один на другой, Дима строит башни, причём один кубик, на котором ничего не стоит, Дима тоже считает башней; иначе говоря, башня у Димы имеет высоту от 1 до 4 кубиков. Сколько различных башен можно построить из имеющихся кубиков?

Ясно, что башен из четырёх кубиков будет $4! = 24$. Башен из трёх кубиков получится столько же: каждая из них получается из одной строго определённой башни высоты 4 снятием одного кубика, но то, что этот кубик Дима то ли держит в руках, то ли куда-то убрал, вместо того чтобы поставить его на вершину башни, никак не изменяет количества комбинаций. Наконец, башен из двух кубиков будет $4 \cdot 3 = 12$, а башен из одного кубика — 4, по числу имеющихся кубиков. $24 + 24 + 12 + 4 = 64$, это и есть ответ задачи.

Теперь мы подошли вплотную к очередной классической задаче, ради которой, в общем и целом, и был затеян весь разговор о перестановках. Как обычно, мы начнём с частного случая:

> *У Коли в мешке лежат всё те же семь шариков для американского бильярда с номерами от единички до семёрки. Вася дал Коле пустой мешок и попросил переложить в него три любых шара. Сколькими разными способами Коля может выполнить просьбу Васи?*

Эта задача отличается от предыдущей задачи про Колю и Васю тем, что шары в мешке, очевидно, перемешаются; иначе говоря, *нас больше не интересует порядок следования элементов в итоговых комбинациях*. Чтобы понять, как эта задача решается, представим себе, что Колю тоже заинтересовало, сколькими же способами он может выбрать три шара из имеющихся семи без учёта порядка, и он для начала выписал на листе бумаги все 210 вариантов, полученных при решении предыдущей задачи, где вместо мешка была полочка, то есть все возможные варианты размещений из семи по три *с учётом* порядка элементов. Зная, что варианты, различающиеся только порядком следования элементов, придётся теперь рассматривать как одинаковые, Коля решил для пробы посмотреть, сколько раз среди 210 выписанных комбинаций встречаются комбинации, состоящие из шаров с номерами 1, 2 и 3. Внимательно просмотрев свои записи, Коля обнаружил шесть таких комбинаций: 123, 132, 213, 231, 312 и 321. Решив проверить какой-нибудь другой набор шаров, Коля просмотрел свой список в поисках комбинаций, использующих шары с номерами 2, 3 и 6; таких комбинаций тоже нашлось шесть: 236, 263, 326, 362, 623 и 632 (эти комбинации уже знакомы нам по рис. 1.8 на стр. 113).

На этом месте своих изысканий Коля стал (будем надеяться, что вместе с нами) догадываться, что то же самое получится для *любого* набора шариков. В самом деле, ведь список из 210 комбинаций включает *все* возможные варианты выбора по три шара из семи с учётом их порядка; как следствие, какие бы мы ни взяли три шара из семи, в нашем списке найдутся, опять-таки, *все* комбинации, состоящие из этих трёх шаров, то есть, попросту, *все перестановки выбранных трёх шаров*; ну а перестановок из трёх элементов, как мы знаем, существует $3! = 3{\cdot}2{\cdot}1 = 6$. Получается, что *любая из интересующих нас комбинаций представлена в списке шесть раз*; иначе говоря, список ровно вшестеро длиннее, чем нужный нам результат. Нам осталось лишь поделить 210 на 6, и мы получим ответ задачи: $35 = \frac{210}{6} = \frac{(7)_4}{3!} = \frac{7!}{4!3!}$.

В общем случае нас интересует, сколькими способами можно выбрать k предметов из имеющихся n без учёта их порядка; соответствующая величина называется *количеством сочетаний из n по k* и обозначается как C_n^k (читается «Це из эн по ка»; буква C происходит от слова

combinations, то есть «комбинации», или «сочетания»). Повторяя вышеприведённые рассуждения для общего случая, заметим, что, если рассмотреть все $(n)_k$ *размещений* (которые отличаются от сочетаний тем, что в них порядок элементов считается важным), то в нём каждое *сочетание* будет представлено $k!$ раз по числу возможных перестановок k элементов, т. е. число $(n)_k = \frac{n!}{(n-k)!}$ превосходит искомое C_n^k ровно в $k!$ раз. Собственно говоря, всё, осталось только поделить $\frac{n!}{(n-k)!}$ на $k!$, и мы получим самую главную формулу школьной комбинаторики:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

А теперь мы скажем вам то, что вам вряд ли могли говорить в школе: **ни в коем случае, ни за какие пряники, ни при каких условиях не запоминайте эту формулу!** Если вы всё-таки случайно запомните её или если вы успели заучить эту формулу наизусть до того, как вам в руки попалась наша книга, постарайтесь её снова забыть, как самый страшный ночной кошмар. Дело в том, что помнить эту формулу просто *опасно*: возникает соблазн применить её без лишних раздумий в любой комбинаторной задаче, где присутствуют какие-нибудь два числа; в большинстве случаев такое применение будет ошибочным и даст неверные результаты.

Вместо запоминания самой формулы запомните лучше схему её вывода. Когда вам *действительно* будет нужно найти число сочетаний из n исходных элементов по k выбираемых элементов, вы сможете вывести формулу для C_n^k в уме за то время, пока будете её писать; для этого, написав «$C_n^k =$» и нарисовав дробную черту, прокрутите в уме примерно следующее: *общее число перестановок из n составляет $n!$, но нам нужны только k из них, так что лишние члены факториала мы убираем, поделив на $(n-k)!$; то, что получилось, есть количество комбинаций с учётом порядка, а нас порядок не волнует, так что получается, что мы каждую комбинацию посчитали $k!$ раз, делим на $k!$ и получаем то, что нам нужно.* Такой подход не позволит применить формулу не по делу, потому что вы будете знать точно, что́ эта формула означает и для чего её можно применять.

Интересно, что ту же самую формулу можно вывести и другим рассуждением. Представьте себе, что мы сначала выставили на полку в ряд n шаров, потом отделили первые k из них и ссыпали в мешок, а остальные $n-k$ ссыпали в другой мешок; естественно, в каждом мешке шары перемешались. Сколько возможно таких (итоговых) комбинаций, в которых шары *разложены по двум мешкам*, причём порядок ссыпания шаров в каждый из мешков нас не волнует? Пойдём по уже знакомой нам схеме рассуждений: изначально у нас было $n!$ комбинаций, но среди них *каждые* $k!$ стали неразличимы из-за того, что k шаров перемешались

в первом мешке, так что осталось $\frac{n!}{k!}$ комбинаций (после того, как первые k шаров ссыпали в первый мешок и там перемешали, а остальные $(n-k)$ шаров пока никуда не ссыпали), но среди этих комбинаций каждые $(n-k)!$ потом тоже стали неразличимы из-за перемешивания шаров во втором мешке. Итого $n!$ превосходит искомое C_n^k в $k!(n-k)!$ раз, то есть $C_n^k = \frac{n!}{k!(n-k)!}$.

Это рассуждение примечательно тем, что оно, в отличие от предыдущего, *симметрично*: оба множителя в знаменателе дроби получаются одинаковым путём. Задача и в самом деле обладает некоторой *симметрией*: *выбрать, какие k шаров пересыпать* в другой мешок, можно, очевидно, тем же числом способов, как и *выбрать, какие k шаров оставить* в мешке. Это выражается тождеством $C_n^k \equiv C_n^{n-k}$.

Для вырожденных случаев делают допущение, что $C_n^0 = C_n^n = 1$ для любого натурального n, но это допущение, как нетрудно убедиться, вполне естественно. В самом деле, C_n^0 соответствует ответу на вопрос «сколькими способами можно *ноль* шаров из n имеющихся пересыпать в другой мешок». Очевидно, такой способ *один*: мы просто ничего не делаем, и все шары остаются лежать в исходном мешке, а второй мешок так и остаётся пустым. Практически так же обстоят дела и с C_n^n: «сколькими способами можно n шаров из мешка с n шарами пересыпать в другой мешок»? Естественно, ровно одним: подставляем второй мешок, пересыпаем в него всё, что нашлось в первом, и дело сделано.

Числа C_n^k называют *биномиальными коэффициентами*, потому что через них можно записать общий вид разложения ***бинома Ньютона***:

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^{n-k} b^k$$

Например, $(a+b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$, при этом числа $1, 5, 10, 10, 5$ и 1 представляют собой $C_5^0, C_5^1, C_5^2, C_5^3, C_5^4$ и C_5^5. Интересно, что большинство профессиональных математиков полагают здесь всё настолько очевидным, что ни до каких объяснений не снисходят; между тем, остальная публика, в том числе и большинство людей, имеющих высшее техническое образование, но не являющихся профессиональными математиками, вообще не видят связи между задачей о пересыпании шаров из мешка в мешок и разложением бинома Ньютона; на вопрос, откуда в формуле бинома взялись комбинации шаров, они обычно отвечают сакраментальным «так получилось», видимо, полагая происходящее то ли случайным совпадением (хотя откуда в математике случайности?), то ли взаимосвязью настолько нетривиальной, что её вообще невозможно проследить.

Между тем, чтобы получить в процессе разложения бинома нашу «задачу о мешках с шарами», достаточно заметить, что речь должна идти не о том, *как разложить бином на слагаемые* (это вопрос слишком общий и затрагивает отнюдь не только комбинации), а о том, *какой коэффициент будет стоять при каждом члене разложения*.

$$
\begin{array}{ccccccccccccccccccccccc}
 & & & & & & C_0^0 & & & & & & & & & & & & & & & & \\
 & & & & & C_1^0 & & C_1^1 & & & & & & & & & & & & & & & \\
 & & & & C_2^0 & & C_2^1 & & C_2^2 & & & & & & & & & & & & & & \\
 & & & C_3^0 & & C_3^1 & & C_3^2 & & C_3^3 & & & & & & & & & & & & & \\
 & & C_4^0 & & C_4^1 & & C_4^2 & & C_4^3 & & C_4^4 & & & & & & & & & & & & \\
 & C_5^0 & & C_5^1 & & C_5^2 & & C_5^3 & & C_5^4 & & C_5^5 & & & & & & & & & & & \\
C_6^0 & & C_6^1 & & C_6^2 & & C_6^3 & & C_6^4 & & C_6^5 & & C_6^6 & & & & & & & & & & \\
 & & & & & & & & & & & & & & 1 & & & & & & & & \\
 & & & & & & & & & & & & & & 1 & & 1 & & & & & & \\
 & & & & & & & & & & & & & & 1 & & 2 & & 1 & & & & \\
 & & & & & & & & & & & & & & 1 & & 3 & & 3 & & 1 & & \\
 & & & & & & & & & & & & & & 1 & & 4 & & 6 & & 4 & & 1 \\
 & & & & & & & & & & & & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & & & & & & & & & & & & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}
$$

Рис. 1.9. Треугольник Паскаля

Как известно, если при раскрытии скобок не приводить однородные члены, то в итоговой сумме получится 2^n слагаемых. Например:

$$
(a+b)^4 = (a+b)(a+b)(a+b)(a+b) = (a+b)(a+b)(aa+ab+ba+bb) =
$$

$$
= (a+b)(aaa+aab+aba+abb+baa+bab+bba+bbb) =
$$

$$
= aaaa+aaab+aaba+aabb+abaa+abab+abba+abbb+
$$

$$
+baaa+baab+baba+babb+bbaa+bbab+bbba+bbbb
$$

Для большей наглядности мы здесь не стали использовать показатели степени. Каждое слагаемое итогового разложения представляет собой произведение, в котором *из каждой исходной «скобки» взято либо a, либо b*, а сама сумма состоит из *всех возможных таких слагаемых*. Нетрудно видеть, что их *возможно* ровно 2^n, ведь из каждой скобки мы должны взять либо a, либо b, то есть мы получаем уже знакомую нам задачу про юных шпионов и лампочки; но к делу это в данном случае не относится.

После приведения подобных членов мы, очевидно, получим сумму одночленов вида Ma^kb^{n-k} (напомним, что в нашем примере разложения $n=4$, но это лишь частная иллюстрация к общим рассуждениям), и нам осталось выяснить, чему равно M; нетрудно догадаться, что M представляет собой ответ на вопрос, *сколькими способами можно из всех n «скобок» выбрать k «скобок», из которых мы возьмём сомножителем слагаемое a, а из остальных возьмём слагаемое b*. В такой формулировке уже становится понятно, что это и есть, собственно говоря, наша задача про шары: вместо шаров у нас «скобки»; вместо перекладывания шара в другой мешок мы выбираем из «скобки» слагаемое b, вместо оставления шара в исходном мешке мы выбираем из «скобки» слагаемое a. В частности, для нашего примера одночлен a^2b^2 встречается в разложении шесть раз:

$$
aabb+abab+abba+baab+baba+bbaa
$$

что соответствует значению C_4^2; после приведения подобных членов в итоговом многочлене появится одночлен $6a^2b^2$. В то же время, например, одночлен b^4 встретился нам всего один раз (в виде $bbbb$), что соответствует $C_4^4=1$, и так далее.

Известен очень простой способ вычисления значений C_n^k, не требующий никаких операций кроме сложения — так называемый ***треугольник Паскаля***

(рис. 1.9). Первая строчка этого «треугольника» состоит из одной единицы, которая соответствует значению C_0^0 (сколькими способами можно из пустого мешка в другой пустой мешок пересыпать ноль шариков? Очевидно, одним). Последующие строки начинаются и заканчиваются единицей, что соответствует C_n^0 и C_n^n; все остальные элементы строк получаются сложением элементов предыдущей строки, между которыми стоит вычисляемый элемент. Например, двойка в середине третьей строки получена как сумма стоящих над ней единичек; две тройки в следующей строке получены сложением стоящих над каждой из них двойки и единицы; в нижней из показанных строк число 15 получено сложением стоящих над ним 5 и 10, и так далее. Треугольник Паскаля основан на тождестве $C_{n+1}^k \equiv C_n^{k-1} + C_n^k$, которое очень легко вывести сугубо аналитически:

$$C_n^{k-1} + C_n^k = \frac{n!}{(k-1)!(n-k+1)!} + \frac{n!}{k!(n-k)!} =$$

$$= n!\left(\frac{1}{(k-1)!(n-k+1)!} + \frac{1}{k!(n-k)!}\right) =$$

$$= n!\left(\frac{1}{(k-1)!(n-k+1)(n-k)!} + \frac{1}{k(k-1)!(n-k)!}\right) =$$

$$= n!\left(\frac{k}{k(k-1)!(n-k+1)(n-k)!} + \frac{(n-k+1)}{k(k-1)!(n-k+1)(n-k)!}\right) =$$

$$= n!\left(\frac{k}{k!(n+1-k)!} + \frac{(n+1-k)}{k!(n+1-k)!}\right) =$$

$$= n!\left(\frac{n+1}{k!(n+1-k)!}\right) = \frac{(n+1)!}{k!(n+1-k)!} = C_{n+1}^k$$

Однако гораздо интереснее в контексте нашего разговора «комбинаторный смысл» этого тождества, который оказывается неожиданно простым. Итак, пусть у нас был мешок с n шарами, пронумерованными от 1 до n. Нам дали ещё один шар, снабжённый номером $n+1$, и спросили, сколькими способами мы сможем теперь, когда у нас есть дополнительный шарик, насыпать k шаров в пустой мешок. Держа $(n+1)$-й шар в руках, мы сообразили, что все наши варианты можно разделить на две непересекающиеся группы. Первая группа вариантов основывается на том, что мы так и продолжаем держать в руках $(n+1)$-й шар, или вообще прячем его в карман, а для пересыпания в пустой мешок используем только шары, которые у нас были изначально. Таких вариантов, как несложно догадаться, C_n^k. Вторая группа вариантов предполагает, наоборот, что мы для начала кидаем в пустой мешок наш $(n+1)$-й шар, и нам остаётся досыпать туда $(k-1)$ шаров из нашего исходного мешка; это можно сделать C_n^{k-1} способами. Общее количество вариантов, таким образом, получается $C_n^k + C_n^{k-1}$, что и требовалось доказать.

Треугольник Паскаля обладает множеством интересных свойств, перечислять которые мы здесь не станем, поскольку и так уже несколько увлеклись. Отметим только одно из них: сумма чисел в любой строке треугольника Паскаля представляет собой 2^n, где n — номер строки, если их нумеровать с нуля (то есть n соответствует степени бинома, коэффициенты разложения которого

составляют данную строку). Иначе говоря, $\sum_{k=0}^{n} C_n^k \equiv 2^n$. Это свойство тоже имеет совершенно тривиальный комбинаторный смысл, который мы предлагаем читателю найти самостоятельно в качестве упражнения.

В заключение разговора о комбинаторике рассмотрим ещё одну хрестоматийную задачу:

> *В турнире по шахматам участвует семь шахматистов, причём предполагается, что каждый с каждым сыграет ровно одну партию. Сколько всего будет сыграно партий?*

Ясно, что каждый из семерых должен сыграть *шесть* партий, по одной с каждым из остальных участников турнира. Но вот следующая фраза почему-то многих начинающих комбинаторов вгоняет в ступор: *поскольку в каждой партии участвуют двое, всего партий будет вдвое меньше, чем* $7 \cdot 6$, т. е. всего будет сыграна $\frac{7 \cdot 6}{2} = 21$ партия.

Поскольку именно с этим вот «в каждой партии участвуют двое» имеются сложности, придётся дать некоторые пояснения, и мы дадим их двумя способами. Для начала давайте припомним, что шахматисты на соревнованиях обязательно записывают все ходы, причём это делают оба участника каждой игры; заполненные протоколы потом сдаются судьям. Представьте себе теперь, что *каждый из участников турнира* заготовил по одному бланку протокола для каждой предстоящей ему партии. Ясно, что таких бланков каждый заготовил шесть, а всего, стало быть, их было заготовлено $6 \cdot 7 = 42$. Теперь шахматисты встретились в зале в день турнира и начали играть; после каждой партии её участники сдают свои протоколы судьям, то есть *после каждой партии судьи получают два протокола.* В конце турнира, очевидно, все 42 протокола оказываются у судей, но ведь судьи получали по *два* протокола после каждой партии — следовательно, всего партий было вдвое меньше, то есть 21.

Есть и второй вариант объяснения. Результаты спортивных соревнований по так называемой «круговой системе», где каждый играет с каждым ровно одну партию, обычно представляют в виде *турнирной таблицы*, причём для каждого участника турнира предусматривается своя строка и свой столбец. Диагональные клетки таблицы, то есть такие, которые стоят на пересечении строки и столбца, соответствующих одному игроку, заштрихованы, поскольку сам с собой никто играть не собирается. Далее, если, например, игрок Б с игроком Д сыграли партию, и Б выиграл, то считается, что партия закончилась со счётом 1:0 в пользу Б; в его строку на пересечении со столбцом Д заносится результат «1:0», тогда как в строку Д на пересечении со столбцом Б заносится результат «0:1» (см. рис. 1.10).

Очевидно, что сначала клеток в таблице было $7 \cdot 7 = 49$, но семь из них сразу же закрасили, и их осталось 42; по итогам каждой пар-

	А	Б	В	Г	Д	Е	Ж
А	XXX						
Б		XXX			1:0		
В			XXX				
Г				XXX			
Д		0:1			XXX		
Е						XXX	
Ж							XXX

Рис. 1.10. Турнирная таблица

тии заполняются ещё *две* клетки, то есть после 21 партии все клетки окажутся заполнены, а турнир окончится.

Будучи переведённой на сугубо математический язык, эта задача превращается в *задачу о количестве рёбер в полном графе*. Напомним, что граф — это некое конечное множество абстрактных *вершин*, а также конечный набор *неупорядоченных пар вершин*, которые называются *рёбрами*; граф изображают в виде рисунка, на котором вершины обозначаются точками, а рёбра — линиями, соединяющими соответствующие вершины. Полным графом называется такой граф, в котором любые две вершины соединены ребром, притом только одним. *Полный граф, имеющий n вершин, имеет $\frac{n(n-1)}{2}$ рёбер*; в самом деле, в каждую вершину входит $(n-1)$ ребро, то есть всего в графе $n(n-1)$ «концов рёбер», но поскольку каждое ребро имеет два конца, их общее число составляет $\frac{n(n-1)}{2}$.

1.5.2. Позиционные системы счисления

Как известно, общепринятая и знакомая нам с дошкольного возраста система записи чисел ***арабскими цифрами*** представляет собой частный случай ***позиционной системы счисления***. Мы используем всего десять цифр; при этом, если число просматривать справа налево, то каждая следующая цифра имеет «вес» в десять раз больший, чем предыдущая, то есть действительное значение цифры зависит от её *позиции* в записи числа (именно поэтому система называется позиционной). Это никоим образом не совпадение: если мы хотим, чтобы каждое целое число могло быть записано в нашей системе, и притом только одним способом, то каждая следующая цифра должна «весить» ровно во столько же раз больше, сколько мы используем цифр. Число, обозначающее одновременно количество используемых цифр и то, во сколько раз каждая следующая цифра «тяжелее» предыдущей, называется ***основанием системы счисления***; для десятичной системы основанием служит, как несложно догадаться, число десять.

Здесь имеется довольно простая связь с комбинаторикой, которую мы рассматривали в предыдущем параграфе. В самом деле, пусть у нас имеется k пронумерованных флагштоков и неограниченное количество флажков $n-1$ разных расцветок, то есть на каждом из k флагштоков мы можем поднять любой из $n-1$ флажков, либо оставить флагшток пустым, и всё это независимо друг от друга; иначе говоря, каждый из k флагштоков независимо находится в одном из n состояний. Тогда общее число комбинаций флажков составит n^k. Если по какой-то причине нам этого количества комбинаций не хватит, придётся добавить ещё один флагшток; причём можно, в принципе, считать, что их изначально есть «сколько угодно», просто все, кроме k первых, пустые.

Ровно это и происходит при записи чисел с помощью позиционной системы счисления. Мы используем n цифр, причём цифра 0 соответствует «пустому флагштоку»; при работе с k разрядами (позициями) это даёт нам n^k чисел, от 0 до n^k-1. Например, существует $1000=10^3$ трёхзначных десятичных чисел — от 0 до 999. При этом мы изначально можем предполагать, что разрядов вообще-то бесконечное количество, просто все они, кроме первых (младших) k, содержат нули.

При прибавлении единицы к числу n^k-1 (в нашем примере — к числу 999) мы исчерпываем возможности k разрядов и вынуждены задействовать ещё один разряд, $(k+1)$-й. *Раньше* этого момента задействовать новый разряд не имеет смысла, ведь мы можем представить все меньшие числа, используя всего k разрядов, и если при этом «залезть» в следующий разряд, мы получим больше одного представления для одного и того же числа. С другой стороны, если возможности комбинаций младших разрядов уже исчерпаны, нам не остаётся иных возможностей, кроме задействования следующего разряда. Логично в этом следующем разряде начать с наименьшей возможной цифры, то есть с единицы, а все младшие разряды обнулить, чтобы «начать сначала»; таким образом, единица $k+1$ разряда *обязана* соответствовать общему количеству комбинаций, которые можно получить в первых k разрядах.

То, что всё человечество сейчас использует именно систему по основанию 10, есть не более чем случайность: основание системы счисления соответствует количеству пальцев у нас на руках. Работа именно с этой системой кажется нам «простой» и «естественной» лишь только потому, что мы привыкаем к ней с раннего детства; на самом деле, как мы позже увидим, считать в двоичной системе гораздо проще, там не нужна таблица умножения (вообще; то есть её там просто нет), а само умножение в столбик, столь ненавистное школьникам младших классов, в двоичной системе превращается в тривиальную процедуру «выписываний со сдвигами». Ещё в XVII веке Готфрид Вильгельм Лейбниц, впервые в истории описавший ***двоичную систему счисления*** в том виде, в котором она известна сейчас, заметил это обстоятельство и заявил, что использование десятичной системы — это роковая ошибка человечества.

Так или иначе, мы можем при желании использовать *любое* количество цифр, начиная от двух, для создания позиционной системы счисления; если при этом следовать традиционному подходу и, исполь-

зуя n цифр, приписывать им значения от 0 до $(n-1)$, то[35] с такой системой записи чисел можно будеть работать во многом по аналогии с хорошо знакомой нам десятичной системой. Например, в любой системе счисления запись числа 1000_n (где n — основание системы счисления) будет означать n^3: в десятичной системе это тысяча, в двоичной — 8, в пятеричной — 125. Необходимо только помнить одну важную вещь. Система счисления определяет, *как будет записано число*, но **само число и его свойства никак от системы счисления не зависят**; простое число всегда останется простым, чётное — чётным, $5 \cdot 7$ будет 35 вне всякой зависимости от того, какими цифрами (да хоть римскими!) мы запишем эти числа.

Прежде чем продолжать рассмотрение других систем, отметим два свойства обыкновенной десятичной записи числа, которые без изменений обобщаются на системы счисления по другому основанию. Первое из них непосредственно вытекает из определения позиционной записи. Если число представлено цифрами $\overline{d_k d_{k-1} \ldots d_2 d_1 d_0}$, то его численное значение будет $\sum_{i=0}^{k} 10^k d_k$; например, для числа 3275 его значение вычисляется как $3 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0 = 3000 + 200 + 70 + 5 = 3275$. Второе свойство требует чуть более длинного объяснения, но, по большому счёту, ничуть не сложнее: если последовательно делить число на 10 с остатком и выписывать остатки, и так до тех пор, пока очередным частным не окажется ноль, то мы получим (в виде выписанных остатков) все цифры, составляющие это число, начиная с самой младшей. Например, поделим 3275 на 10 с остатком, получим 327 и 5 в остатке; поделим 327 на 10, получим 32 и 7 в остатке, поделим 32, получим 3 и 2 в остатке, поделим 3 на 10, получим 0 и 3 в остатке. Последовательность остатков выглядит так: $5, 7, 2, 3$; это и есть цифры числа 3275.

Оба этих свойства обобщаются на систему счисления по *произвольному* основанию, только вместо 10 в вычислениях нужно использовать соответствующее основание системы счисления. Например, для *семеричной* записи 1532_7 численное значение будет $1 \cdot 7^3 + 5 \cdot 7^2 + 3 \cdot 7^1 + 2 = 611$ (естественно, все подсчёты мы выполняем в десятичной системе, поскольку нам так проще). Попробуем теперь выяснить, из каких цифр состоит семеричная запись числа 611, для чего последовательно выполним несколько делений на 7 с остатком. Результатом первого деления будет 87, в остатке 2; результатом второго — 12, в остатке 3; результатом третьего — 1, в остатке 5; результатом четвёртого — 0, в остатке 1. Итак, семеричная запись числа 611 состоит из цифр $2, 3, 5, 1$, если перечислять их начиная с младшей, то есть эта запись — 1532_7 (где-то мы это уже видели).

[35] Другие подходы возможны; например, довольно часто в литературе упоминается система счисления, использующая три цифры, значения которых — 0, 1 и -1; такие системы счисления мы оставим за рамками нашей книги, но заинтересованный читатель легко найдёт их описания в других источниках.

Как видим, первое из двух сформулированных свойств позиционной записи позволяет перевести число из любой системы счисления в ту, в которой мы привыкли проводить вычисления (для нас это система по основанию 10), а второе свойство — перевести число из привычной нам записи (то есть десятичной) в запись в произвольной системе счисления.

Отметим, что при переводе числа из «какой-то другой» системы в десятичную можно сэкономить на умножениях, представив

$$d_k n^k + d_{k-1} n^{k-1} + \ldots + d_1 n + d_0$$

в виде

$$(\ldots((d_k \cdot n + d_{k-1}) \cdot n + d_{k-2}) \cdot n + \ldots + d_1) \cdot n + d_0$$

Например, всё то же 1532_7 можно перевести в десятичную систему, вычислив $((1{\cdot}7{+}5){\cdot}7{+}3){\cdot}7{+}2 = (12{\cdot}7{+}3){\cdot}7{+}2 = 87{\cdot}7{+}2 = 609{+}2 = 611$.

Можно заметить, что традиционный порядок записи чисел от старших цифр к младшим, к которому мы привыкли с детства, при переводах из одной системы счисления в другую часто оказывается не очень удобным; например, нам уже приходилось, выписав последовательность остатков от деления на основание, потом «переворачивать» эту последовательность, чтобы получить искомую запись числа. Более того, если немного подумать, мы заметим, что, видя в тексте какое-нибудь десятичное число, например, 2347450, мы, в сущности, не знаем, что обозначает первая же из его цифр; это, конечно, «два» (или «две»), но *чего?* Десятков тысяч? Сотен тысяч? Миллионов? Десятков миллионов? Выяснить это мы можем, лишь просмотрев запись числа до конца и узнав, сколько в нём цифр; только после этого, вернувшись к началу записи, мы поймём, что в этот раз двойка действительно означала два миллиона, а не что-то иное.

Но почему же весь мир использует именно такую «неудобную» запись? Ответ оказывается неожиданно прост: цифры, которые мы используем, не случайно называются *арабскими*; согласно господствующим историческим представлениям, современная система записи чисел была изобретена индийскими и арабскими математиками предположительно в VI–VII вв. н. э., а своего практически окончательного вида достигла в сохранившихся до наших дней работах знаменитого мудреца Аль-Хорезми (от чьего имени происходит слово «алгоритм») и его современника Аль-Кинди; эти работы были написаны, согласно традиционной датировке, в IX веке. Так или иначе, *арабская письменность предполагает начертание слов и строк справа налево, а не слева направо, как привыкли мы*; так что для создателей десятичной системы счисления цифры в записи числа располагались именно так, как им было удобнее: при просмотре записи числа они сначала видели цифру, обозначающую единицы, потом цифру десятков, потом цифру сотен и так далее.

В программировании приходится часто сталкиваться с двоичной записью чисел, потому что именно так — в виде последовательностей нулей и единиц — в памяти компьютера представляются как числа, так и вся остальная информация. Поскольку выписывать ряды двоичных

цифр оказывается не очень удобно, программисты в качестве сокращённой записи часто применяют системы счисления по основанию 16 и 8: каждая цифра восьмеричной записи числа соответствует трём цифрам двоичной записи того же числа, а каждая цифра шестнадцатеричной записи — четырём двоичным цифрам. Поскольку для шестнадцатеричной системы требуется шестнадцать цифр, а арабских цифр всего десять, к ним добавляют шесть первых букв латинского алфавита: `A` обозначает 10, `B` — 11, `C` —12, `D` — 13, `E` — 14 и `F` — 15. Например, запись $\mathtt{3F}_{16}$ означает 63, $\mathtt{100}_{16}$ соответствует числу 256, а $\mathtt{111}_{16}$ — числу 273.

При работе с современными компьютерами ***двоичные цифры*** (***биты***), как правило, объединяются в группы по восемь цифр (так называемые ***байты***), что объясняет популярность шестнадцатеричной системы, ведь каждому байту соответствуют ровно две цифры такой записи: байт может принимать значения от 0 до 255, в шестнадцатеричной нотации это числа от $\mathtt{00}_{16}$ до $\mathtt{FF}_{16}$.

С восьмеричными цифрами такой фокус не проходит: для записи байта их требуется в общем случае три, потому что числа от 0 до 255 в восьмеричной системе представляются записями от 000_8 до 377_8; но при этом тремя восьмеричными цифрами можно записать числа, в байт не помещающиеся, ведь максимальное трёхзначное восьмеричное число — 777_8 — составляет 511. В частности, для записи *трёх* идущих подряд байтов нужно ровно восемь восьмеричных цифр, но на практике группы по три байта почти никогда не встречаются. Однако у восьмеричной системы счисления есть другое несомненное достоинство: она использует только арабские цифры. Собственно говоря, 8 есть максимальная степень двойки, не превосходящая 10; именно поэтому восьмеричная система была очень популярна у программистов до того, как восьмибитный байт стал устойчивой единицей измерения информации, но к настоящему времени она используется гораздо реже, чем шестнадцатеричная.

Поскольку цифр в двоичной системе всего две, перевод чисел в неё и из неё оказывается проще, чем с другими системами; в частности, если запомнить наизусть степени двойки (а у программистов это в любом случае получается само собой), при переводе в любую сторону можно обойтись без умножений. Пусть, например, нам нужно перевести в десятичную запись число $\mathtt{1001101}_2$; старшая единица, стоящая в седьмом разряде, соответствует шестой степени двойки, то есть 64, следующая единица стоит в четвёртом разряде и соответствует третьей степени двойки, то есть 8, следующая означает 4, и последняя, младшая — 1 (вообще говоря, младшая цифра равна самой себе в любой системе счисления). Складывая 64, 8, 4 и 1, получаем 77, это и есть искомое число.

Перевод из десятичной записи в двоичную можно выполнить двумя способами. Первый из них — традиционный: делить исходное число пополам с остатком, выписывая получающиеся остатки, пока в частном не останется ноль. Поскольку деление пополам нетрудно выполнить в уме, обычно всю операцию проводят, начертив на бумаге вертикальную линию; слева (и сверху вниз) записывают сначала исходное число, потом

103	
51	1
25	1
12	1
6	0
3	0
1	1
0	1
1100111	

76	
38	0
19	0
9	1
4	1
2	0
1	0
0	1
1001100	

Рис. 1.11. Перевод в двоичную систему делением пополам

результаты деления, а справа выписывают остатки. Например, при переводе числа 103 в двоичную систему получается: 51 и 1 в остатке; 25 и 1 в остатке; 12 и 1 в остатке; 6 и 0 в остатке; 3 и 0 в остатке; 1 и 1 в остатке; 0 и 1 в остатке (см. рис. 1.11, слева). Остаётся только выписать остатки, просматривая их снизу вверх, и мы получим $\texttt{1100111}_2$. Аналогично для числа 76 мы получим 38 и 0, 19 и 0, 9 и 1, 4 и 1, 2 и 0, 1 и 0, 0 и 1; выписывая остатки, получим $\texttt{1001100}_2$ (там же, справа).

Есть и другой способ, основанный на знании степеней двойки. На каждом шаге мы выбираем наибольшую степень двойки, не превосходящую оставшееся число, после чего выписываем единицу в соответствующий разряд, а из числа соответствующую степень вычитаем. Пусть, например, нам потребовалось перевести в двоичную систему число 757. Наибольшая степень двойки, не превосходящая его — девятая (512), остаётся 245. Следующая степень двойки будет седьмая (128, поскольку 256 не подходит); останется 117. Дальше точно так же вычитаем 64, остаётся 53; вычитаем 32, остаётся 21; вычитаем 16, остаётся 5; вычитаем 4, остаётся 1, вычитаем 1 (нулевую степень двойки), остаётся 0. Результат будет $\texttt{1011110101}_2$. Этот способ особенно удобен, если исходное число немного превосходит какую-то из степеней двойки: например, число 260 таким способом в «двоичку» переводится почти мгновенно: $256 + 4 = 100000100_2$.

Поскольку, как мы уже говорили, для сокращения записи двоичных чисел программисты часто употребляют системы счисления по основанию 16 и (чуть реже) 8, часто возникает потребность переводить из двоичной системы в эти две и обратно. К счастью, **если основание одной системы счисления представляет собой натуральную степень n основания другой системы счисления, то одна цифра первой системы в точности соответствует n цифрам второй системы**. На практике это свойство применяется только к переводам между двоичной системой и системами по основанию 8 и 16, хотя точно так же, например, можно было бы переводить числа из троичной системы в

Таблица 1.6. Двоичное представление восьмеричных и шестнадцатеричных цифр

восьмеричные		шестнадцатеричные			
d_8	bin	d_{16}	bin	d_{16}	bin
0	000	0	0000	8	1000
1	001	1	0001	9	1001
2	010	2	0010	A	1010
3	011	3	0011	B	1011
4	100	4	0100	C	1100
5	101	5	0101	D	1101
6	110	6	0110	E	1110
7	111	7	0111	F	1111

девятичную и обратно; просто ни троичная, ни девятичная системы счисления не нашли широкого практического применения.

Для перевода числа из восьмеричной системы в двоичную *каждую цифру исходного числа заменяют соответствующими тремя двоичными цифрами* (см. табл. 1.6). Например, для числа $\texttt{3741}_8$ это будут группы цифр 011 111 100 001, незначащий ноль можно будет отбросить, так что в результате получится $\texttt{11111100001}_2$. Для перевода из шестнадцатеричной в двоичную делают то же самое, но каждую цифру заменяют на *четыре* двоичные цифры; например, для $\texttt{2A3F}_{16}$ получим 0010 1010 0011 1111, а после отбрасывания незначащих нулей — $\texttt{101010001111111}_2$.

Для перевода в обратную сторону исходное двоичное число **справа налево** (это важно) разбивают на группы по три или четыре цифры (соответственно для перевода в восьмеричную или шестнадцатеричную системы); если в старшей группе не хватает цифр, её дополняют незначащими нулями. Затем каждую получившуюся группу заменяют соответствующей цифрой. Рассмотрим, например, число $\texttt{10000101111011}_2$. Для перевода в восьмеричную систему разобьём его на группы по три цифры, дописав справа незначащий ноль: 010 000 101 111 011; теперь каждую группу заменим на соответствующую восьмеричную цифру и получим $\texttt{20573}_8$. Для перевода того же числа в шестнадцатеричную систему разобьём его на группы по четыре цифры, дописав в начало два незначащих нуля: 0010 0001 0111 1011; заменив их на соответствующие шестнадцатеричные цифры, получим $\texttt{217A}_{16}$.

Комбинации двоичных цифр, приведённые в таблице 1.6, программисты обычно просто помнят, но специально их зазубривать не обязательно: при необходимости их легко вычислить, а через некоторое время они сами собой уложатся в памяти.

Двоичные дроби переводятся в десятичную систему аналогично тому, как мы переводили целые числа, только цифрам после «двоичной запятой»[36] соответствуют дроби $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ и т. д. Например, для числа $\texttt{101.01101}_2$ имеем $4 + 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} = 5.40625$. Можно поступить и иначе: сообразив, что один знак после «запятой» — это «половинки», два знака — это «четвертинки», три знака — «восьмушки» и так далее, мы можем рассмотреть всю дробную часть как целое число и разделить его на соответствующую степень двойки. В рассматриваемом случае у нас пять знаков после «запятой» — это тридцать вторые части, а $\texttt{1101}_2$ есть 13_{10}, так что имеем $5\frac{13}{32} = 5.40625$.

Обратный перевод дробного числа из десятичной системы в двоичную выполнить тоже несложно, но несколько труднее объяснить, почему это делается именно так. Для начала отдельно переводим целую часть числа, выписываем результат и забываем о ней, оставив только десятичную дробь, заведомо меньшую единицы. Далее нам необходимо выяснить, *сколько половинок* (одна или ни одной) у нас имеется в этой дробной части. Для этого достаточно умножить её на два. В полученном числе целая часть может быть равна нулю или единице, это и есть искомое «количество половинок» в исходном числе. Какова бы ни была полученная целая часть, мы выписываем её в качестве очередной двоичной цифры, а из рабочего числа убираем, поскольку уже учли её в результате. Оставшееся число снова представляет собой дробь, заведомо меньшую единицы, ведь целую часть мы только что отсекли; умножаем эту дробь на два, чтобы определить «количество четвертинок», выписываем, отсекаем, умножаем на два, определяем «количество восьмушек» и так далее.

Например, для уже знакомого нам числа 5.40625 перевод обратно в двоичную систему будет выглядеть так. Целую часть сразу же переводим как обычное целое число, получаем `101`, выписываем результат, ставим двоичную точку и на этом забываем про целую часть нашего исходного числа. Остаётся дробь 0.40625. Умножаем её на два, получаем 0.8125. Поскольку целая часть равна нулю, выписываем в результат цифру 0 (сразу после запятой) и продолжаем процесс. Умножение 0.8125 на два даёт 1.625; выписываем в результат единичку, убираем её из рабочего числа (получается 0.625), умножаем на два, получаем 1.25, выписываем единичку, умножаем 0.25 на два, получаем 0.5, выписываем ноль, умножаем на два, получаем 1.0, выписываем единичку. На этом перевод заканчивается, поскольку в рабочем числе у нас остался ноль, и его, понятное дело, сколько ни умножай, будут получаться одни нули;

[36] Здесь и далее в записи позиционных дробей мы будем использовать для отделения дробной части точку, а не запятую, как это обычно делается в русскоязычной литературе. Дело в том, что в английских текстах в этой роли всегда использовалась именно точка, в результате чего она же используется во всех существующих языках программирования. Занимаясь программированием, следует приучить себя к мысли, что никакой «десятичной запятой» не бывает, а бывает «десятичная точка».

заметим, в принципе мы имеем право так делать — ведь к полученной двоичной дроби можно при желании дописать справа сколько угодно нулей, все они будут незначащие. Выписанный результат у нас получился $\texttt{101.01101}_2$, что, как мы видели, как раз и есть двоичное представление числа 5.40625.

Далеко не всегда всё будет столь благополучно; в большинстве случаев вы получите *бесконечную* (но, конечно, периодическую) двоичную дробь. Чтобы понять, почему это происходит так часто, достаточно вспомнить, что любую конечную или периодическую десятичную дробь можно представить в виде несократимой простой дроби с целым числителем и натуральным знаменателем; собственно говоря, это есть определение *рационального числа*. Так вот, нетрудно видеть, что **в виде конечной двоичной дроби представимы такие и только такие рациональные числа, у которых в знаменателе степень двойки**. Конечно, аналогичное ограничение присутствует и для десятичных дробей, а также и в системе счисления по вообще любому основанию, но в общем случае это ограничение формулируется мягче: *рациональное число, представленное в виде несократимой дроби $\frac{m}{n}$, представимо в виде конечной дроби в системе по основанию N тогда и только тогда, когда существует целая степень числа N, делящаяся нацело на n*. В частности, дробь $\frac{49}{50}$ можно представить в виде конечной десятичной дроби, потому что $10^2 = 100$ делится на 50 нацело; аналогично можно записать в виде конечной десятичной дроби число $\frac{77}{80}$, потому что на 80 без остатка поделится $10^4 = 10000$. В общем случае у нас несократимая простая дробь превратится в конечную десятичную, если её знаменатель раскладывается на простые множители в виде $2^k \cdot 5^m$: при этом нам достаточно выбрать бóльшее из k и m и использовать его в качестве степени, в которую возвести 10.

Для случая двоичной системы всё жёстче: степень двойки, какова бы она ни была, нацело может поделиться только на другую степень двойки. В применении к переводу из десятичной системы в двоичную следует заметить, что любая десятичная дробь есть число вида $\frac{n}{10^k}$; чтобы в знаменателе остались одни двойки, необходимо, чтобы числитель нужное число раз делился на пять. Так, рассмотренное в примере выше 5.40625 есть $\frac{540625}{10^5}$, но числитель 540625 прекрасно делится без остатка на $5^5 = 3125$ (результатом деления будет 173), поэтому после сокращения нужного числа пятёрок в знаменателе остаётся степень двойки, что и позволяет записать это число в виде конечной двоичной дроби. Но так, конечно, происходит не всегда; в большинстве случаев (в частности, всегда, когда последняя значащая десятичная цифра дроби отлична от 5) получаемая двоичная дробь окажется бесконечной, хотя и периодичной. В таком случае нужно выполнять описанную выше процедуру с последовательным умножением на два, пока вы не получите рабочее число, которое уже видели; это будет означать, что вы попали

в период; напомним, что периодическая дробь записывается путём заключения её периода в круглые скобки. Например, для числа 0.35_{10} у нас получится 0.7 (выписываем 0), 1.4 (выписываем 1, оставляем 0.4), 0.8 (выписываем 0), 1.6 (выписываем 1, оставляем 0.6), 1.2 (выписываем 1, оставляем 0.2) — и, наконец, 0.4, которое мы уже видели четыре шага назад. Следовательно, период дроби составляет четыре цифры, а в результате $0.35_{10} = 0.01(0110)_2$.

1.5.3. Двоичная логика

В программировании мы часто сталкиваемся с проверками всевозможных *условий*, таких как «не слишком ли длинная строка», «не оказался ли дискриминант отрицательным», «хватит ли нам отведённого пространства», «существует ли нужный файл», «выбрал ли пользователь этот вариант работы или другой» и прочее; начав программировать, мы очень быстро убедимся, что выполнение даже самых простых программ основано на проверках условий. При этом сами условия представляют собой *логические выражения*, то есть такие выражения, которые *вычисляются*, а результатом вычисления становится *логическое значение — ложь* или *истина*.

Раздел математики, в котором изучаются выражения такого рода, называется математической логикой; надо сказать, что это достаточно сложная область знаний, включающая множество нетривиальных теорий, и на глубокое изучение математической логики может уйти вся жизнь. Мы в нашей книге рассмотрим только одну из самых примитивных составляющих математической логики — так называемую ***алгебру двоичной логики***.

По своей сути двоичная логика похожа на арифметику, но вместо бесконечного множества чисел здесь используется множество, состоящее всего из двух значений: 0 (ложь) и 1 (истина). Над этими значениями определены разные операции, результатом которых тоже является 0 или 1. Одной из самых простых логических операций является ***отрицание***, «не x», которое обозначается $\overline{x}$; в книгах можно встретить также другие обозначения, например $\neg x$. Операция отрицания меняет значение на противоположное, то есть $\overline{1} = 0$ и $\overline{0} = 1$. Поскольку у операции отрицания всего один аргумент, говорят, что она ***унарная***.

Самыми известными и часто употребляемыми логическими ***бинарными*** операциями, то есть операциями *двух* аргументов, можно считать ***логическое или*** и ***логическое и***, которые в математике называются также ***дизъюнкцией*** и ***конъюнкцией***. «Логическое или» между двумя логическими значениями будет истинно, когда хотя бы одно из исходных значений истинно; естественно, они могут быть истинными одновременно, тогда «логическое или» между ними тоже останется истинным. Единственный вариант, когда «логическое или» оказывается

Таблица 1.7. Основные бинарные логические операции

аргументы x	y	конъюнкция $x\&y$, $x \wedge y$	дизъюнкция $x \vee y$	искл. или $x \oplus y$	импликация $x \to y$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1

ложным — это когда оба его аргумента ложны. «Логическое и», напротив, истинно тогда и только тогда, когда истинны оба его аргумента, а во всех остальных случаях ложно.

Операция «логического или» обычно обозначается знаком «$\vee$», что касается операции «логического и», то наиболее популярное обозначение для неё — амперсанд «&», но во многих современных учебниках эту операцию почему-то предпочитают обозначать знаком «$\wedge$».

Помимо конъюнкции и дизъюнкции, достаточно часто встречаются операции ***исключающее или*** и ***импликация***. «Исключающее или», обозначаемое знаком «$\oplus$», истинно, когда истинен один из его аргументов, но не оба сразу; от обычного «или» эта операция отличается своим значением для случая обоих истинных аргументов.

Операция импликации (обозначается «$\to$») несколько более сложна для понимания. Она основана на принципе, что в рассуждениях из истинных посылок может следовать только истина, а вот из ложных посылок может следовать что угодно, то есть может получиться истина, а может — ложь. Чтобы понять, почему это так, можно припомнить, что далеко не все научные теории были истинными, но при этом замечательно использовались и даже давали правильные результаты. В частности, известно, что братья Монгольфье подняли в воздух первый в истории воздушный шар, наполнив его дымом от смеси соломы и шерсти; солому они рассматривали как растительное начало жизни, а шерсть — как животное начало, что, по их мнению, должно было привести к возможности полёта, и полёт *действительно состоялся*, несмотря на то, что животное и растительное начало к этому не имели никакого отношения. Иначе говоря, *возможно получить абсолютно правильный результат, отталкиваясь от абсолютно ложных посылок.* В соответствии с этим импликация ложна лишь в том случае, если её левый аргумент — истина, а правый — ложь (из истины ложь следовать не может), во всех остальных случаях импликация считается истинной.

Если обозначить множество $\{0, 1\}$ буквой B[37], то логические операции двух аргументов можно рассматривать как *функции*, обла-

[37] От слова *Boolean*, по имени английского математика Джорджа Буля, который впервые предложил формальную систему, известную сейчас как алгебра логики, или булева алгебра.

Таблица 1.8. Все возможные двоичные функции двух аргументов

x	y	0	$\&$	$>$	x	$<$	y	$\oplus$	$\vee$	$\downarrow$	$\equiv$	$\overline{y}$	$\leftarrow$	$\overline{x}$	$\rightarrow$	$\mid$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

стью определения которых является $B \times B$ (то есть множество пар $\{(0,0),(0,1),(1,0),(1,1)\}$), а областью значений — само B. Поскольку область определения логических функций *конечна*, их можно задавать таблицами, в которых в явном виде указано значение для каждого элемента из области определения. Такие таблицы для логических функций называются ***таблицами истинности***; в частности, таблица 1.7 содержит таблицу истинности для рассмотренных нами конъюнкции, дизъюнкции, «исключающего или» и импликации.

Поскольку логическая функция двух аргументов полностью определяется своими значениями, которых у неё четыре, то есть, попросту говоря, набором из четырёх двоичных значений, мы, припомнив наши знания комбинаторики, можем заключить, что всего их существует 16. Множество всех возможных логических функций двух аргументов показано в таблице 1.8, где они упорядочены по возрастанию их набора значений, как если бы набор значений был записью двоичного числа. Начинается перечисление с функции «константа 0», которая ложна для любых аргументов, а заканчивается «константой 1», или ***тавтологией*** — функцией, которая, напротив, на любых аргументах истинна. Как можно легко убедиться, таблица содержит, в числе прочего, только что рассмотренные нами конъюнкцию, дизъюнкцию, импликацию и «исключающее или». Кроме того, в таблице обнаруживаются функции x (всегда равна первому аргументу, независимо от второго) и y (всегда равна второму аргументу, независимо от первого), а также их отрицания; функции, обозначенные как «$\downarrow$» и «$|$», называются соответственно «стрелка Пирса» и «штрих Шеффера» и представляют собой отрицание дизъюнкции и отрицание конъюнкции:

$$x \downarrow y = \overline{x \vee y} \qquad\qquad x \mid y = \overline{x \,\&\, y}$$

Функция, обозначенная знаком «$\equiv$», называется эквивалентностью: она истинна, когда её аргументы равны, и ложна, если аргументы различаются. Нетрудно убедиться, что эквивалентность представляет собой отрицание «исключающего или». Остаются ещё три функции, это «импликация наоборот» («$x \leftarrow y = y \rightarrow x$»), а также функции «больше» и «меньше», которые представляют собой отрицание импликаций. Итого, начав с констант, мы перечислили ровно 16 функций, то есть все.

Если рассмотреть логические функции трёх аргументов, то область их определения — множество $B \times B \times B$ — состоит из восьми троек $\{(0,0,0),(0,0,1),(0,1,0),\ldots,(1,1,0),(1,1,1)\}$; как следствие, логическая функция трёх аргументов определяется набором из *восьми* значений, а всего таких функций будет $2^8 = 256$. В общем случае логическая функция n аргументов определена на множестве мощности 2^n и определяется, соответственно, упорядоченным набором из 2^n значений, а всего таких функций, очевидно, 2^{2^n}. Так, функций от четырёх аргументов оказывается $2^{16} = 65536$, функций от пяти аргументов — $2^{32} = 4294967296$ и так далее. **Если в этом абзаце что-то показалось непонятно, перечитайте параграф о комбинаторике; если и это не помогло, обязательно найдите кого-нибудь, кто сможет вам объяснить, что тут происходит.** Дело здесь, разумеется, не в количестве функций; но если вы тут чего-то не поняли, то у вас определённо есть проблемы с простейшей комбинаторикой, а вот это уже никуда не годится.

Вернувшись к функциям двух аргументов, отметим, что конъюнкция и дизъюнкция во многом аналогичны умножению и сложению: так, конъюнкция с нулём, как и умножение на ноль, всегда даёт ноль; конъюнкция с единицей, как и умножение на единицу, всегда даёт исходный аргумент; дизъюнкция с нулём, как и сложение с нулём, тоже всегда даёт исходный аргумент. Из-за этого сходства математики часто в формулах опускают обозначение конъюнкции, будь то «$x \,\&\, y$» или «$x \wedge y$», и пишут просто «xy» или «$x \cdot y$». В частности:

$$x \cdot 0 = 0 \qquad x \cdot 1 = x \qquad x \vee 0 = x \qquad x \vee 1 = 1$$

Запоминать эти соотношения не надо! Достаточно вспомнить, что означает конъюнкция или дизъюнкция, и все четыре приведённых соотношения оказываются совершенно очевидны. В самом деле, конъюнкция равна единице только при обоих единичных аргументах, так что если один из аргументов конъюнкции равен нулю, то каков бы ни был второй, вся она вместе уже единицей не станет, т. е. $x \cdot 0 = 0$; в то же время если один из её аргументов уже заведомо единица, то нужно, чтобы второй был единицей, тогда и вся она окажется единицей, иначе она будет нулём; то есть при одном из аргументов заведомо единичном конъюнкция равна второму аргументу: $x \cdot 1 = x$. Аналогично если один из аргументов дизъюнкции заведомо единичный, то этого достаточно, и в ноль её уже не превратить, т. е. $x \vee 1 = 1$; ну а если один из аргументов дизъюнкции — ноль, то это ещё не приговор, ведь второй может оказаться единицей, и тогда она вся будет единицей: $x \vee 0 = x$.

Точно так же не нуждаются в запоминании соотношения, позволяющие раскрывать скобки в выражениях, состоящих из конъюнкции, дизъюнкции и отрицания:

$$(x \vee y)z = xz \vee yz \qquad\qquad (xy) \vee z = (x \vee z)(y \vee z)$$

Первое соотношение получается из рассуждения «для истинности всего выражения слева необходимо и достаточно, чтобы хотя бы одна из двух переменных x и y была истинна, при этом чтобы была истинна переменная z; но это то же самое, как если сказать, что нам нужно, чтобы или оказались одновременно истинными x и z (истинная x сделает истинной скобку, z должна быть истинной, иначе конъюнкция не может быть истинной), или по тем же причинам должны быть одновременно истинными y и z». Второе соотношение получается рассуждением «истинность выражения могут обеспечить или истинность z, или одновременная истинность x и y; это то же самое, как сказать, что у нас должны быть одновременно истинными две скобки, причём истинность первой из них может обеспечить либо x, либо z, а истинность второй — либо y, либо z».

Приведём ещё несколько элементарных соотношений:

$$x \,\&\, x = x \qquad x \vee x = x \qquad x\overline{x} = 0 \qquad x \vee \overline{x} = 1 \qquad \overline{\overline{x}} = x$$

Их, разумеется, тоже не нужно запоминать. Соответствующие рассуждения предлагаем читателю найти самостоятельно.

Отдельного упоминания заслуживают так называемые *законы де Моргана*:

$$\overline{x \vee y} = \overline{x} \,\&\, \overline{y} \qquad\qquad \overline{x \,\&\, y} = \overline{x} \vee \overline{y}$$

Почему-то то обстоятельство, что эти соотношения являются *именными*, то есть носят имя человека, якобы их открывшего, нагоняет на многих новичков изрядный страх; уж если эти соотношения кому-то потребовалось «открывать», да ещё за это открытие соотношения назвали именем того, кто их открыл, то тут уж точно без зубрёжки никак. Между тем здесь *на самом деле всё совершенно элементарно.* Первое соотношение: «Что нам нужно, чтобы сделать ложной дизъюнкцию? Если хотя бы один аргумент будет истинным, то она вся станет истинной; следовательно, для её ложности необходимо, чтобы оба аргумента были ложными, то есть и x был ложным, и y был ложным». Второе соотношение: «Что нам нужно, чтобы сделать ложной конъюнкцию? Очевидно, достаточно, чтобы хотя бы один из её аргументов был ложным». Вот вам и все «великие и ужасные» законы де Моргана.

В заключение обзора двоичной логики дадим одну рекомендацию. Если вам пришлось решать задачу, в которой дана некая логическая формула и с ней что-то предлагается сделать, то для начала убедитесь, что в формуле применяются только конъюнкция, дизъюнкция и отрицание. Если это не так, немедленно избавьтесь от всех остальных знаков операций, сведя их к первым трём; отметим, что это *всегда*

можно сделать. Возьмём, например, загадочную импликацию. Для её истинности достаточно, чтобы первый аргумент был ложным (ведь из лжи может следовать что угодно); точно так же достаточно, чтобы второй аргумент был истинным (ведь истина может следовать как из истины, так и из лжи). Получаем, что

$$x \to y = \overline{x} \vee y$$

Точно так же, если вам встретилось «исключающее или», замените его одним из следующих способов: «один из аргументов должен быть истинным, но не два одновременно» или «нужно, чтобы один был ложным, другой истинным, или наоборот»:

$$x \oplus y = (x \vee y)\overline{(xy)} \qquad\qquad x \oplus y = x\overline{y} \vee \overline{x}y$$

Заметим, что из первого раскрытием скобок получается второе:

$$(x \vee y)\overline{(xy)} = (x \vee y)(\overline{x} \vee \overline{y}) = x\overline{x} \vee y\overline{x} \vee x\overline{y} \vee y\overline{y} = x\overline{y} \vee \overline{x}y$$

Между прочим, в аналогичном виде можно записать вообще любую логическую функцию от произвольного количества переменных, просто посмотрев на её таблицу истинности. Выбрав те строки, где значение функции — 1, мы для всех таких строк выписываем *конъюнкты*, состоящие из всех переменных, причём те, которые в этой строке равны нулю, в конъюнкт входят со знаком отрицания. Все полученные конъюнкты (которых будет столько, на скольких наборах переменных функция равна единице) записываем через знак дизъюнкции. Например, для стрелки Пирса соответствующая запись будет состоять из одного конъюнкта $\overline{xy}$, а для штриха Шеффера — из трёх конъюнктов $\overline{xy} \vee \overline{x}y \vee x\overline{y}$. Если же, к примеру, мы рассмотрим функцию трёх аргументов $f(x, y, z)$, которая равна единице на четырёх наборах $\{(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)\}$, а на всех остальных наборах равна нулю, то соответствующее выражение для этой функции будет выглядеть так: $\overline{xyz} \vee \overline{xy}z \vee \overline{x}yz \vee xyz$. Такая форма записи логической функции называется ***дизъюнктивной нормальной формой*** (ДНФ).

1.5.4. Виды бесконечности

Материал этого параграфа может показаться вам слишком «заумным»; в принципе, следует признать, что к практическому программированию всё это прямого отношения не имеет, или, точнее говоря, программировать можно без этого. К сожалению, если вы пропустите этот параграф, то следующие параграфы этой главы, в которых рассказывается про алгоритмы и теорию вычислимости, вы тоже не поймёте; но если математика вам уже надоела, вы можете спокойно пропустить их все. Никто не мешает вернуться сюда позже, когда вы будете к этому готовы. Если вы действительно решите пропустить остаток «математической» главы, запомните только одну вещь: термин «алгоритм» на

самом деле гораздо сложнее, чем о нём принято думать. В частности, **определения алгоритма не существует и существовать не может, причём вообще ни в каком виде и ни в каком смысле**. О том, почему это так, рассказывается в следующем параграфе.

Вернёмся к нашему предмету обсуждения. Из школьного курса математики мы знаем, что, например, чисел существует «бесконечно много»; этот факт выражается в том, что, сколь бы огромное число мы ни взяли, мы всегда можем прибавить к нему единицу, получив число ещё бóльшее. В школе обычно этим и удовлетворяются, но в курсе высшей математики приходится отметить, что **бесконечности бывают разные**. «Наименьшая» из них — это так называемая ***счётная бесконечность***; множество считается счётно-бесконечным, если его элементы можно перенумеровать натуральными числами в каком-то порядке. Простейший пример счётной бесконечности — само по себе множество натуральных чисел: числу 1 мы дадим номер первый, числу 2 — номер второй и так далее. Множество целых чисел (в которое, наряду с натуральными, входят также ноль и отрицательные целые, то есть «натуральные с минусом») также является счётно-бесконечным: например, можно первый номер отдать числу 0, второй номер — числу 1, третий номер — числу -1, числа 2 и -2 получат соответственно четвёртый и пятый номера, и так далее; каждое положительное число k при этом получает номер $2k$, а каждое отрицательное число $-m$ — номер $2m+1$ (например, число 107 получит номер 214, а число -751 — номер 1503). Множество *чётных* натуральных чисел тоже счётное: число 2 получает номер первый, число 4 — номер второй, и так до бесконечности, то есть каждое число вида $2n$ получает номер n. Получается, что натуральных чисел *ровно столько же*, сколько натуральных чётных, и, с другой стороны, *ровно столько же*, сколько всех целых. «Количество» натуральных чисел математики обозначают символом $\aleph_0$ (читается «алеф-нуль»).

При внимательном рассмотрении оказывается, что множество рациональных чисел, то есть чисел, представимых в виде несократимой дроби $\frac{m}{n}$, где m — целое, а n — натуральное,— тоже счётное. Чтобы их занумеровать, представьте себе координатную полуплоскость, где по горизонтальной оси откладываются значения знаменателя дроби — напомним, таковые должны быть натуральными, то есть начинаться с единицы,— а по вертикальной откладываются значения числителя. Иначе говоря, нам нужно придумать некую нумерацию для целочисленных точек координатной плоскости, лежащих справа от вертикальной оси, при этом нужно, чтобы в нумерацию каждое число попало только один раз, то есть, например, дробям $\frac{1}{2}$, $\frac{2}{4}$, $\frac{3}{6}$ и т. д. приписывать различные номера не нужно, ведь эти дроби обозначают одно и то же число. Впрочем, это как раз просто: при нумерации следует, во-первых, пропускать все *сократимые* дроби, а во-вторых, все дроби с числителем 0, кроме самой первой такой дроби — $\frac{0}{1}$, которая будет обозначением нуля. Простейший пример такой нумерации строится «уголками», расходящимися от начала координат. Номер первый мы дадим дроби $\frac{0}{1}$. На

первый «уголок» попадут дроби $\frac{1}{1}$. $\frac{0}{2}$ и $\frac{-1}{1}$, но дробь $\frac{0}{2}$ мы, как договорились, пропустим, а вот две остальные (числа 1 и -1) получат номера второй и третий. Двигаясь вдоль следующего «уголка», занумеруем: $\frac{2}{1}$ (№ 4), $\frac{1}{2}$ (№ 5), $\frac{0}{3}$ (пропускаем), $\frac{-1}{2}$ (№ 6) и $\frac{-2}{1}$ (№ 7). На следующем после этого «уголке» мы уже будем вынуждены пропускать сократимые дроби: $\frac{3}{1}$ (№ 8), $\frac{2}{2}$ (пропускаем), $\frac{1}{3}$ (№ 9), $\frac{0}{4}$ (пропускаем), $\frac{-1}{3}$ (№ 10), $\frac{-2}{2}$ (пропускаем), $\frac{-3}{1}$ (№ 11). Продолжая процесс «до бесконечности», мы припишем натуральные номера *всем* рациональным числам.

Предложение «продолжать процесс до бесконечности» может показаться неконструктивным, ведь в нашем распоряжении нет и не может быть «бесконечного» времени, но в данном случае бесконечное время нам и не нужно. Важно то, что *каково бы ни было рациональное число, мы сможем установить, какой именно номер оно имеет в нашей нумерации*, причём сделать это мы сможем за *конечное время*, сколь бы «заковыристое» число нам ни дали.

Можно очень легко доказать, что **множество *бесконечных десятичных дробей*** (то есть тех самых иррациональных[38] чисел) **счётным не является**. В самом деле, пусть мы придумали некую нумерацию бесконечных десятичных дробей. Рассмотрим теперь, например, дробь, имеющую нулевую целую часть; в качестве её первой цифры после запятой возьмём любую цифру, кроме той, которая является первой цифрой в дроби № 1; в качестве второй — любую, кроме той, которая является второй цифрой в дроби № 2, и так далее. Полученная дробь будет заведомо отличаться от *каждой* дроби из попавших в нашу нумерацию, ведь от дроби с номером n она отличается по меньшей мере цифрой в n-ной позиции после запятой. Получается, что в нашей (бесконечной!) нумерации эта новая дробь номера не имеет, причём это никак не зависит от того, как конкретно мы попытались занумеровать дроби. Нетрудно видеть, что таких «неучтённых» дробей — бесконечное количество, хотя, на самом деле, это не так уж важно. Итак, никакая нумерация не может охватить всё множество бесконечных десятичных дробей. Применённая здесь схема доказательства носит название ***канторовского диагонального метода*** в честь придумавшего её немецкого математика Георга Кантора.

Говорят, что множество бесконечных десятичных дробей имеет мощность ***континуум***; математики обозначают это символом $\aleph_1$ («алеф-один»). Чтобы понять, насколько это «много», проведём некий мысленный эксперимент, тем более что его результаты нам пригодятся при рассмотрении теории алгоритмов. Пусть у нас есть некий ***алфавит***,

[38] На всякий случай напомним, что ***иррациональным*** называется число, которое не может быть представлено в виде дроби $\frac{m}{n}$, где m — целое, а n — натуральное; примером такого числа может служить $\sqrt{2}$. Очень важно не делать при этом распространённую, но от этого не менее чудовищную ошибку, записывая в «иррациональные» все бесконечные десятичные дроби, такие как $\frac{1}{3}$ или $\frac{2}{7}$. «Бесконечность» периодических дробей на самом деле обусловлена выбором системы счисления и не имеет никакого отношения к свойству числа как такового; так, в системе счисления по основанию 21 обе упомянутые дроби будут иметь конечное «21-ичное» представление, тогда как $\frac{1}{2}$ окажется дробью бесконечной.

то есть некое *конечное* множество «символов», чем бы эти символы ни были: это могут быть буквы, цифры, вообще любые знаки, но с таким же успехом это могут быть вообще элементы любой природы, лишь бы их было конечное множество. Обозначим алфавит буквой A. Рассмотрим теперь *множество всех* ***конечных*** *цепочек*, составленных из символов алфавита A, то есть множество конечных последовательностей вида $a_1, a_2, a_3, \ldots a_k$, где каждое $a_i \in A$. Такое множество, включающее в себя пустую цепочку (цепочку длины 0), обозначают A^*. Нетрудно видеть, что, коль скоро алфавит конечен и каждая отдельно взятая цепочка тоже конечна (хотя их длину мы не ограничиваем, то есть можно рассматривать цепочки длиной в миллиард символов, в триллион, в триллион триллионов и так далее), всё множество цепочек окажется счётным. В самом деле, пусть алфавит включает n символов. Пустая цепочка получит номер 1; цепочки из одного символа получат номера с 2 по $n+1$; цепочки из двух символов, которых в общей сложности n^2, получат номера с $n+2$ по n^2+n+1; аналогично номера цепочек из трёх символов начнутся с n^2+n+2 и закончатся номером n^3+n^2+n+1 и так далее.

Рассмотрим теперь алфавит, состоящий из всех символов, когда-либо использовавшихся в книгах на Земле. Включим туда латиницу, кириллицу, греческий алфавит, арабские письмена, экзотические азбуки, используемые в грузинском и армянском языке, все китайские, японские и корейские иероглифы, вавилонскую клинопись, иероглифику древнего Египта, скандинавские руны, добавим цифры и все математические значки, подумаем некоторое время, не забыли ли чего-нибудь, добавим всё, что припомним или что нам посоветуют знакомые. Ясно, что такой супер-алфавит, несмотря на его довольно внушительные размеры, будет всё же конечен. Обозначим его буквой $\mathcal{V}$ и посмотрим, что у нас в результате попадёт в множество цепочек $\mathcal{V}^*$ (напомним, цепочки мы тоже рассматриваем только конечные).

Очевидно, в это множество угодят все книги, когда-либо написанные на Земле. Мало того, туда же попадут все книги, которые ещё не написаны, но когда-нибудь будут[39]; все книги, которые никогда не были и не будут написаны, но теоретически могли бы быть написаны; а равно и такие, которые никто и никогда не стал бы писать — например, все книги, текст которых представляет собой хаотическую последователь-

[39]Здесь встречается очевидное возражение, что в будущем могут появиться символы, которые мы не включили в $\mathcal{V}$; однако при ближайшем рассмотрении ничего от этого не меняется, достаточно придумать какой-нибудь способ обозначения таких символов через символы уже имеющиеся. Например, можно придумать какой-нибудь специальный символ, после которого обычными десятичными цифрами будет указываться номер «нового» символа, и эти номера присваивать всем новым символам по мере их появления. Больше того, можно вообще рассматривать алфавит из двух символов 0 и 1, а все остальные символы кодировать комбинациями нулей и единиц; собственно говоря, в памяти компьютеров всё именно так и происходит.

ность символов из разных систем письменности, где немецкая буква «ß» соседствует с дореволюционной русской «ѣ», а японские иероглифы перемежаются вавилонской клинописью.

Если вам и этого мало, представьте себе книгу толщиной в радиус Солнечной системы, а диагональю листа как от нас до соседней галактики, при этом набранную обычным 12-м шрифтом; ясно, что физически такую книгу сделать невозможно, но, тем не менее, в множество $\mathcal{V}^*$ войдёт не просто «эта книга», в него войдут *все такие книги*, отличающиеся друг от друга хотя бы одним символом. И не только эти, Вселенная ведь бесконечна! Кто мешает представить себе книгу размером в миллиард световых лет? А *все* такие книги?

Если ваше воображение ещё не дало сбоя, вам можно только позавидовать. И после всего этого мы вспоминаем, что **множество $\mathcal{V}^*$ «всего лишь» счётное**! Чтобы получить континуум, нам надо рассматривать не просто книги несуразно огромных размеров, нам потребуются *бесконечные* книги, такие, у которых вообще нет размера, которые никогда не кончаются, сколько бы мы ни двигались вдоль текста; в сравнении с этим книга форматом в миллиард световых лет оказывается чем-то вроде детской игрушки. Причём сама такая «бесконечная книга», отдельно взятая, ещё не даёт континуума, она ведь всего одна; нет, нам нужно **рассмотреть все бесконечные книги**, и вот тогда мы увидим, что рассматриваемое множество (бесконечных книг) имеет мощность континуум. Вот только не вполне понятно, как именно мы собираемся эти бесконечные книги «рассматривать», особенно прямо во всём их многообразии, если мы даже *одну* такую книгу представить себе толком не можем.

Вполне очевидно, что оперировать континуальными бесконечностями в каком бы то ни было конструктивном смысле невозможно; счётные бесконечности представляют собой абсолютный предел не только для человеческого, но и для любого другого мозга, если только такой мозг не окажется бесконечным сам по себе. Откровенно говоря, даже работая со счётными бесконечностями, мы никогда не рассматриваем сами бесконечности, мы просто заявляем, что каков бы ни был элемент N, всегда найдётся элемент $N+1$; иначе говоря, каков бы ни был уже рассмотренный набор элементов множества, мы всегда придумаем ещё один элемент, который входит во множество, но ещё не рассмотрен в предложенном наборе. Здесь, по большому счёту, нет никакой «бесконечности», есть просто наш отказ от рассмотрения некоего «потолка», выше которого почему-то запрещено прыгать.

При всём при этом в математике рассматриваются не только континуальные бесконечности, но и бесконечности, превосходящие континуум: простейшим примером такой бесконечности может послужить *множество всех множеств действительных чисел* (его мощность обозначают $\aleph_2$). Конечно, никакого конструктивного смысла такие построения не

несут; коль скоро мы ставим себе прикладные задачи, появление в наших рассуждениях континуальной бесконечности должно служить своего рода «тревожной лампочкой»: *осторожно, выход за пределы конструктивного*. Значит ли это, что математика чем-то плоха? Разумеется, нет; просто математика вовсе не обязана быть конструктивной. Математики постоянно проверяют на прочность границы возможностей человеческого мышления, и за одно это им следует сказать огромное спасибо, как, кстати, и за то, что именно математики обеспечивают широкую публику средствами для развития возможностей мозга; ради одного только этого эффекта — развития собственных интеллектуальных возможностей — математику, несомненно, стоит изучать. Просто не всякая математическая модель пригодна для прикладных или, если угодно, инженерных целей; само по себе это не плохо и не хорошо, это просто факт.

Примечательна ситуация с вопросом о том, существуют ли такие множества, которые нельзя занумеровать (то есть несчётные), но которые при этом «меньше», чем континуум; иными словами, есть ли ещё какие-нибудь бесконечности между счётной и континуальной. *Существование таких бесконечностей невозможно ни доказать, ни опровергнуть*, то есть мы вправе считать, что таких множеств не бывает, но точно так же мы вправе считать, что они бывают. В любом случае, ещё никому не удалось и вряд ли когда-нибудь удастся *конструктивно* построить такое множество, то есть описать элементы, из которых оно будет состоять, подобно тому, как мы описывали натуральные числа для счётных множеств и бесконечные десятичные дроби для континуумов.

1.5.5. Алгоритмы и вычислимость

Рассказывая об истории компьютеров, мы отметили (см. стр. 47), что работа компьютера состоит в проведении *вычислений*, хотя результаты этих вычислений в большинстве случаев не имеют ничего общего с числами: любая информация, с которой работает компьютер, должна быть представлена в некоторой объективной форме, и, как следствие, правила, по которым из одной информации получается другая (а именно это и делает компьютер), суть не что иное, как *функции* в сугубо математическом смысле слова: в роли области определения такой функции выступает множество неких «порций информации», представленных избранным объективным способом, и оно же служит областью значений. Обычно считается, что информация — как исходная, так и получаемая — представляется в виде цепочек символов в некотором ***алфавите***[40] A, то есть каждая «порция информации» есть элемент уже знакомого нам по предыдущему параграфу множества A^*. С учётом этого мы можем

[40]Напомним, что под алфавитом можно понимать произвольное конечное множество, как правило, состоящее хотя бы из двух элементов, хотя в некоторых задачах рассматриваются также и алфавиты из одного символа. Пустым алфавит быть не может.

считать, что работа компьютера всегда состоит в *вычислении некой функции вида* $A^* \mapsto A^*$ (выражение $X \mapsto Y$ обозначает функцию с областью определения X и областью значений Y). При этом мы можем легко заметить определённую проблему. Коль скоро мы *можем вычислить* значение функции (в каком бы то ни было смысле), то, очевидно, мы можем как-то записать наши промежуточные вычисления, то есть представить вычисление в виде текста. Больше того, если функция в каком-то смысле *может быть вычислена*, то, видимо, в виде текста можно представить вообще само по себе правило, как эту функцию надо вычислять. Множество всех возможных текстов, как мы уже знаем (см. стр. 139), счётное. Между тем с помощью всё того же канторовского диагонального метода легко убедиться, что **множество всех функций над натуральными числами имеет мощность континуум**; если заменить тексты, то есть элементы множества A^*, их номерами (а это можно сделать в силу счётности множества A^*), получится, что функций вида $A^* \mapsto A^*$ тоже континуум, тогда как множество всех возможных *правил вычисления функции* — не более чем счётное, ведь их можно записать в виде текстов. Следовательно, далеко не для каждой такой функции можно указать правило, по которому она будет вычисляться; говорят, что одни функции ***вычислимы***, а другие — нет.

Даже если рассматривать только функции, областью определения которых является множество натуральных чисел, а областью значений — множество десятичных цифр от 0 до 9, то и такое множество функций окажется континуумом: в самом деле, каждой такой функции можно взаимно однозначно поставить в соответствие бесконечную десятичную дробь с нулевой целой частью, взяв $f(1)$ в качестве первой цифры, $f(2)$ в качестве второй, $f(27)$ в качестве двадцать седьмой и так далее, ну а множество бесконечных десятичных дробей, как мы уже видели, несчётное, то есть континуум. Ясно, что если расширить область значений до всех натуральных, то функций меньше не станет; впрочем, не станет их и «больше» — их так и останется всё тот же континуум. При этом вычислимых функций, напомним, не более чем счётная бесконечность, ведь каждой из них соответствует описание, то есть текст, а множество всех текстов счётное. Получается, что всех «натуральных» функций гораздо «больше», нежели может существовать всех вычислимых, что бы мы ни понимали под вычислимостью — если только мы при этом подразумеваем, что правила конструктивного вычисления можно записать.

Поскольку бесконечных двоичных дробей тоже континуум, мы можем упростить множество рассматриваемых функций ещё больше, оставив лишь два варианта результата: 0 и 1, или «ложь» и «истину». Таких функций, которые, принимая натуральный аргумент, выдают истину или ложь, тоже оказывается континуум, из чего немедленно следует, что множество всех множеств натуральных чисел тоже несчётное (имеет мощность континуум): в самом деле,

ведь каждая такая функция именно что задаёт множество натуральных чисел, и, напротив, каждому множеству натуральных чисел соответствует функция, выдающая истину для элементов множества и ложь для чисел, в множество не входящих. Этот результат нам в дальнейшем изложении не потребуется, но он настолько красив, что не упомянуть его было бы варварством.

Компьютер проводит свои вычисления, подчиняясь ***программе***, которая воплощает собой некую ***конструктивную процедуру***, или ***алгоритм***. Попросту говоря, для того, чтобы компьютер был нам чем-то полезен — а полезен он может быть только созданием одной информации из другой — обязательно нужен кто-то, кто точно знает, как из этой «одной» информации получить «другую», и знает это настолько хорошо, что может заставить компьютер воплотить это знание на практике, причём без непосредственного контроля со стороны обладателя исходного знания; этот человек, собственно говоря, называется программистом. Как несложно догадаться, алгоритм как раз и есть то самое правило, по которому вычисляется функция; можно сказать, что функцию следует считать вычислимой, если для неё существует алгоритм вычисления.

Простота этих рассуждений на самом деле обманчива; понятия ***алгоритма*** и ***вычислимой функции*** оказываются материей крайне заковыристой. Так, практически в любом школьном учебнике информатики вы найдёте определение алгоритма — не пояснение, о чём идёт речь, не рассказ о предмете, а именно определение, короткую фразу вида «алгоритм — это то-то и то-то» или «алгоритмом называется то-то и то-то». Такое определение обычно выделяется крупным жирным шрифтом, обводится рамочкой, снабжается какой-нибудь пиктограммой с восклицательным знаком — словом, делается всё, чтобы убедить как учеников, так и их учителей, что сие подлежит заучиванию наизусть. К сожалению, такие определения годятся только в качестве упражнения на зазубривание. На самом же деле **определения алгоритма не существует**, то есть какое бы «определение» ни приводилось, оно будет заведомо неверным: каждый автор, дающий такое определение, допускает фактическую ошибку уже в тот момент, когда решает начать его формулировать, и совершенно неважно, какова будет итоговая формулировка. Правильного определения не существует не в том смысле, что «определения могут быть разные» и даже не в том, что «мы сейчас не знаем точного определения, но, возможно, когда-нибудь узнаем»; напротив, **мы точно знаем, что определения алгоритма нет и быть не может**, потому что любое такое определение, каково бы оно ни было, выбьет основу из-под целого раздела математики — ***теории вычислимости***.

Чтобы понять, как же так получилось, нам понадобится очередной экскурс в историю. В первой половине XX века математики заинтересовались вопросом, как среди всего теоретического многообразия матема-

тических функций выделить такие, которые человек с использованием каких бы то ни было механических или любых других приспособлений *может вычислить*. Начало этому положил Давид Гильберт, сформулировавший в 1900 году список нерешённых (на тот момент) математических проблем, известных как «Гильбертовы проблемы»; проблема решения произвольного диофантова уравнения, известная как *десятая проблема Гильберта*, позже оказалась *неразрешимой*, но для доказательства этого факта потребовалось создать теорию, формализующую понятие «разрешимости»: без этого невозможно ничего определённого сказать ни про множество задач, которые можно *конструктивно решить*, ни про то, что следует понимать под конструктивностью решения. Проблемами разрешимости задач (иначе говоря, вычислимости функций) занимались такие известные математики, как Курт Гёдель, Стивен Клини, Алонсо Чёрч и Алан Тьюринг.

Функции, оперирующие с иррациональными числами, пришлось отбросить сразу же. Иррациональных оказалось «слишком много»; в предыдущем параграфе мы дали некоторые пояснения, почему континуальные бесконечности не годятся для конструктивной работы.

Коль скоро континуальные бесконечности конструктивным вычислениям не поддаются, ограничивая нас счётными множествами, Гёдель и Клини предложили для теоретических изысканий рассматривать только функции натуральных аргументов (возможно, нескольких), значениями которых тоже являются натуральные числа; при необходимости любые функции, работающие над произвольными счётными множествами (в том числе, что важно для нас, и над множеством A^*), могут быть сведены к таким «натуральным» функциям путём замены элементов множеств их номерами.

Здравый смысл подсказывает, что даже такие функции не всегда *вычислимы*; отсылка к здравому смыслу здесь требуется, поскольку мы пока не поняли (а на самом деле — так и не поймём), что же такое «вычислимая функция». Тем не менее, как уже было сказано, функций вида $\mathbb{N} \mapsto \mathbb{N}$ (где $\mathbb{N}$ — множество натуральных чисел) — континуум, тогда как алгоритмов, судя по всему, не более чем счётное множество; общее количество возможных функций, даже когда мы рассматриваем «всего лишь» натуральные функции натурального аргумента, оказывается «гораздо больше», чем количество функций, которые можно как-то вычислять.

Изучая вычислимость функций, Гёдель, Клини, Аккерман и другие математики пришли к классу так называемых ***частично-рекурсивных функций***. В качестве определения этого класса рассматривается некий базовый набор очень простых исходных функций (константа, увеличение на единицу и *проекция* — функция нескольких аргументов, значением которой является один из её аргументов) и *операторов*, то есть операций над функциями, позволяющих строить новые

функции (операторы композиции, примитивной рекурсии и минимизации); под частично-рекурсивной функцией понимается любая функция, которую можно построить с помощью перечисленных операторов из перечисленных исходных функций. Слово «частичные» в названии класса указывает на то, что в этот класс обязательно[41] входят функции, которые определены лишь на некотором множестве чисел, а для чисел, не входящих в это множество — не определены, то есть не могут быть вычислены. Следует отметить, что эпитет «рекурсивные» в данном контексте означает, что функции выражаются одна через другую — возможно, даже через саму себя, но не обязательно. Как мы увидим позже, в программировании смысл термина «рекурсия» несколько у́же.

Многочисленные попытки расширить множество вычислимых функций путём введения новых операций успехом не увенчались: каждый раз удавалось доказать, что класс функций, задаваемых новыми наборами операций, оказывается всё тем же — уже известным классом частично-рекурсивных функций, а все новые операции благополучно (хотя в некоторых случаях довольно сложно) выражаются через уже имеющиеся.

Алонсо Чёрч отказался от дальнейших попыток расширить этот класс и заявил, что, по-видимому, как раз частично-рекурсивные функции соответствуют понятию *вычислимой функции* в любом разумном понимании вычислимости. Это утверждение называют ***тезисом Чёрча***. Отметим, что тезис Чёрча не может рассматриваться как теорема — его нельзя доказать, поскольку у нас нет определения вычислимой функции и тем более нет определения «разумного понимания». Но почему бы, спросите вы, не дать какое-нибудь определение, чтобы тезис Чёрча оказался доказуем? Ответ здесь очень простой. **Превратив тезис Чёрча в якобы доказанный факт, мы совершенно безосновательно лишили бы себя перспектив дальнейшего исследования вычислимости и разнообразных механизмов вычисления**.

Пока что все попытки создания набора конструктивных операций, более богатого, нежели предложенный ранее, потерпели неудачу: каждый раз оказывается, что класс функций получается ровно тот же. Вполне возможно, что так будет всегда, то есть класс вычислимых функций никогда не будет расширен; именно это и утверждает тезис Чёрча. Но ведь это не может быть доказано — хотя бы потому, что не вполне понятно, что же такое «конструктивная операция» и каково их множество. Следовательно, всегда остаётся возможность, что в будущем кто-нибудь предложит такой набор операций, который окажется мощнее, нежели базис для частично-рекурсивных функций. Тезис Чёрча в этом случае будет опровергнут, или, точнее говоря, вместо него появится новый тезис, аналогичный имеющемуся, но ссылающийся на другой

[41] Почему это столь обязательно, мы узнаем чуть позже, см. рассуждение на стр. 156.

класс функций. Подчеркнём, что определение вычислимой функции от этого не появится, поскольку даже если класс вычислимых функций окажется расширен, само по себе это не может означать, что его нельзя расширить ещё больше.

С некоторой натяжкой можно считать, что класс частично-рекурсивных функций со всеми его свойствами представляет собой некую *абстрактную математическую теорию* наподобие геометрии Евклида или, скажем, теории вероятности, тогда как понятие вычислимости как таковое находится вне математики, являясь свойством нашей Вселенной («реального мира») наряду со скоростью света, законом всемирного тяготения и тому подобным. Тезис Чёрча в этом случае оказывается своего рода *научной гипотезой* относительно того, как устроен реальный мир; всё окончательно встаёт на свои места, если мы вспомним, что согласно теории научного знания, сформулированной Карлом Поппером, гипотезы не бывают верными, а бывают только *неопровергнутыми*, и исследователь обязан принимать во внимание, что любая гипотеза, сколько бы подтверждений ей не нашлось, может оказаться опровергнута в будущем. Тезис Чёрча утверждает, что любая функция, которая может быть *конструктивно вычислена*, находится в классе частично-рекурсивных функций; это утверждение пока никому не удалось опровергнуть, в связи с чем мы принимаем его за верное. Заметим, *критерий фальсификации Поппера* к тезису Чёрча прекрасно применим. В самом деле, мы *можем* (и достаточно легко) указать такой эксперимент, положительный результат которого опроверг бы тезис Чёрча: достаточно построить некий *конструктивный автомат*, который вычислял бы функцию, не входящую в класс частично-рекурсивных.

Формальная теория алгоритмов построена во многом аналогично теории вычислимости. Считается, что алгоритм есть конструктивная реализация некоего преобразования из входного слова в результирующее, причём как входное слово, так и слово-результат представляют собой конечные цепочки символов в некотором алфавите. Иначе говоря, чтобы можно было говорить об алгоритме, необходимо для начала зафиксировать какой-нибудь алфавит A, и тогда алгоритмы окажутся конструктивными реализациями всё тех же знакомых нам преобразований вида $A^* \mapsto A^*$, то есть, попросту говоря, реализациями (если угодно, конструктивными правилами вычислений) *функций* одного аргумента, для которых аргументом является слово из символов алфавита, и слово же является результатом вычисления. Отметим, что всё сказанное никоим образом не может считаться определением алгоритма, поскольку опирается на такие выражения, как «конструктивная реализация», «конструктивные правила вычислений», а сами эти «термины» остаются без определений. Продолжая аналогию, мы отмечаем, что **не всякое** такое преобразование может быть реализовано алгоритмом, ведь таких преобразований континуум, тогда как алгоритмов, естественно, не более чем счётное множество — ведь что бы мы ни понимали под алгоритмом, мы, во всяком случае, подразумеваем, что его можно как-то *записать*, то есть представить в виде конечного текста, а множество всех воз-

можных текстов счётное. Кроме того, отметим, что было бы не вполне правильно отождествлять алгоритм с тем преобразованием, которое он выполняет, поскольку два *разных* алгоритма могут выполнять одно и то же преобразование; к этому вопросу мы вскоре вернёмся.

Один из основателей теории алгоритмов Алан Тьюринг предложил формальную модель автомата, известную как ***машина Тьюринга***. Этот автомат имеет ленту, бесконечную в обе стороны, в каждой ячейке которой может быть записан символ алфавита либо ячейка может быть пустой. Вдоль ленты движется головка, которая может находиться в одном из нескольких предопределённых ***состояний***, причём одно из этих состояний считается начальным (в нём головка находится в начале работы), а другое — заключительным (при переходе в него машина завершает работу). В зависимости от текущего состояния и от символа в текущей ячейке машина может:

- записать в текущую ячейку любой символ алфавита вместо того, который там записан сейчас, в том числе и тот же самый символ, то есть оставить содержимое ячейки без изменений;
- изменить состояние головки на любое другое, в том числе остаться в том состоянии, в котором головка находилась до этого;
- сдвинуться на одну позицию вправо, на одну позицию влево или остаться в текущей позиции.

Программа для машины Тьюринга, чаще называемая просто «машиной Тьюринга», представляется в виде таблицы, задающей, что́ должна сделать машина при каждой комбинации текущего символа и текущего состояния; по горизонтали отмечают символы, по вертикали — состояния (или наоборот), а в каждой ячейке таблицы записывают три значения: новый символ, новое состояние и следующее движение (влево, вправо или остаться на месте). Перед началом работы на ленте записано входное слово; если после какого-то числа шагов машина перешла в заключительное состояние, считается, что слово, которое записано теперь на ленте, является результатом работы.

Тезис Тьюринга гласит: каково бы ни было разумное понимание алгоритма, любой алгоритм, соответствующий такому пониманию, можно реализовать в виде машины Тьюринга. Этот тезис подтверждается тем, что множество предпринятых попыток создать «более мощный» автомат потерпели неудачу: для каждого создаваемого формализма (формального автомата) удаётся указать, каким способом построить аналогичную ему машину Тьюринга. Самих таких формализмов было построено немало: это и нормальные алгоритмы Маркова, и лямбда-исчисление Чёрча, и всевозможные автоматы с регистрами, причём каждый такой «алгоритмический формализм», будучи рассмотренным в роли конкретно определимой рабочей модели вместо «неуловимого» понятия алгоритма, оказывается в том или ином виде полезен для развития теории, а в ряде случаев — и для практического применения;

существуют, в частности, языки программирования, основанные на лямбда-исчислении, а также языки, вычислительная модель которых напоминает алгоритмы Маркова. При этом для каждого такого формализма было доказано, что его можно реализовать на машине Тьюринга, а машину Тьюринга — на нём.

Тем не менее доказать тезис Тьюринга не представляется возможным, поскольку невозможно определить, что такое «разумное понимание алгоритма»; это не исключает теоретической возможности, что когда-либо в будущем тезис Тьюринга окажется опровергнут: для этого достаточно предложить некий формальный автомат, соответствующий нашему пониманию алгоритма (то есть конструктивно реализуемый), но при этом имеющий конфигурации, которые в машину Тьюринга не переводятся. То, что **пока** никому не удалось предложить такой автомат, формально ничего не доказывает: а вдруг кому-то повезёт больше?

Всё это весьма напоминает ситуацию с вычислимыми функциями, частично-рекурсивными функциями и тезисом Чёрча, и такое подобие не случайно. Как мы уже отмечали, все преобразования вида $A^* \mapsto A^*$ путём замены элемента множества A^* (то есть сло́ва) его номером (что можно сделать, поскольку множество A^* счётное) могут быть превращены в преобразования вида $\mathbb{N} \mapsto \mathbb{N}$, а путём замены номера слова самим словом — обратно. Более того, доказано, что любое преобразование, реализованное машиной Тьюринга, может быть задано в виде частично-рекурсивной функции, а любая частично-рекурсивная функция — реализована в виде машины Тьюринга.

Означает ли это, что «вычислимая функция» и «алгоритм» суть одно и то же? Формально говоря, нет, и тому есть две причины. Во-первых, оба понятия не определены, так что доказать их эквивалентность невозможно, как, впрочем, и опровергнуть её. Во-вторых, как уже говорилось, эти понятия несколько отличаются по своему наполнению: если две *функции*, записанные различным образом, имеют при этом одну и ту же область определения и всегда дают одинаковые значения при одинаковых аргументах, обычно считается, что речь идёт о двух записях *одной и той же функции*, тогда как в применении к двум алгоритмам говорят об *эквивалентности* двух *различных* алгоритмов.

Прекрасным примером таких алгоритмов будет решение известной задачи о ханойских башнях. В задаче фигурируют три стержня, на один из которых надето N плоских дисков, различающихся по размеру, в виде своеобразной пирамидки (внизу самый большой диск, сверху самый маленький). За один ход мы можем переместить один диск с одного стержня на другой, при этом если стержень пуст, на него можно поместить любой диск, но если на стержне уже имеются какие-то диски, то можно поместить только меньший диск сверху на больший, но не наоборот. Брать сразу несколько дисков нельзя, то есть перемещаем мы только один диск за ход. Задача состоит в том, чтобы за наименьшее возможное число ходов переместить все диски с одного стержня на другой, пользуясь третьим в качестве промежуточного.

Хорошо известно, что задача решается за $2^N - 1$ ходов, где N — количество дисков, и, больше того, хорошо известен рекурсивный[42] алгоритм решения, изложенный, например, в книге Я. Перельмана «Живая математика» [5], которая вышла в 1934 году. Базисом рекурсии может служить перенос одного диска за один ход с исходного стержня на целевой, однако ещё проще в качестве базиса использовать вырожденный случай, когда задача уже решена и ничего никуда переносить не надо, то есть количество дисков, которые надо перенести, равно нулю. Если нужно переместить N дисков, то мы пользуемся своим же собственным алгоритмом (то есть *рекурсивно* обращаемся к нему), чтобы сначала перенести $N-1$ дисков с исходного стержня на промежуточный, потом переносим самый большой диск с исходного стержня на целевой, и снова обращаемся к самому себе, чтобы перенести $N - 1$ дисков с промежуточного стержня на целевой.

Чтобы воплотить этот алгоритм в виде программы, нужно придумать некие правила записи ходов. С этим всё довольно просто: поскольку переносится всего один диск, ход представляется в виде пары номеров стержней: с какого и на какой мы собираемся переносить очередной диск. Исходными данными для нашей программы послужит количество дисков. Тексты этой программы на Паскале и на Си мы напишем позже, когда изучим эти языки в достаточной степени; нетерпеливому читателю можем предложить посмотреть § 2.14.2, где приведено решение на Паскале, ну а за решением на Си придётся обратиться ко второму тому нашей книги.

Отметим, забегая вперёд, что рекурсивная подпрограмма, осуществляющая собственно решение задачи, на обоих языках займёт по восемь строк, включая заголовок и операторные скобки, а всё остальное, что придётся написать — это вспомогательные действия, проверяющие корректность входных данных и переводящие их из текстового представления в числовое.

Рассмотрим теперь другое решение той же задачи, обходящееся без рекурсии. Перельман этого решения не привёл, так что его никто в итоге не знает[43]; при этом на русском языке решение описывается гораздо проще, чем рекурсивное. Итак, на *нечётных* ходах (на первом, третьем, пятом и т. д.) самый маленький диск (то есть диск № 1) перемещается «по кругу»: с первого стержня на второй, со второго на третий, с третьего на первый и так далее, либо, наоборот, со первого на третий, с третьего на второй, со второго на первый и так далее. Выбор «направления» этого перемещения зависит от общего количества дисков: если оно чётное, идём в «естественном» направлении, то есть $1 \to 2 \to 3 \to 1 \to \ldots$, если же оно нечётное, идём «обратным» кругом: $1 \to 3 \to 2 \to 1 \to \ldots$. Перенос диска на чётных ходах определяется однозначно тем, что мы не должны при этом трогать самый маленький диск, а сделать ход, не трогая его, можно только одним способом, то есть мы попросту смотрим на те два стержня, на которых

[42] Когда речь идёт об алгоритмах или фрагментах программ, под ***рекурсией*** понимают использование таким алгоритмом (или таким фрагментом программы) *самого себя* для решения более простого случая задачи.

[43] Автор не претендует на лавры изобретателя этого варианта алгоритма, поскольку точно помнит, что ему это решение в студенческие годы рассказал кто-то из старшекурсников, занимавшихся на одном с ним спецсеминаре, но кто это был, за давностью лет вспомнить несколько затруднительно.

нет самого маленького диска, и делаем единственный возможный ход между ними.

Как ни странно, компьютерная программа, воплощающая этот вариант решения головоломки, оказывается значительно (более чем в десять раз!) сложнее, чем вышеприведённая для рекурсивного случая. Дело тут в формальной реализации фразы «смотрим на два стержня и делаем единственный ход». Человек, решая головоломку, действительно посмотрит на стержни, сравнит, какой из верхних дисков меньше, и перенесёт этот диск. Чтобы сделать то же самое в программе, нам придётся запоминать, какие диски на каком стержне в настоящий момент присутствуют, что не слишком сложно, если уметь работать с односвязными списками, но всё же достаточно сложно, чтобы не вставлять текст этой программы в книгу — во всяком случае, целиком. Рекурсивное решение оказалось таким простым, потому что мы не помнили, какие диски где присутствуют, мы и без этого знали, какие ходы надо сделать.

Что можно сказать точно, так это то, что *при тех же входных данных эта программа напечатает такие же ходы, как и предыдущая (рекурсивная)*, хотя сама программа, очевидно, будет написана совершенно иначе. Терминологическая проблема, встающая в этот момент, следующая. Очевидно, что речь идёт о двух разных *программах* (такие программы называются эквивалентными). Но идёт ли речь о двух разных *алгоритмах* или об одном и том же, записанном разными словами? В большинстве случаев принимается «очевидный» ответ, что это два разных алгоритма, пусть и эквивалентных, то есть реализующих одну и ту же *функцию* (естественно, вычислимую).

Тем не менее, обычно считается, что в теории алгоритмов и в теории вычислимых функций речь идёт более-менее об одном и том же. Здравый смысл подсказывает, что это правильно: и то, и другое представляет собой некое «конструктивное» преобразование из некоторого счётного множества в него же само, только множества рассматриваются разные: в одном случае это множество цепочек над неким алфавитом, в другом случае — просто множество натуральных чисел. Нумерация цепочек позволяет легко переходить от одного к другому, не нарушая «конструктивности», чем бы она ни была; следовательно, мы имеем дело с одной и той же сущностью, только выраженной в разных терминах. Различать ли преобразования только с точностью до их «внешних проявлений», как в теории вычислимости, или же с точностью до их «конкретного воплощения», как в теории алгоритмов — это, в конце концов, вопрос традиций. Более того, у многих авторов встречается понятие «тезиса Чёрча-Тьюринга», предполагающее, что эти тезисы не следует разделять: речь ведь идёт об одном и том же, только в разных терминах.

После всего сказанного любые *определения алгоритма* вызывают в лучшем случае недоумение, ведь дав такое определение, мы тем самым отправим на свалку теорию вычислимости и теорию алгоритмов, тезис Чёрча вместе с тезисом Тьюринга, множественные попытки построения чего-то более сложного — например, машины Тьюринга с

несколькими лентами, доказательства их эквивалентности исходной машине Тьюринга, работу сотен, возможно даже тысяч исследователей — а определение при этом всё равно окажется либо неправильным, либо настолько расплывчатым, что его невозможно будет использовать на практике.

Значит ли это, что понятие алгоритма вообще нельзя использовать из-за его неопределённости? Конечно же, нет. Во-первых, если говорить о математике, то мы вообще довольно часто используем понятия, не имеющие определений: например, нельзя дать определения точки, прямой и плоскости, но этот факт никоим образом не отменяет геометрию. Во-вторых, если говорить о сфере инженерно-технических знаний, то строгие определения здесь вообще встречаются очень редко, и это никому не мешает. Наконец, следует принять во внимание тезисы Чёрча и Тьюринга: *с учётом этих тезисов* понятие алгоритма обретает вполне строгое математическое наполнение, нужно только помнить, откуда это наполнение взялось, то есть не забывать о роли тезисов Чёрча и Тьюринга во всей нашей теории.

1.5.6. Алгоритм и его свойства

Несмотря на то, что само понятие алгоритма не может быть определено, наше интуитивное понимание алгоритма позволяет говорить о некоторых свойствах, выполняющихся для любого объекта, который нам может прийти в голову считать алгоритмом. Кроме того, можно говорить о неких *характеристиках* алгоритмов, то есть о свойствах, которые могут выполняться для одних алгоритмов и не выполняться для других.

Одно из базовых свойств любого алгоритма мы уже неоднократно использовали в наших рассуждениях, постулируя, что алгоритм *можно записать в виде текста*, притом, естественно, *конечного*: бесконечные тексты мы в действительности записать не можем сугубо технически, так что, коль скоро алгоритмы имеют отношение к конструктивной деятельности, бесконечными они быть не могут. Представимость любого алгоритма в виде текста, и притом конечного, можно назвать свойствами ***объективности*** и ***конечности алгоритма***.

Ещё одно достаточно очевидное свойство любого алгоритма — его ***дискретность***: каково бы ни было правило, согласно которому из исходной информации получают результирующую, и каков бы ни был исполнитель этого правила, само по себе *исполнение* всегда представляет собой некий *дискретный процесс*, который при ближайшем рассмотрении распадается на какие-то *элементарные шаги/действия*. Понимать дискретность можно также и в том смысле, что любая информация, над которой работает алгоритм, может быть представлена в виде текста, что означает, в частности, что с целыми и рациональными числами

мы работать можем, а с иррациональными у нас возникнут определённые сложности. Впрочем, коль скоро для иррационального числа существует *обозначение*, алгоритм вполне может оперировать таким числом — точнее, не им самим, а его обозначением; так, мы вполне можем придумать такой алгоритм для решения квадратных уравнений, который при невозможности извлечения квадратного корня из дискриминанта выдаст в качестве результата выражение, содержащее этот корень: например, получив на вход уравнение $x^2 - 5x + 6 = 0$, такой алгоритм в качестве результата выдаст числа 2 и 3, тогда как для уравнения $x^2 - x - 1 = 0$ он, используя то или иное текстовое обозначение квадратного корня, выдаст в качестве ответа выражения $\frac{1-\sqrt{5}}{2}$ и $\frac{1+\sqrt{5}}{2}$. Важно понимать, что такой алгоритм *ни в какой момент* не будет оперировать *численным значением* $\sqrt{5}$, поскольку таковое не имеет *дискретного представления*.

Заметим, что число $\sqrt{5}$ имеет довольно простое «аналоговое» представление: достаточно зафиксировать некий эталон единичной длины, например, сантиметра, и взять прямоугольник со сторонами 1 и 2, и его диагональ как раз будет представлять корень из пяти. Иначе говоря, мы можем построить (если угодно, с помощью циркуля и линейки) отрезок, который будет ровно в $\sqrt{5}$ раз длиннее заданного. Если немного подумать, можно, используя исходно целые величины, получить $\sqrt{5}$ в виде силы, действующей на какое-нибудь тело, в виде объёма жидкости в сосуде и тому подобных *аналоговых физических величин*. Так вот, **алгоритмы ни с чем подобным не работают**; как мы уже говорили, вся теория вычислимости построена исключительно на целых числах, и целыми же (в конечном счёте) числами оперируют компьютеры.

В принципе, известны так называемые ***аналоговые ЭВМ***, работающие как раз с непрерывными физическими процессами; исходные, промежуточные и результирующие данные в таких машинах представляются значениями электрических величин, чаще всего напряжения; но функционирование аналоговых ЭВМ не имеет ничего общего с алгоритмами.

Третье фундаментальное свойство, присущее любому алгоритму, не столь очевидно; оно называется ***детерминированностью*** и состоит в том, что алгоритм не оставляет исполнителю никакой «свободы выбора»: следовать предписанной процедуре можно одним и только одним способом. Единственное, что может повлиять на ход выполнения алгоритма — это исходные данные; в частности, при одних и тех же исходных данных алгоритм всегда выдаёт одни и те же результаты.

Читатели, уже сталкивавшиеся с практическим программированием, могут заметить, что программы, в особенности игровые (а также, например, криптографические), часто отходят от этого правила, используя датчики случайных чисел. На самом деле это никоим образом не противоречит теории алгоритмов, просто случайные числа, откуда бы они ни брались, следует рассматривать как составную часть исходных данных.

Свойства объективности/конечности (в том смысле, что всякий алгоритм имеет конечное объективное представление), дискретности и

детерминированности присущи всем алгоритмам без исключения, то есть если нечто не обладает каким-либо из этих свойств, то перед нами заведомо никакой не алгоритм. Конечно, эти свойства следует рассматривать скорее как некие *аксиомы*, то есть утверждения, которые приняты, невзирая на их недоказуемость: просто принято считать, что алгоритмы, что́ бы под ними ни понималось, всегда таковы. С некоторой натяжкой можно утверждать, что алгоритм должен ещё обладать *понятностью для исполнителя*, хотя это уже на грани фола: *понятность* относится скорее не к алгоритму, а к его записи, но ведь алгоритм и его запись — это совсем не одно и то же; больше того, один алгоритм может иметь бесконечно много представлений даже в одной и той же системе записи: например, если в программе переименовать все переменные или поменять местами подпрограммы, алгоритм от этого не изменится.

Наряду с перечисленными *обязательными* свойствами алгоритм может обладать (но может и не обладать) некоторыми *частными* свойствами, такими как массовость, завершаемость (применимость к отдельным входным словам или ко всем возможным входным словам), правильность и полезность (что бы под ними ни понималось) и т. д. Возможно, эти свойства *желательны*, но не более того; это характеристики, которым отдельно взятый алгоритм может соответствовать, а может и не соответствовать, а во многих случаях такое соответствие даже нельзя (невозможно!) проверить. К сожалению, существует много разнообразных литературных источников сомнительного качества (в число которых, как ни прискорбно, входят некоторые школьные учебники информатики), где обязательные свойства алгоритмов сваливают в одну кучу с частными, порождая на выходе совершенно феерический кавардак.

Начнём со свойства ***массовости*** алгоритма, как самого простого; это свойство обычно понимается в том смысле, что алгоритм должен уметь решать некое семейство задач, а не одну задачу; именно для этого алгоритм делают зависимым от входных данных. Свойство массовости, очевидно, можно проверить, то есть, рассматривая конкретный алгоритм, легко определить, обладает он массовостью или нет; но на этом, собственно, и всё. Обязательным оно никоим образом не является, то есть алгоритм, который вообще не зависит от входных слов, алгоритмом быть не перестаёт. Как среди вычислимых функций присутствуют *константы*, так и среди алгоритмов присутствуют *генераторы единственного результата*. Между прочим, именно к этой категории относится знаменитая программа «Hello, world» — излюбленный многими авторами пример первой программы на очередном языке программирования; всё, что она делает — это выдаёт фразу «Hello, world!»[44] и завершается. Кстати, мы тоже начнём изучение Паскаля, а затем и Си именно с этой программы. Очевидно, программа, всегда печатающая

[44] Здравствуй, мир! (*англ.*)

одну и ту же фразу, вообще никак не зависит ни от каких входных данных и, следовательно, никакой массовостью не обладает. Если бы мы считали массовость обязательным свойством алгоритма, пришлось бы считать, что программы, подобные этой, не реализуют никаких алгоритмов.

Если, скажем, рассматривать обыкновенные треугольники на плоскости, то можно заметить, что всякий треугольник подчиняется неравенству треугольника (то есть сумма длин любых двух сторон заведомо больше длины третьей стороны), а ещё его сумма углов всегда равна 180°; это свойства всех треугольников без исключения. Кроме того, среди всех треугольников есть ещё *прямоугольные* треугольники, для которых выполняется теорема Пифагора; разумеется, теорема Пифагора — очень важное и нужное свойство, но было бы нелепо требовать её выполнения для *всех* треугольников. Именно так обстоят дела и с массовостью алгоритмов.

Когда в рассуждениях о свойствах алгоритмов не проводится явной границы между свойствами обязательными, то есть присущими всем алгоритмам без исключения, и свойствами частными, такими, которыми алгоритм может обладать или не обладать, результатом таких рассуждений становятся грубейшие фактические ошибки; сейчас мы постараемся обсудить наиболее популярные из них. Такое обсуждение позволит нам, во-первых, не повторять ошибок, чья популярность никоим образом не отменяет их грубости, и, во-вторых, по ходу обсуждения мы узнаем ещё несколько интересных аспектов теории алгоритмов.

Так, на удивление часто можно встретить утверждение, что *любой алгоритм должен обладать свойствами «правильности» и «завершаемости»*, то есть, иначе говоря, любой алгоритм якобы должен всегда завершаться за конечное число шагов, притом не просто завершаться, а выдавать при этом правильный результат. Конечно, *хотелось бы*, чтобы все алгоритмы (и, стало быть, все компьютерные программы) действительно никогда не зависали и не делали никаких ошибок; как говорится, хотеть не вредно. В реальности дела обстоят совершенно иначе: такое «райски-идеальное» положение вещей оказывается принципиально недостижимо, причём не только технически, но и, как мы вскоре увидим, сугубо математически. С таким же успехом можно было бы хотеть, чтобы все автомобили были оснащены вечными двигателями, а число π стало равно трём, чтобы легче было считать.

Начнём с «правильности». Очевидно, что это понятие попросту невозможно формализовать; более того, зачастую при разработке алгоритма вообще невозможно указать какой бы то ни было критерий проверки «правильности», или, что ещё хуже, разные люди могут оперировать разными критериями при оценивании одного и того же алгоритма. Это свойство предметной области хорошо известно профессиональным программистам: одно и то же поведение компьютерной программы может её автору казаться правильным, а заказчику — не просто неправильным, но даже в некоторых случаях возмутительным. Никакие формальные опи-

сания, никакое сколь угодно подробное тестирование не способны эту ситуацию исправить. Среди программистов небезосновательно считается, что «правильных» программ не бывает, бывают лишь такие программы, в которых пока не найдено ошибок — но это не значит, что ошибок там нет. Некоторое время назад среди программистов-исследователей была популярна тема формальной верификации программ, доказательного программирования и т. п.; никаких обнадёживающих результатов в этой области никто так и не достиг, и популярность направления пошла на убыль. Получается, что **если считать правильность обязательным свойством любого алгоритма, из этого будет следовать, что алгоритмов не существует** — во всяком случае, их нельзя «предъявить», ведь проверить свойство «правильности» технически невозможно.

Со свойством «завершаемости» дела обстоят ещё интереснее. В теории алгоритмов имеется термин «***применимость***»: алгоритм называется *применимым к данному входному слову*, если, имея это слово на входе, алгоритм завершается за конечное число шагов. А теперь самое интересное: **проблема применимости алгоритма к входному слову является алгоритмически неразрешимой**, то есть невозможно — математически! — построить такой алгоритм, который, получив на вход запись другого алгоритма и некоторое входное слово, определил бы, применим данный алгоритм к данному слову или нет. Проблема применимости, известная также под названием ***проблема останова***, является хрестоматийным примером алгоритмически неразрешимой проблемы.

Неразрешимость проблемы применимости можно доказать, как говорится, в три строчки, если привлечь её частный случай — проблему самоприменимости, то есть вопрос о том, остановится ли данный алгоритм, если на вход ему подать его собственную запись. В самом деле, допустим, такой алгоритм есть; назовём его S. Напишем алгоритм S' следующего вида[45]:

$$S'(X) = \text{если } S(X), \text{ то БЕСКОНЕЧНЫЙ_ЦИКЛ, иначе ВЫХОД}$$

Ясно, что алгоритм S' может быть либо самоприменим, либо не самоприменим. Рассмотрим его применение к самому себе, то есть $S'(S')$. Предположим, S' самоприменим; тогда результатом $S(S')$ будет истина, и, следовательно, $S'(S')$ в соответствии с собственным определением уйдёт в бесконечный цикл, то есть окажется не применим к самому себе, что противоречит предположению. Предположим теперь, что алгоритм S' не является самоприменимым. Тогда результатом $S(S')$ будет ложь, так что $S'(S')$ благополучно завершится, то есть окажется самоприменим, что, опять-таки, противоречит предположению. Таким

[45] Здесь мы проводим рассуждения по упрощённой схеме — в частности, не делаем различия между алгоритмом и его записью; кроме того, мы говорим об «алгоритме вообще», тогда как с формальной точки зрения следовало бы использовать один из строго определённых формализмов, хотя бы ту же самую машину Тьюринга. Всё это несложно исправить, но лаконичность изложения от этого сильно пострадает, а наша задача сейчас — не в том, чтобы дать формальное доказательство, а в том, чтобы объяснить, почему дела обстоят так, а не иначе.

образом, алгоритм S' не является ни самоприменимым, ни несамоприменимым; следовательно, его попросту не существует. Как следствие, не существует и алгоритма S, иначе S' можно было бы написать (вне зависимости от того, какой из алгоритмических формализмов мы используем — и ветвление, и бесконечный цикл реализуются везде).

Вернёмся к свойству «завершаемости». Обычно при его упоминании никакой речи о «входных словах» не идёт, то есть авторы, приписывающие алгоритму такое свойство, имеют в виду, по всей видимости, что алгоритм должен быть, следуя классической терминологии, *применим к любому входному слову*. Между тем, на самом деле невозможно (в общем случае) проверить даже применимость алгоритма к одному отдельно взятому входному слову, не говоря уже обо всём бесконечном множестве таких слов. Конечно, можно указать тривиальные случаи таких алгоритмов: например, алгоритм, всегда выдающий одно и то же значение вне зависимости от входного слова, будет заведомо применим к любому входному слову. Однако не только «в общем случае», но и во всех мало-мальски «интересных» случаях, к каковым относятся практически любые полезные компьютерные программы, мы не можем сказать ничего определённого на тему «завершаемости»: алгоритмическая неразрешимость — штука упрямая. Если считать «завершаемость» априорным свойством *любого* алгоритма (то есть считать, что нечто, не обладающее этим свойством, алгоритмом не является), то мы вообще не сможем привести примеры алгоритмов, за исключением самых тривиальных, и уж точно упустим из рассмотрения бóльшую и наиболее интересную часть алгоритмов. Иначе говоря, если включить «завершаемость» в понятие алгоритма, как это, увы, часто делают авторы учебников, мы с таким понятием «алгоритма» в подавляющем большинстве случаев не смогли бы вообще определённо сказать, алгоритм перед нами или нет, то есть попросту не смогли бы отличать алгоритмы от не-алгоритмов; зачем нам, спрашивается, тогда такое определение?

Ситуация начинает выглядеть ещё пикантнее, если учесть, что в теории вычислимых функций *возможная неопределённость функции на подмножестве области её определения* играет чрезвычайно важную роль, причём любые попытки доопределить функции вроде «рассмотрим другую функцию, которая равна нулю во всех точках, в которых исходная функция не определена» терпят сокрушительное фиаско.

Чтобы понять, почему это так, рассмотрим следующее довольно простое рассуждение. Поскольку мы договорились, что любые конструктивные вычисления можно записать в виде текста, из этого следует, что вычислимых функций — счётное множество, то есть существует некоторая нумерация всех вычислимых функций. Пусть теперь мы пытаемся ввести такое понятие вычислимой функции одного аргумента, что всякая функция, вычислимая в этом смысле, оказывается при этом определена на всём множестве натуральных аргументов. Ясно, что, поскольку речь идёт о вычислимых функциях, этих функций тоже не более чем счётное множество. Обозначим их все через $f_1, f_2, f_3, \ldots, f_n, \ldots$; иначе говоря,

пусть нумерованная последовательность f_n исчерпывает множество вычислимых (в нашем смысле) функций. Рассмотрим теперь функцию $g(n) = f_n(n) + 1$. Ясно, что в любом разумном смысле такая функция вычислима, но в то же время она отличается от каждой из f_n, то есть получается, что она в множество вычислимых функций не входит[46].

Как следствие, мы должны либо согласиться с тем, что прибавление единицы может сделать из вычислимой функции невычислимую, либо принять как данность, что при рассмотрении вычислимых функций (что бы мы под таковыми ни понимали) ограничиваться только всюду определёнными функциями нельзя. Заметим, если функции семейства f_n не обязаны быть всюду определёнными, то никакого противоречия у нас не возникает; в самом деле, если функция f_k не определена в точке k, то $g(n)$, определённая вышеуказанным способом, оказывается тоже не определена в точке k, и, как следствие, её отличие от каждой из f_k более не гарантируется её определением; иначе говоря, $g(n)$ может совпадать с любой из тех функций f_k, которые не определены в точке, соответствующей их номеру, и, соответственно, «получает право» не выходить за пределы множества $\{f_k\}$.

Интересно, что в теории вычислимых функций рассматривается некий «упрощённый» класс функций — так называемые ***примитивно-рекурсивные функции***. От частично-рекурсивных функций (см. стр. 144) их отличает отсутствие оператора минимизации, то есть функции строятся на основе тех же примитивных функций (константы, увеличения на единицу и проекции) и операторов суперпозиции и примитивной рекурсии. Все функции, построенные таким образом, заведомо определены для любых значений аргументов, «зацикливаться» им попросту негде; в классе частично-рекурсивных функций возможность «неопределённости» вносит как раз оператор минимизации.

Казалось бы, класс этот достаточно широк, причём, если говорить об арифметике целых чисел, складывается (обманчивое) впечатление, что он покрывает «почти всё». Чтобы привести пример частично-рекурсивной функции, которая не была бы при этом примитивно-рекурсивной, Вильгельму Аккерману пришлось специально придумать функцию, которую потом так и назвали функцией Аккермана; эта функция двух аргументов растёт настолько стремительно, что все её осмысленные значения умещаются в небольшую табличку, за пределами которой эти числа превышают количество атомов во Вселенной.

Если, однако, из области целочисленных функций вернуться в привычный нам мир алгоритмов, выяснится, что, когда речь идёт о нумерации входных и выходных цепочек, количество атомов во Вселенной окажется не таким уж и большим числом, примером же функции, которая, будучи частично-рекурсивной, не является примитивно-рекурсивной, «внезапно» оказывается любая компьютерная программа, использующая «неарифметические» циклы, то есть, по сути, обыкновенные циклы `while`, для которых на момент входа в цикл нельзя (иначе как пройдя цикл) определить, сколько будет у этого цикла итераций. Точно так

[46] Читателю, заинтересовавшемуся вычислимыми функциями, мы можем порекомендовать для начала книгу В. Босса «От Диофанта до Тьюринга» [6], содержащую довольно удачный популярный обзор соответствующей математической теории. Существуют, разумеется, и более специальные пособия. Более подробное изложение теории алгоритмов и вычислимости мы оставим за рамками нашей книги, чтобы не пытаться объять необъятное.

же оказывается «непримитивной» программа, использующая рекурсию, для которой при входе нельзя заранее узнать, сколько получится уровней вложенности[47].

Для человека, имеющего опыт написания компьютерных программ, оказываются довольно очевидными два момента. С одной стороны, если программу писать только с использованием арифметических циклов, таких как `for` в Паскале, и при этом без всякой рекурсии (или с использованием «примитивной» рекурсии, для которой количество вложенных вызовов не превышает заданного числа), то можно будет заранее оценить (сверху) время выполнения такой программы. Зацикливаться таким программам попросту негде, так что какие бы данные ни оказались на входе, за конечное число шагов такая программа заведомо завершится.

С другой стороны, увы, *ни одной мало-мальски полезной программы так не написать*. Как только мы пытаемся решить практическую задачу, нам приходится прибегнуть либо к циклу `while`, либо к рекурсии без «счётчика уровней», либо к оператору перехода назад (каковой на самом деле тоже представляет собой цикл `while`, так что вместо такого перехода как раз и следует использовать цикл). Ну а вместе с таким «недетерминированным повторением» в нашу программу просачиваются источники неопределённостей, алгоритмическая неразрешимость вопроса об останове и прочие прелести, известные пользователям под общим названием «глюки». При взгляде на действительность под таким углом оказывается, что общая «глючность» программ является не следствием безалаберности программистов, как это принято считать, а скорее математическим свойством предметной области. А всё потому, что, добавив в свою программу хотя бы одно недетерминированное повторение (будь то неарифметический цикл или непримитивная рекурсия), мы тем самым выводим алгоритм, реализуемый нашей программой, из класса примитивно-рекурсивных функций в класс частично-рекурсивных.

Таким образом, **неопределённость некоторых вычислимых функций в некоторых точках, или, что то же самое, неприменимость некоторых алгоритмов к некоторым входным словам — это фундаментальное свойство любых моделей конструктивных вычислений**, практически тривиально вытекающее из самых основ теории. С этим не только нельзя ничего сделать, но и *не нужно* ничего делать, иначе мы потеряем гораздо больше, чем приобретём: наши программы станут «правильными» и совершенно перестанут «глючить», только при этом они ещё и станут *бесполезны*.

1.5.7. Последовательность действий тут ни при чём

Среди «определений» алгоритма, которые, несмотря на их изначальную некорректность, всё же встречаются в литературе, и притом гораздо чаще, чем хотелось бы, преобладают вариации на тему «последовательности действий». Так вот, в действительности **алгоритм в**

[47] Отметим, что «оператор примитивной рекурсии» использует целочисленный параметр, который на каждом рекурсивном обращении уменьшается на единицу, так что количество оставшихся рекурсивных обращений всегда в точности равно этому параметру.

общем случае может не иметь ничего общего ни с какими последовательностями действий, во всяком случае, явно описанными.

В самом деле, мы уже упоминали, что для изучения алгоритмов используют самые разные формальные системы: среди них, например, машина Тьюринга (представляемая таблицей перехода между состояниями) и частичные рекурсивные функции (представлемые комбинацией базовых функций и операторов). Отметим, что в число алгоритмических формализмов входят также нормальные алгоритмы Маркова (набор правил по переписыванию слова, находящегося в поле зрения), лямбда-исчисление (системы функций одного аргумента над неким пространством выражений), абстрактная машина с неограниченными регистрами (МНР) и многие другие модели. Последовательность действий, выписанная в явном виде, среди всех этих формализмов присуща только МНР; между тем многократно доказано, что *все эти формализмы попарно эквивалентны*, то есть задают один и тот же класс возможных вычислений. Иначе говоря, **любой алгоритм может быть представлен в любой из имеющихся формальных систем**, среди которых большинство не имеет ничего общего с последовательностями действий.

Если вернуться с математических небес на грешную программистскую землю, мы обнаружим, что и **воплощение алгоритма в виде компьютерной программы далеко не всегда будет описанием последовательности действий**. Здесь всё зависит от используемой *парадигмы программирования*, то есть от стиля мышления, используемого программистом; во многом этот стиль определяется языком программирования. Так, при работе на языке Haskell и других языках *функционального программирования* у нас вообще нет никаких *действий*, есть лишь функции в сугубо математическом смысле — функции, вычисляющие некоторое значение на основе заданных аргументов. Вычисление такой функции выражается через другие функции, и это выражение не имеет ничего общего с *действиями* и их последовательностями. В большинстве случаев для программ на Haskell'е вообще невозможно предсказать, в какой последовательности будут произведены вычисления, поскольку язык реализует так называемую *ленивую семантику*, позволяющую отложить выполнение вычисления до тех пор, пока результат оного не потребуется для другого вычисления.

Если на Haskell'е мы хотя бы задаём *то, как нужно вычислять результат*, пусть даже и не в виде конкретной «последовательности действий», то при работе на языках программирования, имеющих *декларативную семантику*, мы вообще не обращаем внимания на то, *как* искать требуемое решение; мы лишь указываем, какими свойствами оно должно обладать, а то, как его найти, оставляем на усмотрение системы. Среди таких языков наиболее известен реляционный язык Пролог: программа на этом языке пишется в виде набора логических утверждений, а её исполнение представляет собой доказательство теоремы (точнее, попытку её опровержения).

Наконец, суперпопулярная ныне парадигма *объектно-ориентированного программирования* тоже основана не на «последовательностях действий», а на некоем *обмене сообщениями между абстрактными объектами*. Если же говорить о последовательностях действий, то в виде таковых программа пишется

лишь на так называемых *императивных* языках программирования, также называемых иногда «фоннеймановскими». К таким языкам относятся, например, Паскаль, Си, Бейсик и достаточно много других языков; однако популярность фоннеймановского стиля обусловлена исключительно господствующим подходом к построению архитектуры компьютера, а вовсе не «простотой» или тем более «естественностью» императивных конструкций.

В последние годы автору приходилось встречать утверждение, что объектно-ориентированное программирование есть якобы не более чем развитие идеи программирования императивного. Такие утверждения несостоятельны хотя бы потому, что основой императивного программирования является понятие *присваивания*, тогда как в каноническом ООП запрещены любые предположения о внутреннем состоянии объекта и, как следствие, операция присваивания (означающая. что состояние одного объекта должно стать копией состояния другого объекта) заведомо не может существовать. Причиной возникновения ощущения «императивной природы» ООП является исключительно то обстоятельство, что все индустриальные языки программирования, реализующие объектную парадигму, являются не чистыми объектно-ориентированными языками, а, напротив, языками императивными, имеющими ООП-надстройку; таковы и Си++, и Джава, и C#. Неудивительно, что в том же Си++ не просто можно присваивать объекты, но и присутствует специальная языковая поддержка этого; но всё это обусловлено не объектами, а исходно императивной природой языка Си++, основанного на Си.

Ясно, что программы на **любом** реально существующем языке программирования обладают свойством конструктивной вычислимости, иначе не могла бы существовать практическая реализация такого языка — а она существует. Очевидно, таким образом, что **компьютерная программа — это *всегда* алгоритм**, как её ни напиши; определение алгоритма как последовательности действий, следовательно, никуда не годится и может привести к достаточно опасным заблуждениям. Между прочим, в заблуждение авторы таких определений вгоняют также и самих себя: именно здесь кроется, по-видимому, источник часто встречающегося, но при этом совершенно безумного утверждения, что «языки программирования делятся на алгоритмические и неалгоритмические». Разумеется, в действительности **любой язык программирования является алгоритмическим**. Можно было бы, пожалуй, представить себе некий язык программирования, не обладающий свойством детерминированности, основанный, скажем, на эвристиках и, как следствие, не гарантирующий не только *правильности* программ, но даже их стабильности, а результаты выполнения программ делающий совершенно непредсказуемыми. Подобного рода подходы к решению некоторых задач, как ни странно, действительно существуют (вспомнить хотя бы те же нейронные сети), но до создания таких *языков программирования*, к счастью, дело не дошло, да и популярность нейронных сетей давно уже сошла на нет: с непредсказуемыми результатами тяжело работать, такие методы, в частности, полностью исключают отладку, так что практический потенциал неалгоритмических вычислений весьма сомнителен.

Впрочем, говоря о «неалгоритмических» языках, обычно имеют в виду сущность гораздо более простую: именно, все языки, на которых программа записывается иначе как в виде всё той же «последовательности действий». Строго говоря, «неалгоритмическими» тогда следовало бы, наверное, признать вообще

все современные языки, кроме разве что языка ассемблера, ведь даже насквозь императивные Си и Паскаль допускают рекурсивное программирование и, как следствие, написание программы в виде, не имеющем с «последовательностью действий» ничего общего.

Отметим, что для выражения той же мысли о «делении» языков программирования можно предложить вполне корректную формулировку: например, сказать, что *языки программирования делятся на императивные и неимперативные*, что, конечно, потребует хотя бы краткого пояснения термина «императивное программирование», и вот здесь как раз вполне имеет право появиться «последовательность действий».

Защищая свою картину мира, любители определять алгоритм как последовательность действий довольно часто прибегают к утверждению, что в теории алгоритмов-де рассматривается «не тот алгоритм», а теория вычислимости якобы вообще не имеет отношения к программированию. Как водится в таких случаях, опровергнуть такое утверждение оказывается несколько сложно — в самом деле, оно ведь при ближайшем рассмотрении оказывается сугубо терминологическим, ну а споры, центральным пунктом которых является значение того или иного термина — занятие неблагодарное.

Тем не менее, кое-что сказать на эту тему всё же можно и даже нужно. Начнём с алгоритмической неразрешимости: можно сколько угодно апеллировать к «абстрактной математичности» теории алгоритмов, но стоит предложить более-менее грамотному программисту разработать дополнение к операционной системе, которое бы само автоматически находило «зависшие» программы и останавливало бы их — и если программист что-то представляет собой в плане квалификации, мы тут же услышим, что за такую задачу он не возьмётся, да и никто не возьмётся, а если вдруг кто и возьмётся, то всё равно не решит, ведь это алгоритмически неразрешимая проблема. Получается, что в этом случае алгоритм оказался «именно тот».

Больше того, алгоритмическая неразрешимость проблемы применимости (проблемы останова) вдохновляет программистов на быструю и эффективную отправку «в сад» любых идей, связанных с доказательством свойств произвольной программы: уж если невозможно даже формально предсказать, остановится она или нет, то что́ говорить о свойствах более интересных.

Несмотря на это, стоит только упомянуть теорию вычислимости или какие-нибудь термины из неё, вроде «частично-рекурсивных функций», и у нас почему-то резко повышается риск нарваться на собеседника, утверждающего, что «это не тот алгоритм». В принципе, это явление легко объясняется: алгоритмическую неразрешимость обычно демонстрируют, не влезая в дебри теории вычислимости и даже вообще не упоминая её, причём делается это в большинстве случаев ровно так, как сделали мы на стр. 155. Доказательство неразрешимости пробле-

мы самоприменимости алгоритма практически тривиально, так что у большинства будущих программистов, хотя бы раз увидевших его, не возникает никаких трудностей с постижением сути алгоритмической неразрешимости. Иное дело теория вычислимости: это предмет весьма специфический, так что большинство программистов никогда толком и не слышали термина «частично-рекурсивная функция» и уж тем более не знают, что это за зверь. Отмахнуться от непонятного, естественно, гораздо проще, нежели в него вникать.

Больше того, выше мы уже показали, что классы функций, введённые в теории вычислимости, имеют к практическому программированию самое прямое отношение (см. рассуждение на стр. 157), а именно, что примитивно-рекурсивные функции соответствуют классу программ без «непредсказуемых» циклов, тогда как частично-рекурсивные функции соответствуют всему множеству компьютерных программ. Найдутся ли после этого желающие утверждать, что алгоритм в математическом смысле — это «не тот» алгоритм?

1.6. Программы и данные

1.6.1. Об измерении количества информации

Как мы уже знаем, при использовании компьютера для хранения и обработки информации используется двоичное представление; иначе говоря, любая информация в компьютере представляется нулями и единицами. Говорят, что *наименьшей единицей информации в компьютерной обработке является бит*; само слово ***бит*** (*bit*) образовано от английских *BInary digiT*, то есть «двоичная цифра».

Хотя компьютеры весьма удобны для обработки информации, всё-таки информация как таковая существует сама по себе и существовала задолго до появления первых счётных машин. Будучи явлением вполне объективным, информация *поддаётся измерению*, и под этим углом зрения столь привычный нам бит совершенно неожиданно предстаёт в новом свете. *Получить информацию* — значит *уменьшить неопределённость*; в таком случае **бит — это количество информации, которое уменьшает неопределённость вдвое**. Проиллюстрировать сказанное нам позволит следующая задача:

> *Вася задумал натуральное число, не превосходящее 1000, и предложил Пете угадать его, пообещав, что будет честно отвечать на вопросы, но только такие, которые предполагают ответ «да» или «нет». За какое количество вопросов Петя может гарантированно узнать, какое число задумано?*

Очевидно, что каждый ответ Васи даёт Пете ровно один бит информации. Начальная неопределённость составляет 1000 вариантов.

Если каждый полученный бит информации будет снижать неопределённость вдвое, то после получения 10 бит неопределённость понизится в $2^{10} = 1024$ раза, то есть окажется меньше единицы, так что за десять вопросов число должно быть гарантированно узнано. Осталось понять, как правильно сформулировать вопросы.

Если подходить к задаче с абстрактной точки зрения, достаточно перед заданием очередного вопроса каким-то (произвольным) образом разбивать все оставшиеся варианты на два равных по мощности множества, ну или, по крайней мере, отличающихся друг от друга по мощности не более чем на единицу, и спрашивать Васю, находится ли задуманное им число в одном из двух множеств. Например, Петя мог бы взять тысячу карточек, написать на каждой из них число, поделить карточки случайным образом на две колоды по 500 карточек в каждой, вручить одну колоду Васе и спросить, написано ли задуманное им число на одной из карточек; в случае отрицательного ответа выбросить все карточки, которые были вручены Васе, в случае же ответа положительного — наоборот, выбросить все, которые Васе не давались. Оставшиеся 500 карточек поделить на две колоды по 250 и одну из них снова предложить Васе. После второго ответа будет выброшено ещё 250 карточек, оставшиеся будут поделены на две колоды по 125. После третьего ответа оставшееся число карточек окажется нечётным, так что придётся поделить их на неравные части, и в результате после четвёртого ответа у нас, в зависимости от везения, останется то ли 64, то ли 63 карточки; после пятого ответа — то ли 31, то ли 32; после шестого — то ли 15, то ли 16; после седьмого — семь или восемь, после восьмого — три или четыре, после девятого — одна или две. Если осталась одна карточка, то задача решена, если же осталось две — придётся задать десятый вопрос, уж после него у нас гарантированно больше одной карточки остаться не может.

Конечно, технически вся эта канитель с карточками довольно занудна, хотя с математической точки зрения процесс построен безупречно. Но никто ведь не заставляет разбивать варианты на два множества как попало; за счёт небольшой технической поправки Петя вполне может обойтись без рисования карточек, а Васе не придётся их раз за разом просматривать. Достаточно поделить варианты пополам так, чтобы все числа из одного множества были меньше любого из чисел другого множества. На первом шаге Петя может спросить Васю, находится ли задуманное число в диапазоне от 1 до 500. В зависимости от ответа у Пети останутся в рассмотрении числа либо от 1 до 500, либо от 501 до 1000; следующий вопрос будет, соответственно, находится ли задуманное в диапазоне $[1, 250]$ или $[501, 750]$, и так далее; на каждом шаге количество вариантов, оставшихся в рассмотрении, будет уменьшаться вдвое.

Количество информации не обязано измеряться целым количеством битов. Широко известен, например, простенький карточный фокус, в котором ведущий сначала раскладывает на столе неполную колоду карт и предлагает зрителю загадать какую-нибудь карту. Затем ведущий раскладывает карты в три ряда и спрашивает, в каком из рядов находится карта; получив ответ, он собирает карты и снова раскладывает их в три ряда, а затем ещё раз. Получив третий ответ, ведущий откуда-то из середины названного ряда вытаскивает именно ту карту, которую загадал зритель, либо, для пущего эффекта, снова собирает карты в колоду, сколько-то там карт отсчитывает и сбрасывает, а оказавшуюся сверху карту открывает, и это оказывается именно та карта, которая была загадана.

В этом фокусе используется 27 карт, именно поэтому колода нужна неполная; обычно ведущий заранее откладывает в сторону шестёрки, семёрки и одну восьмёрку. Разложив карты в три ряда по девять и узнав от зрителя, в каком из рядов находится нужная карта, ведущий собирает карты, но не смешивает их; карты из ряда, названного зрителем, ведущий кладёт в середину колоды, поместив остальные два ряда сверху и снизу. Затем он раскладывет карты по одной снова в три ряда по девять; нетрудно догадаться, что «нужные» карты оказываются в каждом из новых рядов тремя средними, а остальные (три с любого края в любом ряду), хотя и выложены на стол, но пришли из рядов, которые заведомо не участвуют в рассмотрении. Повторив свой вопрос и снова поместив нужный ряд между двумя другими, ведущий снова раскладывает их по одной, но в этот раз в рассмотрении участвуют на самом деле только те три карты, которые лежат в самой середине каждого ряда; остальные лишь создают «массовку». Получив ответ, в каком ряду загаданная карта в этот раз, ведущий теперь уже точно знает, какая карта была загадана — она ведь находится в середине. Её можно просто достать, но тогда зритель может сразу догадаться, что происходит; поэтому обычно карты снова складывают в колоду, причём «нужный» ряд опять кладут посередине, отсчитывают 13 карт сверху и открывают следующую, четырнадцатую, которая как раз и лежала в середине нужного ряда и, следовательно, именно её загадал зритель.

В нашем разговоре о количестве информации этот простой фокус примечателен лишь тем, что с каждым ответом зрителя неопределённость, исходно составлявшая 27 вариантов, уменьшается *втрое*; такую единицу информации иногда называют ***тритом***. Если попытаться выразить трит в битах, получится иррациональное число $\log_2 3$: в самом деле, сколько раз нужно уменьшать неопределённость вдвое (то есть получать один бит информации), чтобы в итоге она уменьшилась втрое? Естественно, именно такое число раз, которое, если в такую степень возвести двойку, даст тройку, а это и есть пресловутый логарифм.

Рассмотрим более сложный случай.

Два археолога, изучая архив старого за́мка, узнали, что во дворе замка, имеющем форму прямоугольника 60×80 метров, закопан клад. Пока один археолог ездил за металлоискателем, второй археолог продолжил изучение архивов и выяснил, что клад закопан в юго-восточной части двора, причём на расстоянии не более 30 м. от короткой стены замка и не более 20 м. от длинной. Какова информационная ценность его открытия?

Почему-то эта задача ставит в тупик даже многих программистов; между тем нетрудно убедиться, что исходная площадь поиска клада составляла 60×80 м^2, тогда как после обнаружения новых сведений она сократилась до 30×20 м^2, то есть ровно в восемь раз. Это и есть уменьшение неопределённости; коль скоро один бит уменьшает неопределённость вдвое, то здесь мы имеем дело с *тремя битами информации*.

Как правило, в задачах, подобных этой, речь идёт о *вероятностях* наступления неких событий, а затем о том, что было получено сообщение, указывающее на наступление некоторой комбинации таких событий, и спрашивается, какова информационная ёмкость полученного сообщения. Например:

В первый день турнира по настольному теннису Васе предстояло сыграть сначала с Петей, а затем с Колей. Наблюдая тренировочные игры, проходившие до турнира, болельщики оценили, что Вася играет с Колей приблизительно на равных, а вот Петя побеждает Васю в 75% случаев.

Маша не смогла присутствовать на турнире, но болела за Васю и попросила друзей сообщить ей результаты обеих его игр. Через некоторое время ей пришло SMS-сообщение, в котором говорилось, что Вася обыграл обоих соперников. (1) Какова информационная ёмкость этого сообщения? (2) Если бы Вася, наоборот, проиграл обе партии и Маше сообщили бы об этом, какова тогда была бы ёмкость сообщения?

Решаются такие задачи ничуть не сложнее задачи про археологов во дворе замка, просто вместо двора замка здесь мы имеем *пространство элементарных исходов*, а вместо площади — вероятность. Вероятность победы Васи в первой партии, согласно условиям задачи, составляет $\frac{1}{4}$, вероятность победы во второй партии — $\frac{1}{2}$; поскольку для комбинаций независимых событий вероятности перемножаются, получается, что вероятность того, что Вася выиграет обе партии, составляет $\frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8}$, то есть информация о том, что Вася выиграл обе партии, уменьшает неопределённость в восемь раз. Следовательно, информационная ценность сообщения составляет 3 бита.

Вероятность того, что Вася проиграет обе партии, составляет $\frac{3}{8}$, так что здесь информационная ценность сообщения окажется, во-первых, ниже, поскольку наступило более вероятное событие, и, во-вторых, в этот раз искомое число будет не только не целым, но даже и не рациональным: поскольку неопределённость в

этот раз уменьшается в $\frac{8}{3}$ раза, информационный объём сообщения составит $\log_2 \frac{8}{3} = 3 - \log_2 3$.

Наряду с термином «бит» используется также термин «байт» (англ. *byte*), обычно подразумевающий восемь бит. Как ни странно, это не всегда так: существовали компьютеры, у которых минимальная ячейка памяти составляла, например, семь бит, а не восемь, и на этих компьютерах байтом называли обычно как раз количество информации, хранящееся в одной такой ячейке; ввели даже специальный термин ***октет*** для обозначения именно восьми бит. К счастью, вам вряд ли придётся на практике столкнуться с пониманием байта, отличным от восьми бит, но иметь в виду историю возникновения этого термина в любом случае стоит.

Восьмибитный байт может принимать одно из 256 возможных значений; обычно эти значения интерпретируют либо как число от 0 до 255, либо как число от -128 до 127. Восьмибитные ячейки памяти хорошо подходят для хранения букв, составляющих текст, если этот текст написан на языке, использующем алфавит вроде латиницы; для представления каждой отдельной буквы при этом используется так называемый ***код символа***, причём всех различных кодов существенно меньше, чем существует возможных значений байта. С многоязычными текстами всё несколько хуже: для кириллицы кодов ещё хватает, а, например, с иероглифами этот подход уже не годится. К вопросу о кодировке текста мы ещё вернёмся.

Раз уж речь зашла о ячейках памяти, следует заметить, что в них приходится хранить вообще любую информацию, которая обрабатывается компьютером, в том числе *программы* или, точнее, коды команд, из которых программы состоят. Когда диапазона значений одной ячейки не хватает, используют несколько ячеек, находящихся в памяти подряд, и говорят уже не о ячейке, а об ***области памяти***.

Важно понимать, что сама по себе ячейка памяти «не знает», как именно следует интерпретировать хранящуюся в ней информацию. Рассмотрим это на простейшем примере. Пусть у нас есть четыре идущие подряд ячейки памяти, содержимое которых соответствует шестнадцатеричным числам `41`, `4E`, `4E` и `41` (соответствующие десятичные числа — 65, 78, 78, 65). Информацию, содержащуюся в такой области памяти, можно с совершенно одинаковым успехом истолковать как целое число 1095650881; как дробное число (т. н. число с плавающей точкой) 12.894105; как текстовую строку, содержащую имя `'ANNA'`; наконец, как последовательность машинных команд. В частности, на процессорах платформы i386 это будут команды, условно обозначаемые `inc ecx`, `dec esi`, `dec esi`, `inc ecx`; О том, что эти команды делают, мы расскажем во втором томе книги.

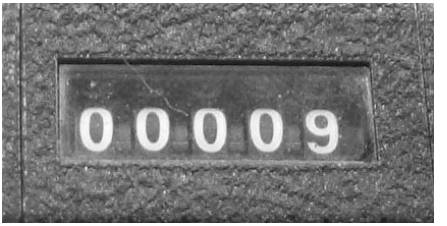

Рис. 1.12. Механический счётчик

1.6.2. Машинное представление целых чисел

О том, что компьютеры используют двоичную систему счисления, мы уже говорили. При этом компьютеры, будучи реально существующими техническими устройствами, накладывают некоторые ограничения на представление целых чисел. В математике множество целых чисел бесконечно, то есть каково бы ни было число N, всегда существует *следующее* число $N+1$. Для этого и количество знаков в записи числа, какую бы систему мы ни использовали, не должно никак ограничиваться — но вот как раз это требование исполнить технически невозможно (даже чисто теоретически: ведь количество атомов во вселенной считается конечным).

На практике для компьютерного представления целого числа выделяется некоторое фиксированное количество разрядов (бит); обычно это 8 бит (одна ячейка), 16 бит (две ячейки), 32 бита (четыре ячейки) или 64 бита (восемь ячеек). Ограничение разрядности приводит к появлению «наибольшего числа», и это касается не только двоичной системы. Представьте себе, например, простое счётное устройство, используемое в электрических счётчиках и механических одометрах старых автомобилей: цепочку роликов, на которых нанесены цифры и которые могут прокручиваться, а проходя через «точку переноса» (с девятки на ноль), прокручивают на единицу следующий ролик. Допустим, такое устройство состоит из пяти роликов (см. рис. 1.12). Сначала мы видим на нём число «ноль»: `00000`. По мере прокручивания правого ролика число будет меняться, мы увидим `00001`, потом `00002` и так до `00009`. Если теперь провернуть самый правый ролик ещё на единичку, мы снова увидим в правой позиции ноль, но при этом самый правый ролик зацепит своего соседа слева и заставит его провернуться на единичку, так что мы увидим `00010`, то есть число «десять»; мы наблюдали при этом хорошо известный с младших классов *перенос*: «девять плюс один, ноль пишем, один в уме». То же самое произойдёт при переходе от числа `00019` к числу `00020` и так далее, а когда мы увидим число `00099` и прокрутим правый ролик ещё на единичку, в зацепление попадут

сразу два его соседа, так что на единицу вперёд прокрутятся сразу три ролика, и мы получим число «сто»: `00100`.

Теперь уже легко понять, откуда берётся такой монстр, как «наибольшее возможное число»: рано или поздно наш счётчик досчитает до `99999`, и теперь увеличивать число окажется некуда; когда мы в очередной раз прокрутим правый ролик на единицу вперёд, он зацепит за собой все остальные ролики, они все перейдут на следующую цифру, и мы снова увидим одни нули. Если бы у нас слева был ещё один ролик, он бы зацепился и показал единичку, так что результат бы был `100000` (что совершенно правильно), но у нас всего пять роликов, шестого нет. Такая ситуация называется ***перенос в несуществующий разряд***; мы уже сталкивались с ней при обсуждении вычитания на арифмометре Паскаля (см. стр. 35). Ясно, что такая ситуация не может возникнуть, когда мы пишем числа на бумаге: всегда можно дописать ещё одну цифру слева; когда же число представлено состоянием некой группы технических устройств, будь то цепочка роликов или набор триггеров в оперативной памяти компьютера, возможности приделать к числу ещё одну цифру у нас нет.

При использовании двоичной системы счисления происходит примерно то же самое с той разницей, что используется всего две цифры. Допустим, мы используем для подсчёта каких-нибудь предметов или событий ячейку памяти, которая содержит восемь разрядов. Сначала в ячейке ноль: `00000000`. Добавив единицу, получаем двоичное представление единицы: `00000001`. Добавляем ещё одну единицу, младший (самый правый) разряд увеличивать некуда, поскольку у нас только две цифры, поэтому он снова становится нулевым, но при этом происходит перенос, в результате которого единица появляется во втором разряде: `00000010`; это двоичное представление числа `2`. Дальше будет `00000011`, `00000100` и так далее. В какой-то момент во всех имеющихся разрядах окажутся единицы, так что прибавлять дальше будет некуда: `11111111`; это двоичное представление числа `255` ($2^8 - 1$). Если теперь прибавить ещё единицу, вместо числа `256` мы получим «все нули», то есть просто ноль; произошел уже знакомый нам перенос в несуществующий разряд. Вообще **при использовании для представления целых положительных чисел позиционной несмешанной системы счисления по основанию N и ограничении количества разрядов числом k максимальное число, которое мы можем представить, составляет $N^k - 1$**; так, в нашем примере со счётчиком было пять разрядов десятичной системы, и максимальным числом оказалось $99999 = 10^5 - 1$, а в примере с восьмибитной ячейкой система использовалась двоичная, разрядов было восемь, так что максимальным числом оказалось $2^8 - 1 = 255$.

Некоторые языки программирования высокого уровня позволяют оперировать сколь угодно большими целыми числами, лишь бы хватило памяти, но мы

пока не будем рассматривать такие инструменты: нам важно понять, как работает компьютер, тогда как языки высокого уровня от нас устройство компьютера стараются по возможности скрыть.

Мы уже отмечали, что одна ячейка обычно состоит из восьми разрядов и может, соответственно, хранить число от 0 до 255, если же требуется хранить число из бóльшего диапазона, используют несколько идущих подряд ячеек памяти, и здесь совершенно неожиданно возникает вопрос, *в каком порядке* располагать части представления одного числа в соседних ячейках памяти. На разных машинах используют два разных подхода к порядку следования байтов. Один подход, называемый *little-endian*[48], предполагает, что первым идёт самый младший байт числа, далее в порядке возрастания, и самый старший байт идёт последним. Второй подход, который называют *big-endian*, прямо противоположен: сначала идёт старший байт числа, а младший располагается в памяти последним.

Например, число 7500 в шестнадцатеричной системе записывается как $1D4C_{16}$. Если представить его в виде 32-битного (4-байтного) целого на компьютере, использующем подход *big-endian*, то область памяти длиной в четыре ячейки, хранящая это число, окажется заполнена так: первые два байта (с наименьшими адресами) получат значение 00, следующий (третий) будет установлен в значение 1D, и последний, четвёртый — в значение 4C: 00 00 1D 4C. Если то же самое число записать в память компьютера, использующего подход *little-endian*, то значения отдельных байтов в соответствующей области памяти окажутся идущими в противоположном порядке: 4C 1D 00 00.

Большинство компьютеров, используемых в наши дни, относятся к категории «little-endian», то есть хранят младший байт первым, хотя машины «big-endian» тоже иногда встречаются.

Посмотрим теперь, как быть, если необходимо работать не только с положительными числами. Ясно, что нужен какой-то другой способ интерпретации комбинаций двоичных разрядов, такой, чтобы какие-то из комбинаций считались представляющими отрицательное число. Будем в таких случаях говорить, что ячейка или область памяти хранит *знаковое целое число*, в отличие от предыдущего случая, когда говорят о *беззнаковом целом числе*.

На заре вычислительной техники для представления отрицательных целых чисел пытались использовать разные подходы, например, хранить знак числа как отдельный разряд. Оказалось, однако, что при этом

[48] «Термины» big-endians и little-endians введены Джонатаном Свифтом в книге «Путешествия Гулливера» для обозначения непримиримых сторонников разбивания яиц соответственно с тупого конца и с острого. На русский язык эти названия обычно переводились как «тупоконечники» и «остроконечники». Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

неудобно реализовывать даже самые простые операции — сложение и вычитание, потому что приходится учитывать знаковый бит обоих слагаемых. Поэтому создатели компьютеров достаточно быстро пришли к использованию так называемого ***дополнительного кода***[49].

Чтобы понять, как устроен дополнительный код, вернёмся к нашему примеру с механическим счётчиком. В большинстве случаев такие роликовые цепочки умеют крутиться как вперёд, так и назад, и если прокрутка вперёд давала нам прибавление единицы, то прокрутка назад будет выполнять вычитание единицы. Пусть теперь у нас все ролики выставлены на ноль и мы откручиваем счётчик назад. Результатом этого будет 99999; оно и понятно, ведь когда мы к 99999 прибавили единицу, то получилось 00000, а теперь мы проделали обратную операцию. Говорят, что у нас произошел ***заём из несуществующего разряда***: как и в случае с переносом в несуществующий разряд, если бы у нас был ещё один ролик, всё бы было правильно (например, $100000 - 1 = 99999$), но его нет. То же самое происходит и в двоичной системе: если во всех разрядах ячейки были нули (`00000000`) и мы вычли единицу, получим все единицы: `11111111`; если теперь снова прибавить единицу, мы снова получим нули во всех разрядах. Это логично приводит нас к идее **использовать в качестве представления числа `-1` единицы во всех разрядах двоичного числа**, сколько бы ни было у нас таких разрядов. Так, если мы работаем с восьмиразрядными числами, `11111111` у нас теперь означает `-1`, а не `255`; если мы работаем с шестнадцатиразрядными числами, `1111111111111111` теперь будет обозначать, опять-таки, `-1`, а не `65535` и так далее.

Продолжая операцию по вычитанию единицы над восьмиразрядной ячейкой, мы придём к заключению, что для представления числа `-2` нужно использовать `11111110` (раньше это было число `254`), для представления `-3` — `11111101` (раньше это было 253) и так далее. Иначе говоря, мы волюнтаристски объявили часть комбинаций двоичных разрядов представляющими отрицательные числа вместо положительных, причём всегда новое (отрицательное) значение комбинации разрядов получается из старого (положительного) путём вычитания из него числа `256`: $255 - 256 = -1$, $254 - 256 = -2$ и т. д. (число `256` представляет собой 2^8, а наши рассуждения верны только для частного случая с восьмиразрядными числами; в общем случае из старого значения нужно вычитать число 2^n, где n — используемая разрядность). Остаётся вопрос, в какой момент остановиться, то есть перестать считать числа отрицательными; иначе, увлёкшись, мы можем дойти до `00000001` и заявить, что это вовсе не `1`, а `-255`. Принято следующее соглашение: **если набор двоичных разрядов рассматривается как представление знакового числа, то *отрицательными* считаются комбинации, старший бит которых равен 1**, а остальные комбинации считаются

[49] Английский термин — two's complement, то есть «двоичное дополнение».

положительными. Таким образом, наибольшее по модулю отрицательное число будет представлено одной единицей в старшем разряде и нулями во всех остальных; в восьмибитном случае это `10000000`, `-128`. Если из этого числа вычесть единицу, получится `01111111`; эта комбинация (старший ноль, остальные единицы) считается представлением *наибольшего знакового числа* и для восьмибитного случая представляет, как несложно видеть, число `127`. Как вы уже догадались, прибавление единицы к этому числу снова даст наибольшее по модулю отрицательное. Переход через границу между комбинациями `011...11` и `100...00` для знаковой целочисленной арифметики[50] представляет собой аналог переноса и займа для несуществующего разряда, который мы наблюдали в арифметике беззнаковой, но называется эта ситуация иначе: ***переполнение***.

Именно такое, а не какое-либо другое расположение границы переполнения даёт две приятные возможности. Во-первых, знак числа можно определить, взяв от него всего один (старший) бит. Во-вторых, оказывается очень простой операция *смены знака числа*. **Чтобы сменить знак числа на противоположный при использовании дополнительного кода, достаточно сменить значения во всех разрядах на противоположные, а к полученному значению прибавить единицу.** Например, число `5` представляется следующим восьмибитным знаковым целым: `00000101`. Чтобы получить представление числа `-5`, мы сначала инвертируем все разряды, получаем `11111010`; теперь прибавляем единицу и получаем `11111011`, это и есть представление числа `-5`. Для наглядности проделаем смену знака ещё раз: инвертируем все биты в представлении числа `-5`, получаем `00000100`, прибавляем единицу, получаем `00000101`, то есть снова число `5`, что и требовалось. Как несложно убедиться, для представления нуля операция смены знака инвариантна, то есть ноль остаётся нулём: `00000000` $\xrightarrow{inv.}$ `11111111` $\xrightarrow{+1}$ `00000000`.

Такая же ситуация несколько неожиданно возникает для числа `-128` (в восьмибитном случае) или, говоря вообще, для максимального по модулю отрицательного числа заданной разрядности: `100000000` $\xrightarrow{inv.}$ `01111111` $\xrightarrow{+1}$ `10000000`. Это обусловлено отсутствием в данной разрядности положительного числа с таким же модулем, то есть при применении операции замены знака к комбинации `100...00` происходит *переполнение*.

Если отрицательные числа представлять в дополнительном коде, сложение и вычитание реализуется на аппаратном уровне абсолютно одинаково вне зависимости от знаков слагаемых и даже от самого факта их «знаковости»: мы можем по-прежнему рассматривать все возмож-

[50] То есть когда сумма двух положительных оказывается отрицательной и наоборот.

ные битовые комбинации как представление неотрицательных чисел (то есть вернуться к беззнаковой арифметике), и схематически сложение и вычитание от этого не изменятся. Благодаря этому отпадает надобность в отдельном электронном устройстве для вычитания: операция вычитания может быть реализована как операция прибавления числа, которому сначала сменили знак, причём это, как ни парадоксально, работает и для беззнаковых чисел.

Дело в том, что благодаря переносам в несуществующий разряд и займам из него *вычитание* числа из другого числа можно выполнить как *сложение* уменьшаемого с неким другим числом, которое само по себе очень легко вычислить; это число как раз и есть дополнительный код исходного вычитаемого. При этом сложение можно выполнять как если бы числа были беззнаковыми. Пусть нам, к примеру, нужно вычислить разность $75 - 19$ на восьмибитных числах. Двоичное представление числа 75 — `01001011`, числа 19 — `00010011`; инвертировав его и прибавив единицу, получаем `11101101` — представление числа -19. Теперь сложим «столбиком» `01001011` и `11101101`, в результате получится двоичное число $\texttt{100111000}_2$, которое имеет девять разрядов. Но у нас всего восемь разрядов, поэтому старший разряд при сложении окажется просто отброшен, и мы получим комбинацию `00111000`, которая представляет собой двоичное представление числа 56.

Арифметическая основа здесь довольно проста. Сделать побитовую инверсию числа — это то же самое, как вычесть это число из $\texttt{11111111}_2 = 255_{10}$; поскольку мы ещё прибавляем единицу, то получается, что при смене знака числа мы заменяем число x на $256 - x$. Вместо вычитания $y - x$ мы, получается, выполняем сложение $y + (256 - x)$, но при этом у нас происходит перенос в несуществующий разряд, в результате которого мы теряем 256; окончательный результат составляет $y + (256 - x) - 256 = y - x$. Этот эффект мы уже обсуждали (см. стр. 35).

1.6.3. Числа с плавающей точкой

Далеко не любые вычисления можно выполнить в целых числах; точнее, выполнить на самом деле можно любые[51], просто это будет не слишком удобно. Поэтому наряду с целыми числами, представление которых описано в предыдущем параграфе, компьютеры также умеют обрабатывать дробные числа, называемые ***числами с плавающей точкой***. Такие числа предполагают отдельное хранение ***мантиссы*** M (двоичной дроби из интервала $1 \leqslant M < 2$) и ***машинного порядка*** P — целого числа, означающего степень двойки, на которую следует

[51] В самом деле, с помощью целых чисел очевидным образом можно представлять числа рациональные (в виде дроби числитель/знаменатель), либо воспользоваться концепцией так называемых чисел с фиксированной точкой, когда считается, что целое число представляет не единицы, а, скажем, стотысячные доли обрабатываемой величины.

умножить мантиссу. Отдельный бит s выделяется под знак числа: если он равен 1 — число считается отрицательным, иначе положительным. Итоговое число, таким образом, вычисляется как $N = (-1)^s M 2^P$. Набор частных соглашений о формате чисел с плавающей точкой, известный как *стандарт IEEE-754*, в настоящее время используется практически всеми процессорами, способными работать с дробными числами.

Поскольку целая часть мантиссы всегда равна 1, её можно не хранить[52], используя имеющиеся разряды для хранения цифр дробной части. Для хранения машинного порядка в разное время применялись разные способы (знаковое целое с использованием дополнительного кода, отдельный бит для знака порядка и т. п.); стандарт IEEE-754 предполагает хранение машинного порядка в виде *смещённого* беззнакового целого числа: соответствующие разряды рассматриваются как целое число без знака, из которого для получения машинного порядка вычитают некоторую константу, называемую *смещением машинного порядка*.

Стандарт IEEE-754 устанавливает три основных типа чисел с плавающей точкой: число обычной точности, число двойной точности и число повышенной точности[53]. Число обычной точности занимает в памяти 32 бита, из которых один используется для хранения знака числа, восемь — для хранения смещённого машинного порядка (величина смещения — 127) и оставшиеся 23 — для хранения мантиссы. Число двойной точности занимает 64 бита, причём на машинный порядок отводится 11 бит, а на мантиссу — 52, и смещение машинного порядка составляет 1023. Наконец, число повышенной точности занимает 80 бит, из них 15 бит отведено на машинный порядок со смещением 16383, а оставшиеся 64 составляют мантиссу, причём в этом формате присутствует однобитовая целая часть мантиссы (обычно единица).

Машинный порядок, состоящий из одних нулей или, наоборот, из одних единиц, представляет собой признак особого случая. Порядок, состоящий из одних нулей, означает:

- при мантиссе, состоящей из одних нулей — в зависимости от знакового бита либо ноль, либо «отрицательный ноль» (это различие бывает полезно, если результат очередной операции столь мал по модулю, что его невозможно представить в виде числа с плавающей точкой — тогда мы хотя бы можем сказать, каков был знак результата);
- при мантиссе, содержащей хотя бы одну единицу — ***денормализованное число***, то есть число настолько малое по модулю, что даже при наименьшем возможном значении машинного порядка ни один значащий бит не попал бы в разряды мантиссы.

Порядок, состоящий из одних единиц, может означать следующее:

- при мантиссе, состоящей из одних нулей — «бесконечность» (положительную или отрицательную в зависимости от знакового бита);

[52] За исключением нескольких особых случаев, о которых речь пойдёт дальше.

[53] Соответствующие англоязычные термины — single precision, double precision и extended precision.

- при первом бите мантиссы, установленном в единицу (для 80-битных чисел — при первых двух битах мантиссы, установленных в единицу), а остальных битах мантиссы, установленных в ноль, знаковый бит, равный единице, означает «неопределённость», а знаковый бит, равный нулю — «не-число типа QNAN» (quiet not-a-number); иногда говорят, что неопределённость есть частный случай QNAN;
- при первом бите мантиссы, равном нулю (для 80-битных — при двух первых битах мантиссы, равных 10) и при наличии в остальной мантиссе единичных битов — «не-число типа SNAN»;
- все остальные ситуации (например, мантисса из одних единиц) означают «неподдерживаемое число».

Ясно, что даже самые простые арифметические действия над числами с плавающей точкой производятся гораздо сложнее, чем над целыми. Например, чтобы сложить или вычесть два таких числа, необходимо сначала произвести *приведение к одному порядку*; для этого мантиссу числа, машинный порядок которого меньше, чем у другого, сдвигают вправо на нужное количество позиций. Дальше производится собственно сложение или вычитание, а затем, если результат оказался меньше единицы или больше либо равен 2, его подвергают *нормализации*, то есть изменяют порядок и одновременно сдвигают мантиссу так, чтобы значение числа не изменилось, но при этом мантисса снова стала удовлетворять условию $1 \leqslant M < 2$. Аналогичная нормализация производится при перемножении и делении чисел.

Как можно заметить, при любых арифметических операциях над числами с плавающей точкой приходится применять сдвиги мантисс — как влево, так и вправо, и при сдвиге вправо младшие биты мантиссы, которым не нашлось места в отведённых разрядах, просто теряются. Возникающая при этом разница между получившимся результатом и тем, что должно было бы получиться, если бы ничего не отбрасывалось, называется ***ошибкой округления***. Вообще говоря, ошибка округления при действиях с двоичными (как, впрочем, и с десятичными) дробями неизбежна, сколько бы разрядов мы ни отвели под хранение мантиссы, ведь даже обыкновенное деление двух целых чисел, которые образуют несократимую дробь и при этом делитель не является степенью двойки, даст в результате *периодическую двоичную дробь*, для «точного» хранения которой нам потребовалось бы бесконечное количество памяти. Бесконечным (хотя и периодическим) оказывается представление в виде двоичной дроби таких чисел, как $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{6}$, $\frac{1}{7}$, $\frac{1}{9}$ и т. д., так что при переводе их в формат числа с плавающей точкой неизбежно приходится отбрасывать значащие биты. Таким образом, **вычисления над числами с плавающей точкой практически всегда дают не точный результат, а приблизительный**[54]. В частности, программисты счи-

[54]Между прочим, автору этих строк неоднократно встречалось утверждение о том, что вообще любые вычисления на компьютере производятся с ошибками; как

тают неправильным пытаться сравнить два числа с плавающей точкой на строгое равенство, потому что числа могут оказаться «не равны» только из-за ошибок округления.

1.6.4. Тексты и языки

Текстовые данные, то есть информация, представленная в виде, понятном человеку — это, пожалуй, самый универсальный способ работы с информацией, ведь в виде текста человек может описать почти всё, что угодно: выразительной мощности естественных языков, таких как русский или английский, для этого обычно достаточно; философы, объединённые общим направлением под названием «лингвистический позитивизм», вообще считают, что границы языка совпадают с границами мышления.

Впрочем, здесь возникает одна интересная проблема: в общем случае текст на естественном языке с крайним трудом поддаётся машинному анализу. Проблемами, связанными с таким анализом, занимается целое направление науки — компьютерная лингвистика. Учёные более-менее справились с морфологическим анализом текста на естественном языке, то есть могут заставить компьютерную программу по виду слова и контексту определить, что это за слово и в какой оно находится форме: например, последовательность букв «полка» может означать существительное «полка» в именительном падеже, но с таким же успехом и существительное «полк» в родительном, и без анализа контекста их никак не различить. Условно успешными можно назвать исследования в области синтаксического анализа текста, который приблизительно соответствует школьному «разбору по членам предложения». Что касается *семантического* анализа текста на естественном языке, результатом которого должно стать понимание смысла написанного, то не факт, что эта задача вообще когда-нибудь будет решена, и рост вычислительной мощности компьютеров здесь никак не поможет: проблема не в количестве вычислений, а в алгоритмах анализа, сложность которых, судя по всему, превышает человеческие возможности.

Как обычно, если машину не удаётся полностью приспособить к потребностям людей, то людям, напротив, приходится приспосабливаться к возможностям машины. Машину невозможно (и в обозримом будущем не станет возможно) заставить понимать текст на естественном языке, причём виновата здесь, разумеется, не машина — к ней вообще неприменимо понятие «быть виноватым» — а люди, программисты, которые отнюдь не всемогущи. Однако стоит чуть-чуть упростить задачу, задать какие-то достаточно простые формальные правила построения языко-

---

можно догадаться, источником этой ахинеи стали всё те же школьные учебники информатики. Не верьте, вас обманывают! Вычисления в целых числах компьютер производит абсолютно точно.

вых конструкций и предложить людям оформлять тексты в соответствии с этими правилами — и программы, написанные программистами, прекрасно справляются с задачей анализа такого текста.

Когда заданы формальные правила построения текста, в рамках которых человек может задавать информацию так, что компьютерные программы его поймут, говорят, что создан ***формальный язык*** — в противоположность *естественным языкам*, сложившимся в ходе развития цивилизации как средство общения людей между собой. Впрочем, формальные языки не ограничены областью использования компьютеров и возникли намного раньше; примерами формальных знаковых систем являются, в частности, хорошо знакомые нам математические формулы, а также нотные знаки, используемые в музыке, и многое другое.

В общем случае под *языком* можно понимать всё (обычно бесконечное) множество текстов, которое соответствует установленным правилам, причём для случая естественных языков такие правила оказываются очень и очень сложны, а для формальных языков могут быть совсем простыми, а могут оказаться тоже сравнительно сложными, хотя и не настолько сложными, как правила естественных языков. Конечно, такое понимание языка оказывается несколько огрублённым, поскольку никак не учитывает семантику (то есть, попросту говоря, смысл) текста, ради которого текст, вообще говоря, и существует; тем не менее, когда речь идёт об автоматическом (т. е. компьютерном) анализе текста, до его смысла дело доходит отнюдь не сразу, а на ранних стадиях анализа понимание языка как множества цепочек символов вполне соответствует поставленным задачам.

Чем сложнее правила языка, тем сложнее получается компьютерная программа, предназначенная для его анализа, поэтому формальные языки обычно создаются как результат компромисса между требованиями задачи и возможностями программистов. Например, когда поставлена задача построить график или гистограмму по результатам каких-то измерений или подсчётов, то для представления исходных данных вполне можно использовать язык, предполагающий запись последовательности чисел в десятичной системе счисления, разделённых символом пробела и/или перевода строки, и не допускающий больше ничего. С другой стороны, если поставлена задача написания сложных компьютерных программ, то для этого нужен формальный язык особого вида — ***язык программирования***; некоторые из таких языков бывают чрезвычайно сложны, хотя, конечно, всё равно не могут сравниться по сложности с естественными языками.

Сказанное никоим образом не означает, что компьютер вообще не может работать с текстом на естественном языке; напротив, компьютеры в современных условиях только этим и занимаются. Книга, которую

вы читаете, была набрана в текстовом редакторе, её оригинал-макет был подготовлен с помощью системы компьютерной вёрстки; когда вы отправляете и получаете электронную почту и прочие сообщения, читаете сайты в Интернете или электронные книги с помощью карманного ридера — во всех этих случаях мы наблюдаем, как компьютер работает с естественным текстом. Конечно, компьютер не понимает, о чём говорится в книге, или на сайте, или в электронном письме, которое читает пользователь, но это и не требуется: во всех перечисленных случаях задача компьютера состоит лишь в том, чтобы принять текст на естественном языке от одного человека и предъявить другому.

Впрочем, компьютеры (точнее, компьютерные программы) могут проделывать с естественноязычными текстами куда более сложные вещи. Одним из самых эффектных, хотя и отнюдь не самых сложных примеров на эту тему можно считать программы, «поддерживающие беседу» с человеком, первая из которых — «Элиза»[55] была написана ещё в 1966 году американским учёным Йозефом Вайценбаумом. У человека, сидящего за клавиатурой компьютера, программы типа «Элизы» могут создать впечатление, что «по ту сторону» находится живой человек, хотя люди, которые знают, в чём дело, на самом деле достаточно легко отличают человека от программы, специально создавая разговорную ситуацию, в которой программе не хватает «совершенства». Примечательно, что такие программы совершенно не вникают в смысл текста, то есть не производят семантического анализа реплик собеседника; вместо этого они анализируют структуру полученных фраз и сами используют слова, полученные от пользователя, для конструирования фраз, смысла которых не понимают.

Спектр существующих формальных языков тоже довольно широк; одних только языков программирования за всю историю компьютеров создано несколько тысяч, хотя далеко не все их них существуют сейчас — многие языки программирования растеряли всех своих сторонников и умерли, как говорится, естественной смертью. Впрочем, *поддерживаемых* языков программирования, то есть таких, на которых мы могли бы писать программы, если бы захотели, и затем исполнять эти программы на компьютерах, тоже существует по меньшей мере несколько сотен.

Кроме них, широко используются так называемые ***языки разметки***, предназначенные для оформления текстов; пожалуй, самый известный из них — язык HTML, используемый для создания гипертекстовых страниц на сайтах в Интернете. Отметим, что во многих школьных

[55] По одной из версий, программа была названа в честь героини пьесы «Пигмалион» Бернарда Шоу; нечто общее тут действительно прослеживается, ведь в пьесе профессор Хиггинс обучает Элизу правильно произносить слова, но поначалу совершенно упускает из виду прочие аспекты, отличающие даму высшего общества от девушки-цветочницы.

учебниках можно найти совершенно безумное утверждение, что HTML якобы является языком программирования; не верьте.

Наконец, вообще любая компьютерная программа, принимающая на вход информацию в виде текста, самим фактом своего существования задаёт определённый формальный язык, состоящий из всех таких текстов, на которых данная программа отрабатывает без ошибок. Такие языки часто бывают очень и очень примитивными и, как правило, не имеют названия.

При обсуждении формальных языков иногда возникают сомнения, можно ли тот или иной язык считать языком программирования, то есть языком, на котором пишут программы. В некоторых случаях ответ зависит от ответа на вопрос, что такое, собственно говоря, «программа», причём этот вопрос оказывается не так прост, как кажется на первый взгляд; во всяком случае, *существуют* такие ситуации, когда на вопрос, является ли некий текст программой или нет, разные люди дают разные ответы. Так, при работе с базами данных довольно популярен язык запросов SQL; встречаются утверждения, что написание запросов на этом языке представляет собой программирование.

Исключить терминологическую неясность позволяет введение более узкого термина; языки, на которых можно записать любой алгоритм, называют ***алгоритмически полными языками***. Поскольку, как мы помним, понятие алгоритма не имеет определения, вместо него используется любой из введённых формализмов, чаще всего машина Тьюринга; с формальной точки зрения алгоритмически полным считают такой язык, на котором можно написать интерпретатор (если угодно, симулятор или модель) машины Тьюринга. Некоторые авторы предпочитают для большей определённости называть такие языки не «алгоритмически полными», а ***Тьюринг-полными***. Стоит заметить, что алгоритмически полными часто оказываются такие языки, которые с самого начала совершенно не предназначались для написания программ. Так, книга, которую вы читаете, подготовлена с помощью языка разметки TEX, созданного Дональдом Кнутом; этот язык в основном состоит из команд, задающих особенности текста, такие как форма шрифта, размеры и расположение заголовков и иллюстраций и т. п., то есть он изначально не предназначен для программирования; тем не менее язык TEX алгоритмически полон, хотя это знают далеко не все его пользователи.

1.6.5. Текст как формат данных. Кодировки

Текст как таковой, в общезначимом смысле этого слова, на каком бы языке он ни был написан, может быть самыми разными способами представлен в памяти компьютера (в виде содержимого ячеек памяти) и в виде файла на диске. Читателю, возможно, уже приходилось сталкиваться с тем, как редактор текстов при сохранении файла предлагает

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:	SPC	*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	`	j	t	~
7:	%	/	9	C	M	W	a	k	u	
8:	&	0	:	D	N	X	b	l	v	
9:	?	1	;	E	O	Y	c	m	w	

Рис. 1.13. Отображаемые символы ASCII и их десятичные коды

выбрать, в каком формате его сохранять — в собственном формате данного конкретного текстового редактора или же в формате, предназначенном для последующего прочтения какой-то другой программой, чаще всего каким-то другим редактором текстов.

Среди всех возможных представлений текста особое место занимает формат, именуемый по-английски *ASCII-text* или *plain text*; последнее может быть приблизительно переведено как «простой текст», «чистый текст» или «плоский текст», тогда как аббревиатура ASCII означает *American Standard Code for Information Interchange*; это название обозначает кодовую таблицу, которая изначально предназначалась для использования в телеграфии на телетайпах[56]. Таблица ASCII была зафиксирована в 1963 году; на тот момент восьмибитные байты ещё не получили массового распространения, поэтому создатели ASCII отвели на представление одного символа минимально возможное количество бит, позволявшее отличать друг от друга все символы, которые на тот момент были сочтены необходимыми. Исходное задание на разработку стандарта кодовой таблицы подразумевало возможность кодирования 26 заглавных и 26 строчных букв латинского алфавита, 10 арабских цифр, а также некоторого количества знаков препинания (включая, кстати, обыкновенный пробел). К тому же было необходимо предусмотреть несколько так называемых *управляющих кодов*, которые не обозначали никаких символов, а использовались, например, для управления телетайпом, находящимся по ту сторону линии связи; стандартной была практика отправки сообщения на телетайп, работающий без контроля со стороны людей, например, ночью в запертом здании.

В 64 кодовых значения уложиться не удалось (в самом деле, одних только букв и цифр будет уже 62), поэтому было принято решение использовать семибитную кодировку, то есть для представления одного символа (или управляющего кода) использовать семь двоичных знаков.

[56] С этими устройствами мы уже сталкивались в § 1.4.1.

Всего различных кодов при этом получается $2^7 = 128$. Первым тридцати двум из них, от 0 до 31, отвели роль управляющих, таких как *перевод строки* (10), *возврат каретки* (13), *табуляция* (9) и другие. Код 32 присвоили символу «пробел»; дальше в таблице (см. рис. 1.13) идут знаки препинания и математические символы, занимающие коды с 33 по 47; коды от 48 до 57 соответствуют арабским цифрам от 0 до 9, так что если из кода цифры вычесть 48, мы всегда получим её численное значение (этим мы ещё воспользуемся).

Заглавные латинские буквы располагаются в ASCII-таблице в позициях с 65 по 90 в алфавитном порядке, а позиции с 97 по 122, опять-таки в алфавитном порядке, заняты строчными буквами. Как несложно убедиться, буквы расположены в таблице так, чтобы двоичное представление заглавной буквы отличалось от двоичного представления строчной буквы ровно одним битом (шестым, то есть предпоследним). Так было сделано специально, чтобы облегчить приведение всех символов обрабатываемого текста к одному регистру, например, для выполнения регистро-независимого поиска.

Оставшиеся свободными позиции заняты разнообразными символами, которые на тот момент показались членам рабочей группы, создававшей таблицу, более полезными, чем другие. Последний отображаемый символ ASCII-таблицы имеет код 126 (это тильда, «~»), а код 127 считается управляющим, как и коды от 0 до 31; этот код, называемый RUBOUT, был изначально предназначен для вычёркивания символа. Дело в том, что в те времена символы часто представлялись пробитыми отверстиями в перфолентах, причём пробитое отверстие соответствовало единице, а непробитое — нулю; перфолента, предназначенная для хранения семибитного текста, имела как раз семь позиций в ширину, то есть каждый символ представлялся ровно одной строчкой из отверстий. Двоичное представление числа 127 состоит из семи единиц, то есть если в любой строке перфоленты «пробить» этот код, получится семь отверстий, причём вне всякой зависимости от того, что там было изначально. При прочтении перфоленты код 127 надлежало пропускать, предполагая, что ранее в этом месте находился символ, который позже был вычеркнут.

Массовый переход к использованию в компьютерах восьмибитных байтов стал причиной возникновения в программировании стойкой ассоциации *байта* с *символом*, поскольку, естественно, для хранения одного ASCII-кода стали использовать байт. В то же время появление восьмого бита позволило создать ряд разнообразных расширений ASCII-таблицы, в которых использовались коды из области 128–255. В частности, в разное время и в разных системах использовалось пять (!) различных расширений ASCII, предусматривающих русские буквы (кириллицу). Исторически первой из них была кодировка КОИ8 (*код обмена информацией восьмибитный*), которая вошла в употребление ещё в середине

1970-х годов на машинах серии ЕС ЭВМ. Основным недостатком этой кодировки является несколько «странный» порядок букв, совершенно не похожий на алфавитный: «ЮАБЦДЕФГХ...» Это затрудняет сортировку строк, требуя лишнего преобразовательного шага, тогда как если символы в таблице располагаются в алфавитном порядке, сортировку можно выполнять, просто сравнивая коды символов в арифметическом смысле.

С другой стороны, именно такой, а не иной порядок букв в таблице КОИ8 имеет вполне определённую причину, которая становится ясна, если выписать строки полученной расширенной ASCII-таблицы по 16 элементов в строке. Оказывается, кириллические буквы в КОИ8 находятся во второй половине 256-кодовой таблицы ровно в тех же позициях, в которых в первой половине таблицы (то есть в исходной таблице ASCII) располагаются их латинские аналоги. Дело в том, что раньше часто встречались (да и до сих пор ещё встречаются) ситуации, когда у текстовых данных кто-то принудительно «отбрасывает» восьмой (старший) бит. КОИ8 — единственная из русских кодировок, которая в такой ситуации сохраняет читаемость. Например, словосочетание «`добрый день`» при отбрасывании восьмого бита превратится в «`DOBRYJ DENX`»; читать такой текст не очень удобно, но всё же можно. Именно поэтому кодировка КОИ8 долгое время считалась единственной допустимой в телекоммуникациях и сдала позиции только под натиском Unicode, о котором речь пойдёт ниже. Отметим, что русский алфавит, содержащий 33 буквы, «чуть-чуть не вписался» в две подряд идущие строки по 16 кодов; «не повезло» в этот раз букве «ё», которую в КОИ8 отправили «на выселки», приписав её строчному и заглавному варианту коды $\texttt{A3}_{16}$ и $\texttt{B3}_{16}$, тогда как все остальные буквы занимают непрерывную область кодов от $\texttt{C0}_{16}$ до $\texttt{FF}_{16}$.

В эпоху MS-DOS, то есть в 1980-е годы и в начале 1990-х, на персональных компьютерах самой популярной кодировкой кириллицы была так называемая «альтернативная» русская кодировка, известная также под названием cp866 (кодовая страница № 866). В ней, надо сказать, символы тоже располагались не слишком удачно: алфавитный порядок сохранялся, то есть для сортировки её можно было использовать без промежуточных преобразований (за исключением буквы «ё», которой опять не повезло), но при этом заглавные русские буквы в cp866 шли подряд, тогда как между первыми шестнадцатью и вторыми шестнадцатью строчными вклинились коды для 48 символов псевдографики. Интересно, что эта кодировка до сих пор встречается — например, в некоторых версиях Windows она применяется при работе с консольными приложениями; она же применялась в качестве единственной кодировки кириллицы в системах семейства OS/2.

Во время массового перехода пользователей на системы линейки Windows многие были неприятно удивлены тем, что в этих системах

используется ещё одна кодировка кириллицы, совершенно не похожая ни на cp866, ни на КОИ8. Это была кодировка cp1251, содержавшая заодно символы из других кириллических алфавитов — украинского, белорусского, сербского, македонского и болгарского, но при этом её создатели забыли, например, про казахский и многие другие неславянские языки, использующие кириллицу. Отметим, что букве «ё» в очередной раз не повезло — она оказалась вне основной кодовой области.

Компьютеры Apple Macintosh тоже использовали и до сих пор используют свою собственную кодировку, так называемую MacCyrillic. Следует отметить и ещё один «стандарт», ISO 8859-5, чья кодовая таблица отличается от всех перечисленных; впрочем, этот стандарт, созданный очередным комитетом с совершенно непонятными целями, никогда нигде не применялся.

Можно выделить несколько основных свойств текстового представления данных:

- любой фрагмент текста является текстом;
- объединение нескольких текстов тоже является текстом;
- представление любого символа занимает ровно один байт, что, в частности, позволяет извлечь из текста любой символ по его номеру, не просматривая при этом весь текст;
- представление текста не зависит от архитектуры компьютера, от размера машинного слова, от порядка байтов в представлении целых чисел и от других особенностей, то есть является универсальным для всех компьютеров в мире;
- данные в текстовом формате (текст) могут быть прочитаны человеком на любом компьютере с помощью тысяч различных программ.

С ростом количества известных и используемых в компьютерной обработке информации символов возникла потребность навести в этой области какой-то порядок, в связи с чем в 1987 году был создан реестр Unicode, который часто (и ошибочно) принимают за *кодировку*. На самом деле основой Unicode является так называемое «универсальное множество символов» (*universal character set*, UCS), то есть попросту реестр известных символов, в котором они снабжены номерами; что касается кодировок, то таковых на основе Unicode создано по меньшей мере четыре: UCS-2, UCS-4 (она же UTF-32), UTF-8 и UTF-16.

Кодировки UCS-2 и UCS-4 предполагают использование для хранения одного символа соответственно двух байтов и четырёх. Поскольку к настоящему моменту в реестре Unicode содержится больше 110 000 символов, кодировка UCS-2 с задачей представления любых символов не справляется (два байта могут хранить всего 65536 различных значений, а этого уже давно не хватает) и к настоящему времени считается устаревшей; что касается UCS-4, то её стараются не применять из-за слишком большого и непроизводительного расхода памяти: в самом деле,

для хранения большинства символов достаточно одного байта, а больше трёх не нужно вообще никогда и вряд ли когда-нибудь потребуется: странно было бы ожидать, что количество символов в реестре Unicode когда-нибудь превысит 16 с лишним *миллионов*, ведь в имеющиеся сейчас 110 с небольшим *тысяч* уже вошли не только все применяющиеся в мире азбуки, включая иероглифические, но и всевозможная экзотика вроде иероглифов древнего Египта и вавилонской клинописи.

Следует отметить, что многобайтовые кодировки полностью лишены большинства из вышеперечисленных достоинств текстового представления данных; их вообще можно рассматривать как *текстовые данные* с очень большой натяжкой: в самом деле, они даже *зависят от порядка байтов в представлении целых чисел*; в связи с этим стандарты требуют, чтобы в файле, содержащем «текст» в какой-либо из кодировок на основе Unicode, первые два (или четыре) байта представляли псевдосимвол с кодом $\mathtt{FEFF}_{16}$, что позволяет программе, читающей этот текст, понять, какой порядок байтов использовался при его записи. В результате объединение двух таких «текстов» уже не обязательно представляет собой текст, ведь два исходных текста могут оказаться в разном порядке байтов, а пресловутый $\mathtt{FEFF}_{16}$, встреченный в середине текста, не воспринимается как указание на смену порядка байтов; фрагмент «текста» в многобайтовых кодировках тем более не обязан быть корректным текстом.

В качестве единственного исключения можно рассматривать кодировку UTF-8, которая продолжает использовать для кодирования текстов именно последовательность байтов, а не многобайтовых целых чисел. Более того, первые 127 кодов в ней совпадают с ASCII и считается, что если байт начинается с нулевого бита, то этот байт соответствует *однобайтовому* коду символа. Таким образом, обычный текст в традиционном формате ASCII оказывается корректным с точки зрения UTF-8 и интерпретируется одинаково как в ASCII, так и в UTF-8. Для символов, не вошедших в ASCII, UTF-8 предполагает использование последовательностей из двух, трёх или четырёх байтов, а при необходимости — и пяти, и шести, хотя пока это не нужно: в Unicode просто нет стольких символов.

При этом в UTF-8 байт, начинающийся с битов 110, означает, что мы имеем дело с символом, код которого представлен двумя байтами; первый байт трёхбайтного символа начинается с битов 1110; первый байт четырёхбайтного символа — с битов 11110. Дополнительные байты (второй и последующие в представлении данного символа) начинаются с битов 10. Таким образом, для представления полезной информации в двухбайтовом коде используется 11 битов из 16, в трёхбайтовом — 16 битов из 24, в четырёхбайтовом — 21.

Текст, представленный в UTF-8, не зависит от порядка байтов в целых числах; фрагмент такого текста остаётся корректным, за исключением, возможно, «кусочков» представления одного символа в начале и в конце; объединение двух и более текстов в UTF-8 остаётся текстом в UTF-8. Очевидный недостаток у UTF-8 только один: поскольку для одного символа используются коды переменной длины, для извлечения символа по его номеру приходится просмотреть весь предшествующий ему текст. Кроме того, тексту в UTF-8 присущ недостаток, общий для всех кодировок, основанных на Unicode: никто не может вам гарантировать, что на компьютере, где кто-то попытается прочитать текст, сформированный в UTF-8, в используемом шрифте найдутся корректные изображения для всех символов, которые вы используете. В самом деле, создать

шрифт, представляющий *сто десять тысяч* разнообразных символов, не так просто.

Самой странной из всех кодировок Unicode представляется кодировка UTF-16: она использует двухбайтные числа, но *иногда* для представления одного символа применяются два таких числа. Эта кодировка не имеет никаких достоинств в сравнении с другими, но обладает всеми их недостатками одновременно; в частности, если нас в принципе устраивает переменная длина кода символа, то гораздо лучше будет использовать UTF-8, ведь она не зависит от порядка байтов и плюс к тому совместима с ASCII, а никаких недостатков по сравнению с UTF-16 у неё нет. При этом именно UTF-16 используется в системах линейки Windows, начиная с Windows-2000; это обусловлено тем, что в более ранних системах линейки, начиная с Windows NT, использовалась UCS-2, тоже предполагающая двухбайтные коды; как мы уже знаем, двух байтов не хватило для представления всех символов Unicode.

Так или иначе, представление текста, основанное на кодировках Unicode, за исключением UTF-8, вообще невозможно рассматривать как *текстовые данные* в исходном программистском смысле; с UTF-8 в этом плане дела обстоят несколько лучше, однако если ASCII-символы заведомо присутствуют и корректно отображаются в любой операционной среде, то о других символах этого сказать нельзя. Поэтому **в ряде случаев использование символов, не вошедших в набор ASCII, считается нежелательным или вообще запрещается**; например, это касается текстов компьютерных программ на любых языках программирования.

1.6.6. Бинарные и текстовые данные

При обработке самой разной информации часто возникает выбор, в каком виде представить эту информацию: в *текстовом*, то есть в таком виде, в котором её сможет прочитать человек, или в *бинарном*; под бинарным в принципе подразумевается любое представление, отличное от текстового, но обычно данные при этом хранятся в файлах и передаются по компьютерным сетям в том же виде, в котором они представлены в ячейках памяти компьютера во время их обработки программами.

Для того, чтобы уяснить разницу между текстовым и бинарным представлением данных, мы проведём небольшой эксперимент, и в его проведении нам поможет программа `hexdump`, обычно имеющаяся на любой Unix-системе. Для начала создадим[57] два файла, в которые запишем последовательность чисел 1000, 2001, 3002, ..., 100099 — то есть всего 100 чисел, каждое следующее больше предыдущего на 1001. Хитрость в том, что эти числа мы в один файл запишем в текстовом представлении, по одному в строке, а в другой файл — в виде четырёхбайтных целых чисел, точно так, как они представляются в памяти компьютера при

[57] Эти файлы вы найдёте в архиве примеров к этой книге; программы на Паскале, которые можно использовать для генерации этих двух файлов, будут приведены в § 2.12.

выполнении всевозможных расчётов. Файлы мы назовём `numbers.txt` и `numbers.bin`. Для начала посмотрим, что у нас получилось:

```
avst@host:~/work$ ls
numbers.bin  numbers.txt
avst@host:~/work$ ls -l
total 8
-rw-r--r-- 1 avst avst 400 Мар 23 16:21 numbers.bin
-rw-r--r-- 1 avst avst 592 Мар 23 16:21 numbers.txt
avst@host:~/work$
```

Как видим, двоичный файл получился размером ровно 400 байт, что вполне понятно: мы записали в него 100 чисел по 4 байта каждое. Текстовый файл получился размером 592 байта, то есть чуть больше; он мог бы быть и меньше, если бы мы записали в него, например, числа от 1 до 100, то есть такие числа, представление которых состоит из меньшего количества цифр. Посмотрим, откуда взялось число 592. Числа от 1000 до 9008 имеют в своём текстовом представлении по четыре цифры, и таких чисел девять; числа от 10009 до 99098 имеют по пять цифр каждое, и таких чисел у нас девяносто; последнее число нашей последовательности, 100099, представлено шестью цифрами, и такое число у нас одно. Кроме того, каждое число записано на отдельной строке, а строки, как мы знаем, разделены *символом перевода строки* (символ с кодом 10); поскольку чисел всего сто, символов перевода строки тоже сто — после каждого числа (в том числе и после последнего; вообще считается, что корректный текстовый файл должен заканчиваться корректной строкой, т. е. последним символом в нём должен быть как раз символ перевода строки). Итого имеем $4 \cdot 9 + 5 \cdot 90 + 6 \cdot 1 + 100 = 36 + 450 + 6 + 100 = 592$, что мы и увидели в выдаче команды `ls -l`.

При этом файл `numbers.txt` можно выдать на печать, например:

```
avst@host:~/work$ cat numbers.txt
1000
2001
3002
```

[...]

```
98097
99098
100099
avst@host:~/work$
```

Его также можно открыть в любом текстовом редакторе — если это сделать, мы увидим всё те же сто чисел, точнее, их десятичное представление, и сможем при желании внести изменения. Иначе говоря,

мы можем работать с файлом ***numbers.txt****, применяя наши обычные средства работы с текстом.*

Иное дело — файл `numbers.bin`. При попытке выдать его на печать мы увидим откровенную белиберду, что-то вроде такого:

```
avst@host:~/work$ cat numbers.bin
Хя.
ёu^G0#'+K.т2.6.:>xBaFJJ3NRVHYв]юа.ei{mdqMu6y}Язц.~gP9.".
.T.щЁф....цjrSк<o%свВзЮчиБ.ФЙHmPVЖ?З(ЧЗЦл.pY!B%+)
ЩОФ4о8.<.@DsH\LEP.TX\И_pc.g.kovs_wH{1avst@host:~/work$
```

— причём нам ещё повезёт, если наш терминал не придёт в негодность, случайно получив среди всей этой белиберды управляющие последовательности, перепрограммирующие его поведение. Впрочем, привести терминал в чувство можно в любой момент командой `reset`. Так или иначе, мы видим, что `numbers.bin` не предназначен для человека; то же самое можно сказать про любой файл, не являющийся текстовым.

Как устроен файл `numbers.bin`, нам поможет лучше понять программа `hexdump`, которая позволяет вывести на печать значения байов заданного файла (в шестнадцатеричной нотации), а при указании дополнительного ключика `-C` показать ещё и соответствующие байтам символы — точнее, те из них, которые можно отобразить на экране.

```
avst@host:~/work$ hexdump -C numbers.bin
00000000  e8 03 00 00 d1 07 00 00  ba 0b 00 00 a3 0f 00 00 |Х...я.......ё...|
00000010  8c 13 00 00 75 17 00 00  5e 1b 00 00 47 1f 00 00 |....u...^...G...|
00000020  30 23 00 00 19 27 00 00  02 2b 00 00 eb 2e 00 00 |0#...'...+..К...|
00000030  d4 32 00 00 bd 36 00 00  a6 3a 00 00 8f 3e 00 00 |т2...6...:...>..|
00000040  78 42 00 00 61 46 00 00  4a 4a 00 00 33 4e 00 00 |xB..aF..JJ..3N..|

[...]

00000170  a4 6b 01 00 8d 6f 01 00  76 73 01 00 5f 77 01 00 |.k...o..vs.._w..|
00000180  48 7b 01 00 31 7f 01 00  1a 83 01 00 03 87 01 00 |H{..1...........|
00000190
avst@host:~/work$
```

Самая левая колонка здесь — это *адреса*, то есть порядковые номера байтов от начала файла; они идут с интервалом 10_{16}, поскольку каждая строка содержит изображение шестнадцати очередных байтов. Дальше идут, собственно, сами байты, а в колонке справа — символы, коды которых содержатся в соответствующих байтах (или точка, если символ отобразить нельзя). Посмотрев в эту колонку, мы увидим уже знакомую нам белиберду.

Вернувшись к содержимому байтов, вспомним, что числа у нас по условию задачи, во-первых, четырёхбайтные, и, во-вторых, наш компьютер относится к классу *little-endian*, то есть младший байт числа идёт первым. Взяв первые четыре байта из нашего дампа, мы видим

`e8 03 00 00`; переставив их в обратном порядке, получаем `000003e8`; переводя из шестнадцатеричной системы в десятичную, имеем (с учётом того, что `E` обозначает 14): $3 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0 = 768 + 224 + 8 = 1000$, это и есть, как мы помним, первое число, записанное в файл. На всякий случай проделаем то же самое для последних четырёх байтов нашего дампа: `03 87 01 00`; переставив их, получаем $\texttt{00018703}_{16} = 1 \cdot 16^4 + 8 \cdot 16^3 + 7 \cdot 16^2 + 0 \cdot 16^1 + 3 \cdot 16^0 = 65536 + 8 \cdot 4096 + 7 \cdot 256 + 3 = 100099$; как видим, всё сходится.

Для полноты картины применим `hexdump` к текстовому файлу:

```
avst@host:~/work$ hexdump -C numbers.txt
00000000  31 30 30 30 0a 32 30 30  31 0a 33 30 30 32 0a 34 |1000.2001.3002.4|
00000010  30 30 33 0a 35 30 30 34  0a 36 30 30 35 0a 37 30 |003.5004.6005.70|
00000020  30 36 0a 38 30 30 37 0a  39 30 30 38 0a 31 30 30 |06.8007.9008.100|
00000030  30 39 0a 31 31 30 31 30  0a 31 32 30 31 31 0a 31 |09.11010.12011.1|
00000040  33 30 31 32 0a 31 34 30  31 33 0a 31 35 30 31 34 |3012.14013.15014|
```

[...]

```
00000230  0a 39 36 30 39 35 0a 39  37 30 39 36 0a 39 38 30 |.96095.97096.980|
00000240  39 37 0a 39 39 30 39 38  0a 31 30 30 30 39 39 0a |97.99098.100099.|
00000250
avst@host:~/work$
```

В правой колонке находится вполне читаемый текст, что и понятно, ведь файл как раз текстовый; точками здесь программа `hexdump` была вынуждена заменить только символы перевода строки. Просматривая значения байтов, начиная с первого, мы видим $\texttt{31}_{16}$ (десятичное 49) — ASCII-код цифры 1, затем три раза $\texttt{30}_{16}$ — ASCII-код нуля, потом $\texttt{0A}_{16}$ (десятичное 10) — пресловутый символ перевода строки, и так далее.

Такое представление удобно для человека, но программам работать с ним труднее; для проведения каких бы то ни было вычислений числа приходится переводить в то представление, с которым может справиться центральный процессор. Конечно, такое преобразование, как мы увидим позднее, выполнить совсем не трудно, просто о нём необходимо помнить и, конечно же, понимать, что, когда и в каком виде у нас представлено.

1.6.7. Машинный код, компиляторы и интерпретаторы

Как мы уже говорили, практически все современные цифровые вычислительные машины работают по одному и тому же принципу. Вычислительное устройство (собственно сам компьютер) состоит из ***центрального процессора***, ***оперативной памяти*** и ***периферийных устройств***. В большинстве случаев все эти компоненты подключаются к ***общей шине***.

Оперативная память состоит из одинаковых ***ячеек памяти***, каждая из которых имеет свой уникальный номер, называемый ***адресом***.

Ячейка содержит несколько (чаще всего — восемь) двоичных разрядов, каждый из которых может находиться в одном из двух состояний, обычно обозначаемых как «ноль» и «единица». Это позволяет ячейке как единому целому находиться в одном из 2^n состояний, где n — количество разрядов в ячейке; так, если разрядов восемь, то возможных состояний ячейки будет $2^8 = 256$, или, иначе говоря, ячейка может «помнить» число от 0 до 255.

В центральном процессоре имеется некоторое количество ***регистров*** — схем, напоминающих ячейки памяти; поскольку регистры находятся непосредственно в процессоре, они работают очень быстро, но их количество ограничено, так что использовать регистры следует для хранения самой необходимой информации. Процессор обладает способностью копировать данные из оперативной памяти в регистры и обратно, производить над содержимым регистров арифметические и другие операции; в некоторых случаях операции можно производить и непосредственно с данными в ячейках памяти, не копируя их содержимое в регистры[58].

Количество информации, которое может обработать процессор в один приём (за одну команду), называется ***машинным словом***. Размер большинства регистров в точности равен машинному слову. В современных системах машинное слово, как правило, больше, чем ячейка памяти; так, машинное слово процессора Pentium составляет 32 бита, то есть четыре восьмибитовые ячейки памяти.

Здесь необходимо сделать одно важное замечание. Процессор Pentium и его современные «наследники», с которыми мы собираемся работать, являются очередными представителями *линейки процессоров* x86, и ранние представители этой линейки (до 8028б включительно) были 16-разрядными, то есть их машинное слово составляло 16 бит. Программисты, работающие с этими процессорами на уровне машинных команд, привыкли называть «словом» именно два байта информации, а четыре байта называли «двойным словом». Когда с выходом очередного процессора размер слова удвоился, программисты не стали менять привычную терминологию, что порождает определённую путаницу. К этому вопросу нам придётся вернуться, когда мы доберёмся до программирования на уровне команд центрального процессора.

Как мы уже знаем (см. § 1.3.2, стр. 55), программа записывается в оперативную память в виде цифровых кодов, обозначающих те или иные операции, а специальный регистр процессора, который называется ***счётчиком команд*** (англ. *instruction pointer*, то есть «указатель на инструкцию») определяет, из какого места памяти нужно брать

[58] Наличие или отсутствие такой возможности зависит от конкретного процессора; так, процессоры Pentium могут, минуя регистры, прибавить заданное число к содержимому заданной ячейки памяти и произвести некоторые другие операции, тогда как процессоры SPARC, применявшиеся в компьютерах фирмы Sun Microsystems, умели только копировать содержимое ячейки памяти в регистр или, наоборот, содержимое регистра в ячейку памяти, но никаких других операций над ячейками памяти выполнять могли.

инструкции для выполнения. Процессор выполняет ***цикл обработки команд*** — то есть извлекает из памяти код очередной команды, увеличивает счётчик команд, дешифрует извлечённый код, исполняет предписанные действия, снова извлекает из памяти очередную команду, и так до бесконечности.

Представление программы, состоящее из кодов машинных команд и, как следствие, «понятное» центральному процессору, называется ***машинным кодом***. Процессор легко может дешифровать такие коды команд, но человеку их запомнить очень трудно, тем более что во многих случаях нужное число приходится *вычислять*, подставляя в определённые места кодовые цепочки двоичных битов. Вот, например, два байта, записываемые в шестнадцатеричной системе как `01 D8` (соответствующие десятичные значения — 1, 216) обозначают на процессорах Pentium команду «взять число из регистра `EAX`, прибавить к нему число из регистра `EBX`, результат сложения поместить обратно в регистр `EAX`». Запомнить два числа `01 D8` несложно, но ведь разных команд на процессоре Pentium — несколько сотен, да к тому же здесь сама команда — только первый байт (`01`), а второй (`D8`) нам придётся вычислить в уме, вспомнив (или узнав из справочника), что младшие три бита в этом байте обозначают первый регистр (первое слагаемое, а также и место, куда следует записать результат), следущие три бита обозначают второй регистр, а самые старшие два бита здесь должны быть равны единицам, что означает, что оба операнда являются регистрами. Зная (или, опять же, подсмотрев в справочнике), что номер регистра `EAX` — 0, а номер регистра `EBX` — 3, мы теперь можем записать двоичное представление нашего байта: `11 011 000` (пробелы вставлены для наглядности), что и даёт в десятичной записи 216, а в шестнадцатеричной — искомое `D8`.

Если нам потребуется освежить в памяти кусочек нашей программы, написанный два дня назад, то чтобы его прочитать, нам придётся вручную раскладывать байты на составляющие их биты и, сверяясь со справочником, вспоминать, что же какая команда делает. Если программиста заставить составлять программы вот таким вот способом, ничего полезного он не напишет за всю свою жизнь, тем более что в любой, даже самой небольшой, но практически применимой программе таких команд будет несколько тысяч, ну а самые большие программы состоят из *десятков миллионов* машинных команд.

При работе с языками программирования высокого уровня, такими как Паскаль, Си, Лисп и др., программисту предоставляется возможность написать программу в виде, понятном и удобном для человека, а не для центрального процессора. Процессор такую программу выполнить уже не может, и чтобы всё-таки заставить исполниться программу, написанную на языке высокого уровня, приходится прибегнуть к одному из двух возможных способов *трансляции* программы. Эти два способа называются *компиляцией* и *интерпретацией*.

В первом случае применяют ***компилятор*** — программу, принимающую на вход текст программы на языке программирования высокого уровня и выдающую эквивалентный машинный код[59]. Например, в следующей части книги мы будем писать программы на Паскале; набрав текст программы (так называемый ***исходный текст***) и сохранив его в файле, мы будем запускать *компилятор Паскаля*, который, прочитав текст нашей программы, либо выдаст сообщения об ошибках, либо, если программа написана в соответствии с правилами языка Паскаль, создаст её эквивалент в виде исполняемого файла, в котором будет находиться машинный код нашей программы. Запуская нашу программу, мы потребуем от операционной системы загрузить этот машинный код в оперативную память и передать ему управление, в результате чего процессор выполнит те действия, которые мы описали в виде текста на языке Паскаль.

Следует подчеркнуть, что компилятор — это тоже программа, написанная на каком-то языке программирования; в частности, наш компилятор Паскаля сам, как это ни странно, написан на Паскале, а для компиляции каждой следующей версии его создатели используют предыдущую версию своего же компилятора.

Второй способ исполнения программ, написанных на языках высокого уровня, называется *интерпретацией*. Программа-интерпретатор прочитывает из файла исходный текст и выполняет предписанные этим текстом действия шаг за шагом, ничего никуда, вообще говоря, не переводя. Современные интерпретаторы обычно для удобства и увеличения скорости выполнения создают какое-то своё внутреннее представление нашей программы, но с машинным кодом такое представление не имеет ничего общего.

Отметим, что из одного школьного учебника в другой кочует совершенно бредовая фраза, что якобы интерпретатор *переводит программу в машинный код пошагово и тут же этот код выполняет*. Если вам что-то подобное говорили или вы сами такое прочитали в какой-нибудь очередной не пойми кем написанной книжке (пусть даже имеющей гриф министерства образования, бывает и такое) — не верьте, вас в очередной раз обманывают! На современных машинах и при работе под управлением современных операционных систем так вообще невозможно сделать, поскольку даже если мы сформируем какой-то машинный код, записать его мы можем только в область памяти, предназначенную для хранения данных (как мы увидим позже, область машинного кода во время работы программы не может быть изменена), а передать управление на этот код мы не сможем, так как исполнение команд из области, предназначенной для данных, тоже запрещено. В общем, даже

[59] Вообще говоря, компилятор — это программа, переводящая программы с одного языка на другой; перевод на язык машинного кода — это лишь частный случай, хотя и очень важный.

если бы мы хотели так сделать, то ничего бы не получилось — но самое интересное, что так делать вовсе не нужно! Интерпретатор сам по себе является *программой*, и он может просто выполнить нужные действия, никуда, ни в какой код их не переводя. Тот странный человек, который первым придумал фразу про пошаговый перевод в машинный код с немедленным исполнением, сам, очевидно, никогда никаких программ не писал, иначе такая сумасшедшая идея просто не пришла бы ему в голову.

С одним интерпретатором мы уже сталкивались: это *командный интерпретатор*, обрабатывающий команды, которые мы набираем в командной строке. Как мы видели в § 1.4.13, на языке этого интерпретатора, который называется Bourne Shell, можно писать вполне настоящие программы. Другие интерпретаторы мы пока рассматривать не будем, но вообще интерпретируемое исполнение характерно для широкого круга языков программирования, таких как Перл, Пайтон, Руби, Лисп и многих других.

Интересно заметить, что в наше время границы между компиляцией и интерпретацией постепенно размываются. Многие компиляторы переводят программу не в машинный код, а в некое промежуточное представление, обычно называемое «байт-кодом»; именно так работают компиляторы языков Java и C#. Во многих случаях компиляторы порождают такой код, который уже во время исполнения осуществляет интерпретацию какой-то части промежуточного представления программы. С другой стороны, интерпретаторы с целью повышения эффективности работы *тоже переводят программу в промежуточное представление*, которое, правда, затем выполняют сами. Существуют компиляторы, вроде бы создающие отдельный исполняемый файл, но при внимательном рассмотрении этого файла оказывается, что в него целиком включён интерпретатор внутреннего представления программы плюс само это представление.

Так или иначе, при компилируемом исполнении часто используются элементы интерпретации, а при интерпретируемом — элементы компиляции, и встаёт вопрос о том, где между этими двумя подходами провести границу и существует ли такая граница вообще. Мы рискнём предложить довольно простой ответ на этот вопрос, позволяющий в каждом конкретном случае определённо сказать, имеет здесь место компилируемое или интерпретируемое исполнение. **Интерпретатор во время исполнения интерпретируемой программы сам вынужден находиться в памяти**, тогда как компилятор нужен лишь на этапе компиляции, а исполняться программа может без его участия.

Программирование на языках высокого уровня удобно, но, к сожалению, не всегда применимо. Причины могут быть самые разные. Например, язык высокого уровня может не учитывать некоторые особен-

ности конкретного процессора, либо программиста может не устраивать тот конкретный способ, которым компилятор реализует те или иные конструкции исходного языка с помощью машинных кодов. В этих случаях приходится отказаться от языка высокого уровня и составить программу в виде *конкретной последовательности машинных команд*. Однако, как мы уже видели, составлять программу непосредственно в машинных кодах очень и очень сложно. И здесь на помощь приходит программа, называемая ***ассемблером***.

Ассемблер — это частный случай компилятора: программа, принимающая на вход текст, содержащий условные обозначения машинных команд, удобные для человека, и переводящий эти обозначения в последовательность соответствующих кодов машинных команд, понятных процессору. В отличие от самих машинных команд, их условные обозначения, называемые также ***мнемониками***, запомнить сравнительно легко. Так, команда из приведённого ранее примера, код которой, как мы с некоторым трудом выяснили, равен `01 D8`, в условных обозначениях[60] выглядит так:

```
add  eax, ebx
```

Здесь нам уже не надо заучивать числовой код команды и вычислять в уме обозначения операндов, достаточно запомнить, что словом `add` обозначается сложение, причём в таких случаях всегда первым после обозначения команды стоит первое слагаемое (не обязательно регистр, это может быть и область памяти), вторым — второе слагаемое (это может быть и регистр, и область памяти, и просто число), а результат всегда заносится на место первого слагаемого. Язык таких условных обозначений (мнемоник) называется ***языком ассемблера***.

Программирование на языке ассемблера коренным образом отличается от программирования на языках высокого уровня. На языке высокого уровня (на том же Паскале) мы задаём лишь общие указания, а компилятор волен сам выбирать, каким именно способом их исполнить — например, какими регистрами и ячейками памяти воспользоваться для хранения промежуточных результатов, какой применить алгоритм для выполнения какой-нибудь нетривиальной инструкции и т. д. С целью оптимизации быстродействия компилятор может переставить инструкции местами, заменить одни на другие — лишь бы результат остался неизменным. **В программе на языке ассемблера мы совершенно однозначно и недвусмысленно указываем, из каких машинных команд будет состоять наша программа, и никакой свободы ассемблер (в отличие от компилятора языка высокого уровня) не имеет**.

[60] Здесь и далее используются условные обозначения, соответствующие ассемблеру NASM, если не сказано иное; подробно ассемблер NASM будет рассматриваться в следующем томе нашей книги.

В отличие от машинных кодов, мнемоники доступны для человека, то есть программист может работать с мнемониками без особого труда, но это не означает, что программировать на языке ассемблера просто. Действие, на описание которого мы бы потратили один оператор языка высокого уровня, может потребовать десятка, если не сотни строк на языке ассемблера, а в некоторых случаях и больше. Дело тут в том, что компилятор языка высокого уровня содержит большой набор готовых «рецептов» решения часто возникающих небольших задач и предоставляет все эти «рецепты» программисту в виде удобных высокоуровневых конструкций; ассемблер же ничего подобного не содержит, так что в нашем распоряжении оказываются только возможности процессора.

Интересно, что для одного процессора может существовать несколько разных ассемблеров. На первый взгляд это кажется странным, ведь не может же один и тот же процессор работать с разными системами машинных кодов (так называемыми ***системами команд***). В действительности ничего странного здесь нет, достаточно вспомнить, что же такое на самом деле ассемблер. Система команд процессора, разумеется, не может измениться (если только не взять другой процессор). Однако для одних и тех же команд можно придумать разные обозначения; так, уже знакомая нам команда `add eax, ebx` в обозначениях, принятых в компании AT&T, будет выглядеть как `addl %ebx, %eax` — и мнемоника другая, и регистры не так обозначены, и операнды не в том порядке, хотя получаемый машинный код, разумеется, строго тот же самый — `01 D8`. Кроме того, при программировании на языке ассемблера мы обычно пишем не только мнемоники машинных команд, но и ***директивы***, представляющие собой прямые приказы ассемблеру. Следуя таким указаниям, ассемблер может зарезервировать память, объявить ту или иную метку видимой из других модулей программы, перейти к генерации другой секции программы, вычислить (прямо во время ассемблирования) какое-нибудь выражение и даже сам (следуя, разумеется, нашим указаниям) «написать» фрагмент программы на языке ассемблера, который сам же потом и обработает. Набор таких директив, поддерживаемых ассемблером, тоже может быть разным, как по возможностям, так и по синтаксису.

Поскольку ассемблер — это не более чем программа, написанная вполне обыкновенными программистами, никто не мешает другим программистам написать свою программу-ассемблер, что часто и происходит. Ассемблер NASM, который рассматривается во втором томе нашей книги — это один из ассемблеров, существующих для процессоров семейства 80x86; существуют и другие ассемблеры для этих процессоров.

Часть 2

Язык Паскаль и начала программирования

В этой части книги мы попробуем перейти от слов к делу и научиться азам написания компьютерных программ, для чего нам потребуется язык Паскаль. Изначально Паскаль был предложен швейцарским учёным Никлаусом Виртом в 1970 году в качестве языка для обучения программированию; им же был написан самый первый компилятор Паскаля.

Довольно неожиданным для начинающих оказывается утверждение, что в наше время невозможно однозначно определить, что́ же такое, собственно, язык Паскаль. Дело в том, что за десятилетия истории Паскаля самые разные люди и организации создавали свои собственные трансляторы этого языка и при этом вносили в язык разнообразные расширения и изменения; такие видоизменённые варианты языка программирования обычно называют ***диалектами***. К нынешнему времени понятие «язык Паскаль» настолько размыто, что всегда требует уточнений. Как исходное определение языка, данное Виртом, так и существующие стандарты Паскаля очень ограничены в своих возможностях, и на них, собственно говоря, давно никто не оглядывается.

Реализация, которую мы будем использовать в качестве учебного пособия, называется Free Pascal и отсчитывает свою историю с 1993 года; её автор Флориан Пол Клэмпфль (Florian Paul Klämpfl) начал разработку собственного компилятора Паскаля в ответ на заявление фирмы Борланд о прекращении развития линейки компиляторов Паскаля для MS-DOS. В настоящее время компилятор Free Pascal доступен для всех наиболее популярных операционных систем, включая Linux и FreeBSD (а также Windows, MacOS X, OS/2, iOS; на момент написания этой книги было заявлено о скором появлении поддержки для Android). Free Pascal поддерживает сразу несколько различных диалектов Паскаля (на

выбор программиста) и включает огромное количество разнообразных возможностей, пришедших из других версий Паскаля.

Как ни странно, изучать всё это разнообразие мы не станем; напротив, набор возможностей, которыми мы станем пользоваться на протяжении этой части книги, будет весьма ограниченным. Дело в том, что Free Pascal интересует нас не сам по себе, не в качестве инструмента для профессионального программирования (хотя он, вне всякого сомнения, может выступать в качестве такого инструмента), а лишь как учебное пособие, которое позволит нам освоить начала и основные приёмы императивного программирования. В дальнейшем нас ожидает знакомство с языками Си и Си++, и подходить к их изучению желательно с уже сформированными представлениями о структурном программировании, рекурсии, указателях и других базовых возможностях, которые используются в компьютерных программах; Паскаль как раз и позволит нам всё это изучить, но **весь** Паскаль (и тем более возможности Free Pascal во всём их многообразии) для этого нам не потребуются.

При желании читатель, несомненно, сможет самостоятельно довести своё знание как Free Pascal, так и других реализаций Паскаля до любого уровня, какого пожелает, воспользовавшись литературой и другими материалами, в изобилии представленными в Интернете; однако вполне возможно, что после знакомства с другими языками программирования использовать в качестве профессионального инструмента именно Паскаль читателю уже не захочется. Так или иначе, вопрос о выборе профессионального инструмента — пока что дело будущего.

Задачу, стоящую перед нами сейчас, можно сформулировать одной фразой: мы на самом деле изучаем не Паскаль, а *программирование* как таковое, Паскаль же интересует нас лишь как иллюстрация, не более того.

2.1. Первые программы

Напомним для начала кое-что из того, что уже обсуждалось во вводной части. Компьютер — точнее, его центральный процессор — может выполнять программу, представленную в машинном коде, но для человека писать машинный код практически нереально из-за грандиозной трудоёмкости такой работы. Поэтому программисты пишут программы в виде текста, соответствующего правилам того или иного *языка программирования*, а затем применяют программу-транслятор; это может быть либо *компилятор*, который переводит программу целиком в какое-то другое представление (например, в машинный код), либо *интерпретатор*, который ничего никуда не переводит, а просто выполняет шаг за шагом действия, предписанные текстом выполняемой

программы. Текст, написанный программистом в соответствии с правилами языка программирования, называют ***исходным текстом*** или ***исходным кодом*** программы.

Язык Паскаль, который мы начинаем изучать, обычно относят к ***компилируемым***; это означает, что в большинстве случаев при работе с Паскалем применяются именно компиляторы[1]. Сама программа на Паскале, как и едва ли не на любом языке программирования[2], представляет собой *текст* в ASCII-представлении, которое мы обсуждали в § 1.6.4. Следовательно, для написания программы нам придётся воспользоваться каким-нибудь из ***текстовых редакторов***; о некоторых из них мы рассказали в § 1.4.10. Результат мы сохраним в файле с именем, имеющем ***суффикс***[3] «`.pas`», который обычно означает текст программы на Паскале. Затем мы запустим компилятор, который в нашем случае называется `fpc` от слов *Free Pascal Compiler*, и если всё будет в порядке, то результатом наших упражнений станет исполняемый файл, который мы сможем запустить.

Для наглядности и удобства мы перед началом наших экспериментов создадим пустую директорию[4] и все эксперименты будем проводить в ней. В примерах диалогов с компьютером мы здесь и далее воспроизведём приглашение командной строки, состоящее из имени пользователя (`avst`), имени машины (`host`), текущей директории (напомним, что домашняя директория пользователя обозначается символом «`~`») и знака «`$`», традиционно используемого в приглашении. Итак, начнём:

```
avst@host:~$ mkdir firstprog
avst@host:~$ cd firstprog
avst@host:~/firstprog$ ls
avst@host:~/firstprog$
```

Как видим, мы создали директорию `firstprog`, зашли в неё (то есть сделали её текущей) и, дав команду `ls`, убедились, что она пока что пустая, то есть не содержит ни одного файла. **Если что-то здесь оказалось не совсем понятно, обязательно (и немедленно!) перечитайте § 1.4.4**.

Теперь самое время запустить редактор текстов и набрать в нём текст программы. Автор этих строк предпочитает редактор `vim`, но если вы совсем не хотите его изучать (а это, надо признать, всё-таки не

[1] Интерпретаторы Паскаля тоже существуют, но применяются очень редко.

[2] Из этого правила существуют исключения, но они настолько редки, что вы можете совершенно спокойно не обращать на них никакого внимания.

[3] Напомним, что в системах семейства Unix нет привычного для многих пользователей понятия «расширения» имени файла; слово «суффикс» означает примерно то же самое — несколько символов в самом конце имени, которые отделены от основного имени точкой, но, в отличие от других систем, Unix не рассматривает суффикс как что-то иное, нежели просто кусочек имени.

[4] Ещё раз напомним о недопустимости использования термина «папка»!

очень просто), с тем же успехом вы можете воспользоваться другими редакторами, такими как `joe` или `nano`. Во всех случаях принцип запуска редактора текстов одинаков. Сам по себе редактор — это просто программа, имеющая имя для запуска, и этой программе следует сразу при запуске указать (в виде параметра) имя файла, который вы хотите редактировать. Если такого файла ещё нет, редактор создаст его при первом сохранении.

Программа, которую мы напишем, будет очень простой: вся её работа будет заключаться в том, чтобы выдать на экран[5] строку на английском языке, каждый раз одну и ту же. Практической пользы от такой программы, конечно, никакой нет, но нам сейчас важнее заставить работать хоть какую-нибудь программу, просто чтобы убедиться, что мы это можем. В примере программа будет выдавать фразу `Hello, world!`[6], так что мы назовём файл её исходного текста `hello.pas`. Итак, запускаем редактор (если хотите, подставьте вместо `vim` слово `joe` или `nano`):

```
avst@host:~/firstprog$ vim hello.pas
```

Теперь нам нужно набрать текст программы. Выглядеть он будет вот так:

```
program hello;
begin
    writeln('Hello, world')
end.
```

Сейчас самое время дать некоторые пояснения. Первая строчка программы — это так называемый ***заголовок***, показывающий, что весь текст, идущий дальше, представляет собой программу (слово `program`), которую её автор назвал именем `hello` (вообще-то здесь можно написать любое имя, состоящее из латинских букв, цифр и символа подчёркивания, только начинаться это имя должно обязательно с буквы; такие имена мы будем называть ***идентификаторами***). Заголовок заканчивается символом «точка с запятой». Вообще говоря, современные реализации Паскаля позволяют не писать заголовок, но мы этим пользоваться не будем: программа без заголовка выглядит не так наглядно.

Слово `begin`, которое мы написали на второй строке, означает по-английски *начало*; в данном случае оно обозначает начало ***главной***

---

[5] Правильнее говорить, конечно, не об экране, а о *стандартном потоке вывода*, см. § 1.4.9, но мы пока позволим себе вольное обращение с терминами, чтобы раньше времени не усложнять понимание происходящего.

[6] *Здравствуй, мир!* (англ.) Традицию начинать изучение языка программирования с программы, которая печатает именно эту фразу, когда-то давно ввёл Брайан Керниган для языка Си; традиция сама по себе ничем не плоха, так что можно последовать ей и при изучении Паскаля. Впрочем, вы можете воспользоваться любой фразой.

части программы, но наша программа настолько проста, что фактически из одной только этой «главной части» и состоит; позже мы будем писать программы, в которых «главная часть» будет совсем маленькой в сравнении со всем остальным.

Расположенное на следующей строке `writeln('Hello, world!')` как раз и делает то, для чего написана наша программа — выдаёт надпись «`Hello, world!`». Поясним, что слово «`write`» по-английски означает «писáть», а загадочная добавка «`ln`» происходит от слова *line* и означает, что после выполнения печати нужно перевести строку (чуть позже мы этот момент рассмотрим подробнее). Получившееся слово «`writeln`» в Паскале используется для обозначения ***оператора вывода с переводом строки***, а в скобках перечисляется всё, что оператор должен вывести; в данном случае это одна строка.

Читатель, уже знакомый с Паскалем, может возразить, что в большинстве источников `writeln` называют не оператором, а «встроенной процедурой»; но это не совсем корректно, ведь словом «процедура» (без эпитета «встроенная») обозначается подпрограмма, которую пишет сам программист, а встроенными процедурами следует называть такие процедуры, которые программист мог бы написать, но ему не нужно это делать, поскольку компилятор их в себе уже содержит. Ничего похожего на `writeln` с его переменным числом параметров и директивами форматирования вывода мы сами написать не можем; иначе говоря, если бы этой «встроенной процедуры» в языке не было, мы не смогли бы сделать её сами. С `writeln` и другими подобными сущностями связан свой собственный синтаксис (двоеточие после аргумента, за которым идёт целочисленная ширина в знакоместах), то есть это именно часть языка Паскаль, которая обрабатывается компилятором по-своему, а не в соответствии с обобщёнными правилами. В такой ситуации называть `writeln` именно оператором, а не чем-то другим, кажется существенно логичнее. Справедливости ради отметим, что слова `write, writeln, read, readln` и т. п. компилятор рассматривает как обычные идентификаторы, а не как зарезервированные слова, что является весомым аргументом против отнесения этих сущностей к операторам; впрочем, Free Pascal относит эти слова к категории *модификаторов* (modifiers), в которую также входят, например, `break` и `continue`, которые никто не называет иначе как операторами.

Строка, предназначенная к печати, заключена в апострофы, чтобы показать, что этот фрагмент текста обозначает сам себя, а не какие-то конструкции языка. Если бы мы не поставили апострофов, компилятор попытался бы понять, чтó мы имеем в виду под словом «`Hello`», и, не найдя никакого подходящего смысла, выдал бы сообщение об ошибке, а переводить нашу программу в машинный код бы в итоге не стал; но коль скоро слово заключено в апострофы, оно обозначает само себя и ничего больше, так что компилятору тут ни о чём задумываться не нужно. Последовательность символов, заключённая в апострофы и задающая текстовую строку, называется ***строковым литералом*** или ***строковой константой***.

Слово «оператор» представляет собой пример довольно неудачного перевода английского термина; в оригинале эта сущность называется *statement*, что правильнее было бы перевести как «утверждение», «предложение» или как-нибудь ещё.

Дело в том, что в математике традиционно *оператором* называется определённый класс *отображений* (то есть попросту функций), причём в этом значении слово «оператор» представляет собой точный перевод английского *operator*; совершенно естественно, что это английское слово проникло и в программистскую терминологию: при описании многих языков программирования словом *operator* англоязычные специалисты стали обозначать то, что мы по-русски называем *операциями* — сложение, вычитание, умножение, деление, сравнения и прочее в таком духе. Всё это не было проблемой до тех пор, пока не потребовалось перевести с английского текст, содержащий одновременно слова *operator* и *statement*. Даже с этим можно справиться, переводя *operator* как *операцию*, а не как *оператор*; но когда в очередном языке программирования (например, в C++) слово `operator` становится ключевым, проблемы с терминологией оказываются попросту неизбежны: когда человек видит слово «operator», его довольно трудно убедить не произносить русского слова «оператор».

Так или иначе, в русскоязычной программистской литературе оператором называют конструкцию языка программирования, предписывающую выполнение некоторого действия — не вычисления, а именно действия. Полезно помнить, что по-английски это называется вовсе не *operator*, а *statement*.

Последняя строка нашей программы состоит из слова `end` и точки. Слово `end` по-английски означает «конец» (опять же, в данном случае — конец главной части программы). Правила языка Паскаль требуют программу завершить точкой — то ли для верности, то ли просто для красоты.

Итак, текст набран, сохраняем его и выходим из редактора текстов (в vim'е для этого нажимаем Esc-двоеточие-wq-Enter; в nano — Ctrl-O, Enter, Ctrl-X, в joe — Ctrl-K, потом ещё x; в дальнейшем мы не будем рассказывать, как то или иное действие сделать в разных редакторах, ведь в § 1.4.10 всё это уже обсуждалось). Вновь получив приглашение командной строки, убеждаемся, что теперь наша директория уже не пуста — в ней есть файл, который мы только что набрали:

```
avst@host:~/firstprog$ ls
hello.pas
avst@host:~/firstprog$
```

Для наглядности дадим команду, показывающую содержимое файла; с большими текстами так лучше не делать, но наша программа состоит всего из четырёх строк, так что мы можем себе позволить лишний эксперимент. Сама команда называется, как мы уже знаем, `cat`, а параметром ей служит имя файла:

```
avst@host:~/firstprog$ cat hello.pas
program hello;
```

```
begin
    writeln('Hello, world!')
end.
avst@host:~/firstprog$
```

Кстати, пока файл в нашей директории всего один, можно не набирать его имя на клавиатуре, а нажать клавишу Tab, и интерпретатор командной строки сам напишет это имя за нас.

Теперь, убедившись, что файл у нас действительно есть и в нём написано то, что мы ожидали, мы можем запустить компилятор. Напоминаем, он называется `fpc`; как водится, ему нужен параметр, и это, в который уж раз, наше имя файла с исходным текстом:

```
avst@host:~/firstprog$ fpc hello.pas
```

Компилятор напечатает несколько строк, немного различающихся в зависимости от конкретной его версии. Если среди них затесалась строка, похожая на

```
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
```

можете смело не обращать на неё внимания, как и на все остальные строки, если только в них не встречается слово `Error` (ошибка), `Fatal` (фатальная ошибка), `warning` (предупреждение) или `note` (замечание). Сообщение об ошибке (со словом `Error` или `Fatal`) означает, что текст, который вы подсунули компилятору, не соответствует правилам языка Паскаль, так что результатов компиляции вы не получите — компилятор просто не знает, что делать. Предупреждения (со словом `warning`) выдаются, если текст программы формально соответствует требованиям языка, но из каких-то соображений компилятор считает, что результат будет работать не так, как вы ожидали (скорее всего, неправильно); единственным исключением является вышеприведённое предупреждение про `output sections`, его на самом деле выдаёт не компилятор, а вызванная им программа `ld` (компоновщик), и нас это предупреждение не касается. Наконец, замечания (сообщения со словом `note`) компилятор выдаёт, если какая-то часть программы кажется ему странной, хотя и не должна, по его мнению, приводить к неправильной работе.

Например, если бы мы вместо `writeln` написали `writenl`, а потом ещё забыли бы поставить точку в конце программы, то увидели бы, помимо прочего, примерно такие сообщения:

```
hello.pas(3,12) Error: Identifier not found "writenl"
hello.pas(5) Fatal: Syntax error, "." expected but "end of file" found
```

Первым сообщением компилятор ставит нас в известность, что слова `writenl` он не знает, так что из нашей программы ничего хорошего уже не получится; второе сообщение означает, что файл кончился, а компилятор так и не дождался

точки, и это его настолько огорчило, что рассматривать нашу программу дальше он вообще отказывается (этим `Fatal` отличается от просто `Error`).

Обратите внимание на цифры в скобках после имени файла; `hello.pas(3,12)` означает, что ошибка обнаружена в файле `hello.pas`, в строке № 3, в столбце № 12, а `hello.pas(5)` означает ошибку в строке № 5 — такой в нашей программе вообще-то нет, но к тому времени, когда компилятор обнаружил, что файл неожиданно кончился, строка № 4 уже тоже осталась позади, ну а что в пятой строке ничего нет — это уже другой вопрос.

Номера строк, выдаваемые вместе с сообщениями об ошибках и предупреждениями — это очень ценная информация, которая позволяет нам быстро найти то место в нашей программе, где компилятору что-то не понравилось. Вне зависимости от того, каким редактором текстов вы пользуетесь, желательно сразу же разобраться, как в нём найти строку по номеру, иначе программировать будет очень сложно.

Убедиться, что компиляция прошла успешно, можно, в очередной раз посмотрев содержимое текущей директории с помощью команды `ls`:

```
avst@host:~/firstprog$ ls
hello  hello.o  hello.pas
avst@host:~/firstprog$
```

Как видим, файлов стало больше. Файл `hello.o` нас не очень интересует, это так называемый ***объектный модуль***, который компилятор подаёт на вход компоновщику, а потом почему-то забывает удалить; а вот файл, который называется просто `hello` без суффикса — это и есть то, ради чего мы запускали компилятор: ***исполняемый файл***, то есть попросту файл, содержащий машинный код, соответствующий нашей программе. Для верности попробуем получить об этих файлах больше информации:

```
avst@host:~/firstprog$ ls -l
total 136
-rwxr-xr-x 1 avst avst 123324 2015-06-04 19:57 hello
-rw-r--r-- 1 avst avst   1892 2015-06-04 19:57 hello.o
-rw-r--r-- 1 avst avst     47 2015-06-04 19:57 hello.pas
avst@host:~/firstprog$
```

В первой колонке видим у файла `hello` установленные права на исполнение (буква «`x`»). Заодно замечаем, насколько этот файл больше, чем исходный текст: файл `hello.pas` занимает всего 47 байт (по количеству символов в нём), тогда как получившийся исполняемый файл «весит» более 120 **кило**байт. На самом деле всё не так уж плохо: как мы вскоре сможем убедиться, с ростом исходной программы получающийся исполняемый файл почти не будет увеличиваться. Просто компилятор вынужден загнать в исполняемый файл сразу всё, что нужно для выполнения практически любых операций ввода-вывода, а мы эти возможности пока почти не используем.

Осталось лишь запустить полученный файл. Делается это так:

```
avst@host:~/firstprog$ ./hello
Hello, world!
avst@host:~/firstprog$
```

Поясним, почему обязательно надо написать «`./hello`», а не просто «`hello`». Если в имени команды нет ни одного символа «`/`», командный интерпретатор пытается найти встроенную команду с таким именем, а если такой нет — то выполняет поиск исполняемого файла, причём ищет он этот файл в системных директориях, таких как `/bin`, `/usr/bin`, `/usr/local/bin` и т. п.; точнее говоря, поиск выполняется в директориях, которые перечислены в переменной окружения `PATH`; вы можете увидеть список этих директорий примерно следующим образом:

```
avst@host:~/firstprog$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
avst@host:~/firstprog$
```

Ясно, что вашей текущей (рабочей) директории в этом списке нет; если написать просто «`hello`», команды с таким именем командный интерпретатор не найдёт и выдаст сообщение об ошибке. Если же в имени команды присутствует символ «`/`», который, как мы знаем, используется для отделения имени файла от имени директории, а также имён директорий друг от друга, то командный интерпретатор считает, что ему уже задано, какой именно файл запускать, и искать ничего не надо. Осталось сказать, что точкой обозначается текущая директория, какова бы она ни была; следовательно, команда «`./hello`» означает дословно «запустить файл `hello` из текущей директории», что нам и требуется.

Естественно, в программе может быть больше одного оператора; в простейшем случае они будут исполняться один за другим. Рассмотрим для примера такую программу:

```
program hello2;
begin
    writeln('Hello, world!');
    writeln('Good bye, world.')
end.
```

Теперь у нас в программе не один оператор, как раньше, а два; можно заметить, что **между операторами ставится символ «точка с запятой»**. Обычно точку с запятой ставят в конце очередного оператора, чтобы отделить его от следующего, если, конечно, этот «следующий» есть; если его нет, точка с запятой не нужна. Отметим, что слово `end` оператором не является, так что точку с запятой перед ним обычно не ставят.

Если эту программу откомпилировать и запустить, она напечатает сначала (в результате выполнения первого оператора) «`Hello, world!`», а затем (в результате выполнения второго оператора) «`Good bye, world.`»:

```
avst@host:~/firstprog$ ./hello2
Hello, world!
Good bye, world.
avst@host:~/firstprog$
```

Вернёмся теперь немного назад и поясним чуть более подробно буквы «`ln`» в названии оператора `writeln`, которые, как мы уже сказали, означают перевод строки. В языке Паскаль присутствует также оператор `write`, работающий абсолютно так же, как и `writeln`, но не делающий перевода строки по окончании операции вывода. Попробуем отредактировать нашу первую программу, убрав буквы «`ln`»:

```
program hello;
begin
    write('Hello, world!')
end.
```

После этого снова запустим компилятор `fpc`, чтобы обновить наш исполняемый файл, и посмотрим, как наша программа будет работать теперь. Картина на экране получится примерно такая:

```
avst@host:~/firstprog$ ./hello
Hello, world!avst@host:~/firstprog$
```

В этот раз, как и раньше, наша программа выдала надпись «`Hello, world!`», но строку не перевела; когда после её завершения интерпретатор командной строки напечатал очередное приглашение, оно появилось в той же строке, где и выдача нашей программы.

Приведём ещё один пример. Напишем такую программу:

```
program NewlineDemo;
begin
    write('First');
    writeln('Second');
    write('Third');
    writeln('Fourth')
end.
```

Сохраним эту программу в файл, например, `nldemo.pas`, откомпилируем и запустим:

```
avst@host:~/firstprog$ fpc nldemo.pas
.........................
avst@host:~/firstprog$ ./nldemo
FirstSecond
ThirdFourth
avst@host:~/firstprog$
```

В программе мы вывели первое и третье слово с помощью `write`, тогда как второе и четвёртое — с помощью `writeln`, то есть с переводом строки. Эффект от этого наблюдается вполне наглядный: слово «`Second`» выведено в той же строке, что и «`First`» (после которого программа не сделала перевода строки), так что они слились вместе; то же самое произошло со словами «`Third`» и «`Fourth`».

Прежде чем завершить разговор о простейших программах, отметим ещё один очень важный момент — то, как мы расположили друг относительно друга разные части наших исходных текстов. Заголовок программы, а также слова `begin` и `end`, обозначающие начало и конец её главной части, мы писали в начале строки, тогда как операторы, составляющие главную часть, мы сдвинули вправо, поставив перед ними четыре пробела; кроме того, все эти элементы мы расположили каждый на отдельной строке.

Интересно, что компилятору всё это совершенно не нужно. Мы могли бы написать, например, и так:

```
program hello; begin writeln('Hello, world!') end.
```

или вот так:

```
          program hello; begin
writeln('Hello, world!') end.
```

Что касается нашей второй программы, в которой больше операторов, то там гораздо больше и возможностей для бардака — например, можно было бы написать что-то вроде

```
      program NewlineDemo; begin write('First');
  writeln('Second');  write(
  'Third');  writeln('Fourth')  end.
```

С точки зрения компилятора при этом ничего бы не изменилось; более того, пока речь идёт о совсем примитивных программах, вряд ли что-то изменится и с нашей собственной точки зрения, разве что взбунтуется наше эстетическое чувство. Однако для мало-мальски сложных программ ситуация резко меняется. Читатель вскоре сам сможет увидеть, что в текстах программ достаточно тяжело разбираться; иначе говоря, чтобы по имеющемуся тексту программы понять, что́ она делает и как она это делает, нужно прилагать интеллектуальные усилия, которые во многих случаях превышают усилия, требующиеся для написания текста программы с нуля. Если при этом текст ещё и написан как попало, разобраться в нём оказывается, как говорил известный герой Булгакова, решительно невозможно. Самое интересное, что безнадёжно заблудиться можно не только в чужой программе, но и в своей собственной,

причём иногда это происходит раньше, чем программа будет дописана. Нет ничего проще (и обиднее), чем запутаться в собственном коде, не успев толком ничего написать.

На практике программистам приходится читать программы — как свои, так и чужие — едва ли не больше времени, чем писать их, так что, вполне естественно, на *читаемость программного кода* приходится обращать самое пристальное внимание. Опытные программисты говорят, что **текст программы предназначен прежде всего для прочтения человеком, и лишь во вторую очередь — для обработки компьютером**. Практически все существующие языки программирования предоставляют программисту определённую свободу по части оформления кода, что позволяет самую сложную программу сделать понятной с первого взгляда любому её читателю, знакомому с используемым языком; но с тем же успехом можно сделать очень простую программу непонятной её собственному автору.

Для улучшения читаемости программ применяется целый ряд приёмов, которые вместе составляют *грамотный стиль оформления программного кода*, и одним из самых важных моментов здесь является применение ***структурных отступов***. Почти все современные языки программирования допускают в начале любой строки текста *произвольное количество*[7] *пробельных символов* — пробелов или табуляций, что позволяет оформить программу так, чтобы её структуру можно было «схватить» расфокусированным взглядом, не вчитываясь. Как мы вскоре увидим, структура программы формируется по принципу вложения одного в другое; техника структурных отступов позволяет подчеркнуть структуру программы, попросту *сдвигая вправо* любые вложенные конструкции относительно того, *во что они вложены*.

В наших простейших программах уровень вложения оказался только один: операторы `writeln` и `write` оказались вложены в главную часть программы. При этом ни заголовок, ни сама главная часть ни во что не вложены, так что их мы написали, начиная с крайней левой колонки символов в тексте программы, а операторы `writeln` и `write` сдвинули вправо, чтобы показать, что они вложены в главную часть программы. Единственное, что мы здесь выбрали достаточно произвольно — это *размер* структурного отступа, который в наших примерах на протяжении всей книги составит четыре пробела. В действительности во многих проектах (в частности, в ядре ОС Linux) для структурного отступа используют символ табуляции, причём ровно один. Можно встретить размер отступа в два пробела — именно такие отступы приняты в коде

[7] Любопытным исключением из этого утверждения оказывается язык Python, который как раз не допускает *произвольного* количества пробелов в начале строки — напротив, в нём на уровне синтаксиса имеется жесткое *требование* к количеству таких пробелов, соответствующее принципам оформления структурных отступов. Иначе говоря, большинство языков программирования *допускают* соблюдение структурных отступов, тогда как Python такого соблюдения *требует*.

программ, выпускаемых Фондом свободного программного обеспечения (FSF). Совсем редко используется три пробела; такой размер отступа иногда встречается в программах, написанных для Windows. Другие размеры отступа использовать не следует, и этому есть ряд причин. Одного пробела слишком мало для визуального выделения блоков, левый край текста при этом воспринимается как нечто плавное и не служит своей цели. Количество пробелов, превышающее четыре, трудно вводить: если их больше пяти, их приходится считать при вводе, что сильно замедляет работу, но и пять пробелов оказывается вводить очень неудобно. Если же использовать больше одной табуляции, то на экран по горизонтали почти ничего не поместится.

Выбрав какой-то один размер отступа, его следует придерживаться во всём тексте программы. Это касается также и других решений, принятых по поводу стиля оформления: исходный текст программы должен быть стилистически однородным, иначе его читателям придётся постоянно перестраивать своё восприятие, что довольно утомительно.

2.2. Выражения и переменные

2.2.1. Арифметические операции и понятие типа

Операторы `write` и `writeln` могут печатать не только строки. Например, если нам срочно потребуется перемножить 175 и 113, мы можем написать для этого программу[8]:

```
program mult175_113;
begin
    writeln(175*113)
end.
```

Откомпилировав и запустив эту программу, мы увидим на экране ответ `19775`. Символом «`*`» в Паскале (и в большинстве других языков программирования) обозначается операция умножения, а конструкция «`175*113`» представляет собой пример ***арифметического выражения***. Мы могли бы сделать эту программу чуть более «дружественной пользователю», показав в её выводе, к чему, собственно говоря, относится выдаваемый ответ:

```
program mult175_113;
begin
    writeln('175*113 = ', 175*113)
end.
```

[8]Конечно, гораздо проще воспользоваться калькулятором или арифметическими возможностями командного интерпретатора (см. § 1.4.13), но сейчас это неважно.

Здесь мы предложили оператору `writeln` два аргумента, предназначенных к печати: строку и арифметическое выражение. Таких аргументов мы можем задать сколько угодно, перечислив их, как и в этом примере, через запятую. Если теперь откомпилировать и запустить новый вариант программы, выглядеть это будет примерно так:

```
avst@host:~/firstprog$ ./mult
175*113 = 19775
avst@host:~/firstprog$
```

Обратите внимание на фундаментальную (с точки зрения компилятора) разницу между символами, входящими в строковый литерал, и символами вне его: если выражение `175*113`, находящееся вне строкового литерала, было вычислено, то совершенно те же символы, но находящиеся между апострофов и входящие, таким образом, в строковый литерал, компилятор вовсе не пытался рассматривать в качестве выражения и вообще в любом ином качестве, нежели просто как символы. Поэтому в соответствии с нашими указаниями программа сначала напечатала один за другим все символы из заданного строкового литерала (в том числе, как легко видеть, пробелы), а затем следующий аргумент оператора `writeln`, в роли которого выступает выражение со значением `19775`.

Разумеется, умножение — далеко не единственная арифметическая операция, которую поддерживает язык Паскаль. Сложение и вычитание в Паскале обозначаются естественными для этого знаками «`+`» и «`-`», так что, например, выражение `12 + 105` даст в результате `117`, а выражение `10 - 25` даст `-15`. Так же, как и при записи математических формул, в выражениях Паскаля операции имеют разные ***приоритеты***; например, приоритет умножения в Паскале, как и в математике, выше, чем приоритет сложения и вычитания, поэтому значением выражения `10 + 5 * 7` будет `45`, а не `105`: при вычислении этого выражения сначала производится умножение, то есть `5` умножают на `7`, получается `35`, и только после этого полученное число прибавляют к десяти. Точно так же, как и в математических формулах, в выражениях Паскаля мы можем использовать круглые скобки для изменения порядка выполнения операций: `(10 + 5) * 7` даст в результате `105`.

Отметим, что в Паскале предусмотрены также *унарные* (то есть имеющие один аргумент) операции «`+`» и «`-`», то есть можно, например, написать `-(5 * 7)`, получится `-35`: унарная операция `-`, как и следовало ожидать, меняет знак числа на противоположный. Унарная операция `+` тоже предусмотрена, но большого смысла в ней нет: её результат всегда равен аргументу.

Несколько сложнее обстоят дела с операцией деления. Обычное математическое деление обозначается косой чертой «`/`»; нужно только не забывать, что операция деления отличается от умножения, сложения

и вычитания: даже если её аргументы целые, результат в общем случае не может быть выражен целым числом. Это приводит к несколько неожиданному для начинающих эффекту. Например, если мы напишем в программе оператор `writeln(14/7)`, в его выдаче мы можем даже не сразу опознать число 2:

```
2.0000000000000000E+0000
```

Чтобы понять, в чём тут загвоздка и почему `writeln` не напечатал просто «2», потребуется довольно пространное объяснение, вводящее понятие ***типа выражения***[9]. Поскольку это одно из основополагающих понятий в программировании и в дальнейшем мы в любом случае без него не обойдёмся, попытаемся разобраться с ним прямо сейчас.

Отметим для начала, что все числа, которые мы записывали в приведённых выше примерах, целые; если бы мы написали что-нибудь вроде `2.75`, то речь бы пошла о *числе другого типа* — так называемом числе с плавающей точкой. Мы с вами подробно рассматривали представление чисел обоих видов (см. § 1.6.2 и 1.6.3) и видели, что они хранятся и обрабатываются совершенно по-разному. Заметим, что «`2`» и «`2.0`» — с математической точки зрения одно и то же число, но в программировании это совершенно разные вещи, поскольку у них разное представление, и для выполнения операций над ними требуются совершенно разные последовательности машинных команд. Более того, целое число 2 может быть представлено как двухбайтное, однобайтное, четырёхбайтное или даже восьмибайтное целое, знаковое или беззнаковое[10]. Всё это тоже примеры ситуаций, когда речь идёт о разных типах. В Паскале каждый тип имеет своё имя; например, знаковые целые числа могут быть типа `shortint`, `integer`, `longint` и `int64` (соответственно однобайтное, двухбайтное, четырёхбайтное и восьмибайтное знаковое целое), соответствующие беззнаковые типы называются `byte`, `word`, `longword` и `qword`, а числа с плавающей точкой обычно относят к типу `real` (хотя тот же Free Pascal поддерживает и другие типы чисел с плавающей точкой).

Как это часто бывает в инженерно-технических дисциплинах, понятие типа плохо поддаётся определению: если даже попытаться такое определение дать, скорее всего, оно окажется в чём-то не соответствующим реальности. Чаще всего в литературе можно встретить утверждение, что тип есть множество возможных значений, но, приняв такое определение, мы не сможем объяснить, чем различаются всё те же 2 и `2.0`; следовательно, в понятие типа должно входить не только множество значений, но и принятое для данного типа машинное представление этих значений. Кроме того, многие авторы подчёркивают

[9] Разумеется, если мы просто напишем в программе число, такая запись (так называемая числовая константа, или числовой литерал) тоже будет частным случаем выражения.

[10] Если здесь что-то непонятно, обязательно перечитайте § 1.6.2.

важную роль *набора операций*, определённых для данного типа, и это, в принципе, разумно. Заявив, что **тип выражения фиксирует множество значений, их машинное представление и набор операций, определённых над этими значениями**, мы, по-видимому, окажемся близки к истине.

Типы (и выражения) бывают не только числовыми. Например, тип `boolean`, предназначенный для работы с логическими выражениями, предусматривает всего два значения: `true` (истина) и `false` (ложь); набор операций над значениями этого типа включает хорошо знакомые нам конъюнкцию, дизъюнкцию, отрицание и «исключающее или». Использовавшееся в нашей самой первой программе `'Hello, world!'` есть ничто иное, как константа (а значит, выражение) типа `string` (строка), и над строками даже есть операция, правда, всего одна — «сложение», обозначаемое символом «+», которое на самом деле означает конкатенацию (то есть, попросту говоря, соединение) двух строк в одну. Строки также можно сравнивать, но результатом сравнения, конечно, будет уже не строка, а логическое значение, то есть значение типа `boolean`. Для работы с одиночными символами используется тип `char`, и так далее.

Вернёмся к операции деления, с которой весь этот разговор, собственно говоря, и начался. Теперь уже нетрудно догадаться, почему `writeln(14/7)` повёл себя таким неожиданным образом. Результат операций умножения, сложения и вычитания обычно имеет тот же тип, что и аргументы операции, то есть при сложении или умножении двух целых мы получаем целое; если же мы попытаемся сложить два числа с плавающей точкой, то и результат будет числом с плавающей точкой. С делением всё иначе: его результат *всегда* имеет тип числа с плавающей точкой, чем и обусловлен полученный эффект.

Говоря подробнее, оператор `writeln`, если не принять специальных мер, выводит числа с плавающей точкой в так называемой ***научной нотации*** — в виде мантиссы и порядка, причём мантисса представляет собой десятичную дробь, удовлетворяющую условию $1 \leqslant m < 10$, и выдаётся на печать с 16 знаками после запятой; после мантиссы в научной нотации следует буква `E` (от слова *exponent*), отделяющая от мантиссы запись *порядка* — положительного или отрицательного целого числа p, представляющего собой *степень десяти*; всё число равно $m \cdot 10^p$. Это поведение оператора `writeln` можно изменить, если задать в явном виде, сколько знакомест мы хотим выделить на печать числа и сколько из них — на знаки после запятой. Для этого в операторах `write` и `writeln` после числового выражения добавляют двоеточие, целое число (сколько знакомест всего), ещё одно двоеточие и ещё одно число (сколько знаков после запятой). Например, если написать `writeln(14/7:7:3)`, то напечатано будет `2.000`, причём, поскольку здесь всего пять знаков, *перед этим* будет напечатано ещё два пробела.

Кроме обычного деления, Паскаль предусматривает ***целочисленное деление***, известное из школьной математики как ***деление с***

остатком. Для этого вводятся ещё две операции, обозначаемые словами `div` и `mod`, которые означают деление (с отбрасыванием остатка) и остаток от такого деления. Например, если написать

```
writeln(27 div 4, ' ', 19 mod 4);
```

то напечатано будет два числа через пробел: «`6 3`».

2.2.2. Переменные, инициализация и присваивание

Все примеры, приведённые в предыдущем параграфе, обладают одним фундаментальным недостатком — они печатают каждый раз одно и то же, невзирая ни на какие обстоятельства, потому что решают не просто одну и ту же задачу (так поступает большинство программ), а *одну и ту же задачу для одного и того же частного случая*. Любители рассуждать о свойствах алгоритмов, возможно, заявили бы, что алгоритмы, реализованные в наших примерах, не обладают свойством массовости.

Было бы гораздо лучше, если бы программа, умеющая решать некую задачу, пусть даже всего одну и очень простую, всё же решала бы её *в общем виде*, то есть запрашивала бы у пользователя (или брала бы откуда-то ещё) значения величин и работала бы с ними, а не с теми величинами, которые жестко заданы прямо при написании программы в её исходном тексте. Чтобы этого достичь, нам потребуется одна очень важная возможность: мы должны уметь *хранить в памяти*[11] некую информацию и работать с ней. В Паскале[12] для этого используются так называемые ***переменные***.

Переменная в простейшем случае обозначается ***идентификатором*** — словом, которое может состоять из латинских букв, цифр и знака подчёркивания, но начинаться должно с буквы[13]; такой идентификатор называется ***именем переменной***. Позже мы столкнёмся с переменными, которые не имеют имён, но до этого нам пока далеко. Например, мы можем назвать переменную «`x`», «`counter`», «`p12`»,

[11] Имеется в виду оперативная память компьютера, точнее, та её часть, которая отведена нашей программе для работы.

[12] Это верно не только для Паскаля, но и, пожалуй, для большинства существующих языков программирования — но, тем не менее, не для всех. Существуют экзотические языки программирования, вообще не предусматривающие переменных. Кроме того, во многих языках программирования переменные присутствуют, но используются совершенно иначе и устроены совсем не так, как в Паскале; примерами таких языков могут служить Пролог, Рефал, Хаскелл и другие. Впрочем, если рассматривать только так называемые *императивные* языки программирования, для которых стиль мышления программиста схож с паскалевским, то во всех таких языках понятие переменной присутствует и означает примерно одно и то же.

[13] Со знака подчёркивания идентификатор тоже можно начать, но в программах на Паскале так обычно не делают; а вот с цифры идентификатор начинаться не может.

«`LineNumber`» или «`grand_total`». Отметим, что Паскаль не различает заглавные буквы и строчные, то есть по правилам Паскаля слова «`LineNumber`», «`LINENUMBER`», «`linenumber`» и «`LiNeNuMBeR`» обозначают одну и ту же переменную; иной вопрос, что использование для одного и того же идентификатора различных вариантов написания считается у программистов крайне дурным тоном.

С переменной в каждый момент времени связано некое ***значение***; говорят, что *переменная хранит значение* или что *значение содержится в переменной*. Если имя переменной встречается в арифметическом выражении, производится так называемое ***обращение к переменной***, при котором в выражение вместо имени переменной подставляется её значение.

Паскаль относят к категории ***строго типизированных языков программирования***; это, в частности, означает, что каждая переменная в программе на Паскале имеет строго определённый тип. В предыдущем параграфе мы рассматривали понятие ***типа выражения***; тип переменной можно понимать, с одной стороны, как тип выражения, состоящего из одного только обращения к этой переменной, а с другой стороны — как тип выражения, значение которого может в такой переменной храниться. Например, самый популярный в программах на Паскале тип, который называется `integer`, подразумевает, что переменные этого типа используются для хранения целочисленных значений, могут содержать числа от `-32768` до `32767` (числа типа `integer`, двубайтные знаковые целые), и обращение к такой переменной тоже будет, естественно, выражением типа `integer`.

Прежде чем использовать в программе переменную, её необходимо описать, для чего в программу между заголовком и главной частью вставляют ***секцию описания переменных***; эта секция начинается словом «`var`» (от слова *variables* — переменные), за которым следуют одно или несколько ***описаний переменных*** с указанием их типов. Например, секция описаний переменных может выглядеть так:

```
var
    x: integer;
    y: integer;
    flag: boolean;
```

Здесь переменные `x` и `y` имеют тип `integer`, а переменная `flag` — тип `boolean` (напомним, что этот тип, иногда называемый логическим, подразумевает только два возможных значения — `true` и `false`, то есть «истина» и «ложь»). Переменные одного типа можно сгруппировать в одно описание, перечислив их через запятую:

```
var
    x, y, z: integer;
```

В Паскале предусмотрено несколько различных способов для занесения значения в переменную. Например, можно задать ***начальное значение переменной*** прямо в секции описаний; это называется ***инициализацией***:

```
var
    x: integer = 500;
```

Если этого не сделать, в переменной всё равно будет содержаться какое-то значение, но какое — предсказать невозможно, это может быть произвольный мусор. Такая переменная называется ***неинициализированной***. Использование «мусорного» значения ошибочно по своей сути, так что компилятор, видя такое использование, выдаёт предупреждение; к сожалению, далеко не всегда он может с этим справиться, в некоторых случаях программа может быть написана так «хитро», что в ней будет производиться обращение к неинициализированной переменной, но компилятор этого просто не заметит. Иногда бывает и наоборот.

Значение переменной можно в любой момент изменить, выполнив так называемый ***оператор присваивания***. В Паскале присваивание обозначается знаком «`:=`», слева от которого записывается переменная, которой нужно присвоить новое значение, а справа — выражение, значение которого будет занесено в переменную. Например, оператор

```
x := 79
```

(читается «`x` положить равным `79`») занесёт в переменную `x` значение `79`, и с того момента, когда этот оператор сработал, в выражениях, содержащих переменную `x`, будет использоваться именно это значение. Старое значение переменной, каково бы оно ни было, при выполнении присваивания теряется. Работу присваивания можно проиллюстрировать на следующем примере:

```
program assignments;
var
    x: integer = 25;
begin
    writeln(x);
    x := 36;
    writeln(x);
    x := 49;
    writeln(x)
end.
```

В момент выполнения первого из операторов `writeln` в переменной `x` содержится начальное значение, заданное при описании, то есть число `25`; именно оно и будет напечатано. Затем оператор присваивания, расположенный в следующей строке, изменит значение `x`; старое значение `25`

будет потеряно, и переменная будет теперь содержать значение 36, которое и напечатает второй оператор writeln; после этого в переменную будет занесено значение 49, и последний оператор writeln напечатает как раз его. В целом выполнение этой программы будет выглядеть так:

```
avst@host:~/firstprog$ ./assignments
25
36
49
avst@host:~/firstprog$
```

Несколько более сложен для понимания другой пример присваивания:

```
x := x + 5
```

Здесь сначала вычисляется значение выражения справа от знака присваивания; поскольку само присваивание пока не произошло, при вычислении используется *старое* значение x, то, которое было в этой переменной непосредственно перед началом выполнения оператора. Затем вычисленное значение, которое для нашего примера будет на 5 больше, чем значение x, заносится обратно в переменную x, то есть, грубо говоря, в результате выполнения этого оператора значение, содержавшееся в переменной x, становится на пять больше: если там было 17, то станет 22, если было 100, станет 105 и так далее.

2.2.3. Идентификаторы и зарезервированные слова

В предыдущем параграфе нам потребовалось описывать переменные, давая им *имена*, такие как «x», «flag», «LineNumber» и т. п., мы даже ввели понятие *идентификатора*: слово, которое может состоять из латинских букв, арабских цифр и знака подчёркивания, причём начинаться должно обязательно с буквы. Если вспомнить самое начало разговора о Паскале, то аналогичное имя-идентификатор нам потребовалось в заголовке программы в качестве имени для всей программы целиком. Кроме программы и переменных, в Паскале существует целый ряд сущностей, описываемых пользователем (то есть программистом), которым тоже нужны имена; постепенно мы обсудим большую часть этих сущностей и научимся их использовать. Общее правило тут одно: в качестве имени для чего бы то ни было в Паскале может выступать идентификатор, и правила для создания идентификаторов используются одни и те же, независимо от того, для чего очередной идентификатор понадобился.

При этом нам встречались слова, которые предоставляет нам сам Паскаль, то есть такие, которые нам не нужно было описывать: это, с

одной стороны, слова `program`, `var`, `begin`, `end`, `div`, `mod`, а с другой — слова `write` и `writeln`, `integer`, `boolean`.

Несмотря на то, что все эти слова в той или иной степени являются частью языка Паскаль, они относятся к разным категориям. Слова `program`, `var`, `begin`, `end`, `div` и `mod` (а также многие другие, часть из которых мы изучим в будущем) по правилам Паскаля считаются ***зарезервированными словами*** (иногда их называют также ***ключевыми словами***); это значит, что мы не можем использовать их в качестве имён для переменных или чего-то другого; формально они вообще не считаются идентификаторами, отсюда название «зарезервированные»: свойство зарезервированности как раз и проявляется в том, что эти слова не могут служить идентификаторами.

С другой стороны, слова `write`, `writeln` и даже слово `integer`, хотя и введены Паскалем, тем не менее представляют собой обыкновенные идентификаторы (на самом деле `write` и `writeln` — не совсем обыкновенные, а вот `integer` и `boolean` — действительно обыкновенные), то есть мы можем описать переменную с именем `integer` (какого-нибудь другого типа), но при этом потеряем возможность использовать тип `integer`, ведь его имя будет занято под переменную. Конечно же, так делать не следует; но стоит иметь в виду, что такая возможность существует, хотя бы из соображений общей эрудиции.

Стоит отметить, что в делении слов, предоставляемых языком Паскаль, на зарезервированные слова и «встроенные идентификаторы» наблюдается определённая степень произвола. Например, слова `true` и `false`, используемые для обозначения логической истины и логической лжи, в классических вариантах Паскаля, а также и в знаменитом Turbo Pascal считались простыми идентификаторами (о чём большинство программистов даже не подозревало, поскольку никогда и никому не приходило в голову использовать эти слова для чего-то другого). Создатели Free Pascal сочли это неудобным, так что компилятор Free Pascal рассматривает эти два слова (а также `new`, `dispose` и `exit`) как зарезервированные.

2.2.4. Ввод информации для её последующей обработки

Кроме присваивания, существуют и другие ситуации, в которых переменная изменяет своё текущее значение; один из самых типичных примеров — выполнение так называемых ***операций ввода***. В ходе такой операции программа получает информацию из внешнего источника — с клавиатуры, из файла на диске, из канала связи по компьютерной сети и т. п.; естественно, полученную информацию нужно где-то сохранить, и для этого используются переменные.

В языке Паскаль наиболее популярным средством для операций ввода служит ***оператор ввода***, обозначаемый словом `read`, и его вариация `readln`. Рассмотрим для начала простенький пример — программу,

которая вводит с клавиатуры целое число, возводит его в квадрат и печатает полученный результат:

```
program square;
var
    x: integer;
begin
    read(x);
    x := x*x;
    writeln(x)
end.
```

Как видим, в программе используется одна переменная типа `integer`, которая называется `x`. Главная часть программы при этом включает в себя три оператора. Первый из них, `read`, предписывает выполнить чтение целого числа с клавиатуры, после чего занести прочитанное число в переменную `x`. Выполнение программы при этом остановится до тех пор, пока пользователь не введёт число, причём в связи с определёнными особенностями режима работы терминала (точнее, в нашем случае — его программного эмулятора, который повторяет особенности работы настоящих терминалов) программа «увидит» введённое число не раньше, чем пользователь нажмёт клавишу Enter.

Второй оператор в нашем примере — это оператор присваивания; выражение, стоящее в его правой части, берёт текущее значение переменной `x` (то есть значение, которое только что было прочитано с клавиатуры оператором `read`), умножает его само на себя, т. е. возводит в квадрат, а результат заносит обратно в переменную `x`. Третий оператор — уже хорошо знакомый нам `writeln` — печатает получившийся результат. Выполнение этой программы может выглядеть так:

```
avst@host:~/firstprog$ ./square
25
625
avst@host:~/firstprog$
```

Сразу после запуска программа «замирает», так что неопытный пользователь может подумать, что она вообще зависла, но на самом деле программа просто ждёт, когда пользователь введёт требуемое число. В примере выше число `25` ввёл пользователь, а число `625` выдала программа.

Кстати, сейчас самое время показать, почему выражение «ввод с клавиатуры» не вполне соответствует действительности и правильнее будет говорить о «вводе из стандартного потока ввода». Для начала создадим текстовый файл, содержащий одну строку, а в этой строке — число, пусть это для разнообразия будет `37`. Файл мы назовём `num.txt`. Для его создания можно воспользоваться тем же редактором

текстов, который вы применяете для ввода текстов программ, но можно поступить и проще — например, так:

```
avst@host:~/firstprog$ echo 37 > num.txt
avst@host:~/firstprog$ cat num.txt
37
avst@host:~/firstprog$
```

Теперь запустим нашу программу `square`, перенаправив ей ввод из файла `num.txt`:

```
avst@host:~/firstprog$ ./square < num.txt
1369
avst@host:~/firstprog$
```

Число `1369` — это квадрат числа `37`; его напечатала наша программа. При этом исходное число она, как мы видим, с клавиатуры не вводила — она в соответствии с нашими указаниями прочитала его из файла `num.txt`. В этом несложно убедиться: отредактируйте файл `num.txt`, заменив число `37` на какое-нибудь другое, и снова запустите программу `quad` с перенаправлением из файла, как показано в примере; в этот раз программа напечатает квадрат того числа, которое вы занесли в файл.

В нашем примере мы сначала использовали системную команду `echo` с перенаправлением вывода, чтобы сформировать файл, содержащий число, а затем запускали нашу программу с перенаправлением ввода из этого файла. Примерно тех же результатов можно добиться без всяких промежуточных файлов, запустив команду `echo` одновременно с нашей программой так называемым *конвейером* (см. § 1.4.9); при этом вывод команды `echo` пойдёт непосредственно на ввод нашей программе. Делается это так:

```
avst@host:~/firstprog$ echo 37 | ./square
1369
avst@host:~/firstprog$ echo 49 | ./square
2401
avst@host:~/firstprog$
```

Перенаправим вывод в файл `result.txt`:

```
avst@host:~/firstprog$ echo 37 | ./square > result.txt
avst@host:~/firstprog$
```

В этот раз на экран вообще ничего не вывелось, зато результат оказался записан в файл, в чём легко убедиться:

```
avst@host:~/firstprog$ cat result.txt
1369
avst@host:~/firstprog$
```

Теперь мы знаем на собственном опыте: **вывод «на экран» далеко не всегда идёт на экран, а ввод «с клавиатуры» отнюдь не всегда осуществляется с клавиатуры**. Именно поэтому правильнее говорить о выводе в стандартный поток вывода и о вводе из стандартного потока ввода. Подчеркнём, что **сама программа вообще не знает, откуда идёт её ввод и куда направлен её вывод**, все перенаправления осуществляет интерпретатор командной строки непосредственно перед тем, как запустить нашу программу.

2.2.5. Берегись нехватки разрядности!

> Приехал как-то Илья Муромец рубить голову Змею Горынычу. Приехал — и отрубил Змею голову. А у Змея две головы выросли. Отрубил Илья Муромец Змею две головы — а у того четыре выросли. Отрубил он ему четыре — а у того восемь голов выросло. Рубил Муромец головы, рубил, рубил-рубил, в конце концов нарубил в общей сложности 65535 голов — вот тут-то и помер Змей Горыныч. Потому что был он шестнадцатиразрядный.

Продолжим эксперименты с программой `square`, начатые в предыдущем параграфе, но на этот раз возьмём числа побольше.

```
avst@host:~/firstprog$ echo 100 | ./square
10000
avst@host:~/firstprog$ echo 150 | ./square
22500
avst@host:~/firstprog$ echo 200 | ./square
-25536
avst@host:~/firstprog$
```

Если с первыми двумя запусками всё было вроде бы в порядке, то на третьем что-то явно пошло не так. Чтобы понять, что происходит, вспомним, что переменные типа `integer` в Паскале могут принимать значения от -32768 до 32767; но ведь квадрат числа 200 равен 40000, то есть в переменную типа `integer` он попросту не помещается! Отсюда нелепый результат, ко всему ещё и отрицательный.

Результат, несмотря на его нелепость, очень просто объяснить. Мы уже знаем, что имеем дело со *знаковым целым числом* и у нас произошло *переполнение* (см. § 1.6.2, стр. 171). Разрядность наших чисел составляет 16 бит, так что при переполнении итоговое число получается на $2^{16} = 65536$ меньше, чем должно быть. Правильный результат должен был составлять $200^2 = 40000$, но

в результате переполнения из него вычлось 65536, так что результат составил $40000 - 65536 = -25536$; ровно это мы и наблюдаем в примере.

Наибольшее число, которое наша программа обрабатывает корректно — 181, его квадрат составляет 32761; квадрат числа 182, составляющий 33124, в разрядность числа типа `integer` уже «не лезет». Но всё не так страшно, просто нужно применить другой тип переменной. Наиболее очевидным кандидатом на роль такого типа оказывается `longint`, имеющий разрядность 32 бита; переменные этого типа могут принимать значения от -2147483648 до 2147483647 (т. е. от -2^{31} до $2^{31} - 1$). В программе достаточно изменить одно слово — просто заменить `integer` на `longint`:

```
program square;
var
    x: longint;
begin
    read(x);
    x := x*x;
    writeln(x)
end.
```

и возможности нашей программы (после её перекомпиляции) резко возрастут:

```
avst@host:~/firstprog$ echo 182 | ./square
33124
avst@host:~/firstprog$ echo 200 | ./square
40000
avst@host:~/firstprog$ echo 20000 | ./square
400000000
avst@host:~/firstprog$
```

Конечно, радоваться рано, свой предел есть и здесь:

```
avst@host:~/firstprog$ echo 46300 | ./square
2143690000
avst@host:~/firstprog$ echo 46340 | ./square
2147395600
avst@host:~/firstprog$ echo 46341 | ./square
-2147479015
avst@host:~/firstprog$
```

но это всё же лучше, чем то, что было.

Можно расширить разрядность числа ещё сильнее, применив тип `int64`, использующий знаковые 64-битные числа[14]. После замены `longint` на `int64`

[14] Если числа типа `longint` есть практически в любой реализации Паскаля, то тип `int64` — это особенность избранной нами реализации (то есть Free Pascal), так что, если вы попробуете использовать его в других версиях Паскаля, велика вероятность, что его там не окажется.

и перекомпиляции наша программа сможет возвести в квадрат прямо-таки «огромные» числа:

```
avst@host:~/firstprog$ echo 3000000000 | ./square
9000000000000000000
```

хотя, конечно, кто ищет — тот всегда найдёт; разумеется, максимально возможное число есть и для `int64`:

```
avst@host:~/firstprog$ echo 3037000499 | ./square
9223372030926249001
avst@host:~/firstprog$ echo 3037000500 | ./square
-9223372036709301616
```

Последнее, что мы можем сделать для расширения диапазона чисел — это заменить знаковые числа беззнаковыми. Много это не даст, мы выгадаем всего один бит, но чисел разрядности, превышающей 64, в Паскале (во всяком случае, во Free Pascal) нет. Итак, меняем `int64` на `qword` (от слов *quadro word*, то есть «учетверённое слово»; под «словом» на архитектурах семейства x86 традиционно понимается 16 бит) и пробуем:

```
avst@host:~/firstprog$ echo 3037000500 | ./square
9223372037000250000
```

Поскольку максимально возможное значение нашей переменной теперь составляет $2^{64} - 1 = 18446744073709551615$, мы можем предсказать, на каком числе программа даст сбой. Квадрат числа 2^{32} составляет 2^{64}, что на единицу больше допустимого. Следовательно, наибольшее число, которое наша программа ещё сможет возвести в квадрат — это $2^{32} - 1 = 4294967295$. Проверяем:

```
avst@host:~/firstprog$ echo 4294967295 | ./square
18446744065119617025
avst@host:~/firstprog$ echo 4294967296 | ./square
0
```

Вот такой вот несколько неожиданный эффект от переполнения, или, говоря более строго, от *переноса в несуществующий разряд*, ведь мы на сей раз имеем дело с беззнаковыми. Помните анекдот про Змея Горыныча?

2.3. Управление последовательностью выполнения

2.3.1. Простая последовательность операторов

Все программы, которые мы писали до сих пор, выполнялись последовательно, оператор за оператором. Интересно, что тривиальная, в сущности, идея *последовательного выполнения инструкций* далеко не всем людям покоряется сразу и без боя; если вы не чувствуете себя уверенно, попробуйте написать что-нибудь вроде такого:

```
program sequence;
begin
    writeln('First');
    readln;
    writeln('Second');
    readln;
    writeln('Third')
end.
```

Поясним, что оператор `readln` работает примерно так же, как и уже знакомый нам `read` с тем отличием, что, прочитав всё, что требовалось, он обязательно дожидается окончания строки на вводе. Поскольку в нашем примере мы не указали параметров для этого оператора, он только это — ожидание окончания строки на вводе, то есть, попросту, ожидание нажатия клавиши Enter — и будет делать. Если нашу программу откомпилировать и запустить, она напечатает слово «`First`» и остановится в ожидании; пока вы не нажмёте Enter, больше ничего происходить не будет. В программе отработал её первый оператор и начал выполняться второй; он не завершится, пока на вводе не будет прочитан символ перевода строки.

Нажав Enter, вы увидите, что программа «ожила» и напечатала слово «`Second`», после чего снова остановилась. Когда вы нажали Enter, первый из двух `readln`'ов завершился, потом отработал второй `writeln`, который как раз и напечатал слово «`Second`»; после этого начал выполнение второй `readln`. Как и первый, он будет выполняться, пока вы не нажмёте Enter. Нажав Enter во второй раз, вы увидите, что программа напечатала «`Third`» и завершилась: это сначала завершился предпоследний оператор (`readln`), а затем отработал последний.

Во многих учебниках по информатике и программированию, особенно школьных, буквально все примеры программ заканчиваются этим вот `readln`'ом; временами `readln` в конце программы становится столь привычен, что и ученики, и даже некоторые учителя начинают воспринимать его «как мебель», совершенно забывая, для чего он, в сущности, нужен в конце программы. Весь этот полушаманский кавардак начался с того, что в большинстве школ в качестве учебного пособия используются системы семейства Windows, а поскольку нормальную программу для Windows написать очень сложно, программы пишутся «консольные», а точнее — просто DOS'овские. Запускать программы ученикам, разумеется, предлагают исключительно из-под интегрированной среды типа Turbo Pascal или чем там кто богат, ну а озаботиться правильной настройкой этой среды при этом никто не считает нужным. В результате при запуске программы, созданной в интегрированной среде, операционная система открывает для выполнения такой программы окошко эмулятора MS-DOS, но это окошко автоматически исчезает, как только программа завершится; естественно, мы просто физически не успеваем прочитать, что наша программа напечатала.

Отметим, что эту «проблему» можно побороть самыми разными способами — настроить среду так, чтобы окно не закрывалось, или же просто запустить сеанс командной строки и выполнять откомпилированные программы из него;

к сожалению, вместо этого учителя предпочитают показывать учениками, как вставлять в программы совершенно не относящийся к делу оператор, который в действительности нужен, чтобы окошко не закрывалось, пока пользователь не нажмёт Enter; впрочем, этого ученикам обычно не объясняют.

К счастью, мы с вами не используем ни Windows, ни интегрированные среды, так что подобные проблемы нас не касаются. Отметим заодно, что нелепый `readln` в конце каждой программы — далеко не единственная проблема, возникающая при использовании интегрированных сред. Например, привыкнув запускать программу на исполнение из-под интегрированной среды нажатием соответствующей клавиши, ученики *упускают из виду понятие исполняемого файла, а вместе с ним компилятор и роль, которую он играет* — многие вчерашние школьники уверены, что компиляция нужна, чтобы проверить программу на наличие ошибок, и больше, собственно говоря, низачем.

В качестве самостоятельного упражнения возьмите любой сборник англоязычных[15] стихов и напишите программу, которая будет печатать какой-нибудь сонет строчка за строчкой, каждый раз дожидаясь нажатия Enter.

Последовательность выполнения операторов в программе часто изображают схематически в виде так называемых ***блок-схем***. Блок-схема для простой последовательности операторов показана на рис. 2.1 (слева). Отметим, что обычные действия на блок-схемах традиционно изображают в виде прямоугольника, начало и конец рассматриваемого фрагмента программы обозначают маленьким кружком, а *проверку условия* — ромбиком; об этом — в следующем параграфе.

2.3.2. Конструкция ветвления

Единожды и навсегда заданная последовательность действий, какую мы использовали до сих пор, годится только для совсем тривиальных задач, которые к тому же приходится специально подбирать. В более сложных случаях последовательность приходится разветвлять на разные варианты, некоторые фрагменты программы повторять несколько (или даже очень много) раз подряд, временно перескакивать в другие места программы, с тем чтобы потом вернуться обратно, и так далее. Пожалуй, самая простая конструкция, нарушающая жесткую последовательность исполнения операторов — это так называемое ***ветвление***; при его выполнении сначала проверяется некоторое ***условие***, которое может оказаться истинным или ложным, причём во время написания программы мы не знаем, каким это условие окажется во время её исполнения (чаще всего бывает и так и так, причём за время одного

[15] Очень важно, чтобы стихи были именно англоязычные. Применение символов кириллицы в текстах программ недопустимо, несмотря на то, что компиляторы такое обычно позволяют. Грамотное оформление программы, способной «говорить по-русски», требует изучения возможностей специальных библиотек, позволяющих создавать многоязычные программы; все не-английские сообщения при этом находятся не в самой программе, а в специальных внешних файлах.

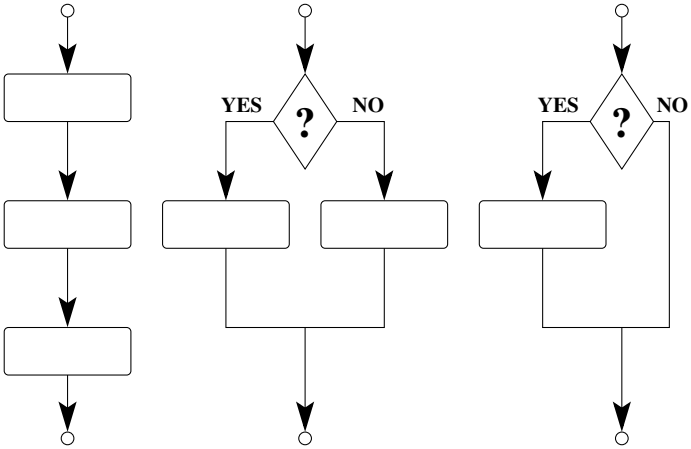

Рис. 2.1. Блок-схемы для простого следования, полного и неполного ветвления

исполнения программы). Ветвление может быть полным или неполным. При полном ветвлении (рис. 2.1, в центре) в программе указывается один оператор для выполнения при истинном условии и другой — для выполнения, если условие окажется ложным; при неполном ветвлении (рис. 2.1, справа) указывается всего один оператор, и его выполняют лишь в том случае, если условие оказалось истинным.

В языке Паскаль простейшие случаи ветвления задаются с помощью ***оператора if***, который несколько отличается от операторов, встречавшихся нам до сей поры. Дело в том, что этот оператор *сложный*: он содержит внутри себя другие операторы, в том числе, кстати, может содержать и другой оператор `if`. Начнём с простого примера: напишем программу, которая вычисляет модуль[16] введённого числа. Как известно, модуль равен самому числу, если число неотрицательное, а для отрицательных чисел модуль получается сменой знака. Для простоты картины будем работать с целыми числами. Программа будет выглядеть так:

```
program modulo;
var
    x: integer;
begin
    read(x);
    if x > 0 then
        writeln(x)
    else
```

[16]Вообще-то в Паскале есть встроенная функция для вычисления модуля, но мы здесь этот факт проигнорируем.

```
        writeln(-x)
end.
```

Как видим, здесь сначала выполняется чтение числа; прочитанное число размещается в переменной `x`. Затем в случае, **если** введённое число строго больше нуля (*выполняется условие* `x > 0`), то печатается само это число, в противном же случае печатается значение выражения `-x`, то есть число, полученное из исходного сменой знака.

Примечательно здесь то, что конструкция

```
if x > 0 then
    writeln(x)
else
    writeln(-x)
```

целиком представляет собой **один оператор**, но *сложный*, поскольку в нём *содержатся* операторы `writeln(x)` и `writeln(-x)`.

Если говорить строже, то оператор `if` составляется следующим образом. Сначала пишется ***ключевое слово*** `if`, с помощью которого мы сообщаем компилятору, что сейчас в нашей программе будет конструкция ветвления. Затем записывается ***условие***, представляющее собой так называемое ***логическое выражение***; такие выражения подобны простым арифметическим выражениям, но результатом их вычисления является не число, а ***логическое значение*** (или *значение типа* ***`boolean`***), то есть *истина* (`true`) или *ложь* (`false`). В нашем примере логическое выражение образовано ***операцией сравнения***, которая обозначена знаком «`>`» («больше»).

После условия необходимо написать ключевое слово `then`; по нему компилятор узнает, что наше логическое выражение кончилось. Далее идёт *оператор*, который мы хотим выполнять в случае, если условие выполнено; в нашем случае таким оператором выступает `writeln(x)`. В принципе, условный оператор (то есть оператор `if`) на этом можно закончить, если нам нужно неполное ветвление; но если мы хотим сделать ветвление полным, мы пишем ключевое слово `else`, а после него — ещё один оператор, задающий действие, которое мы хотим выполнить, если условие оказалось ложно. Синтаксис оператора `if` можно выразить следующим образом:

`if` <*условие*> `then` <*оператор1*> [`else` <*оператор2*>]

Квадратными скобками здесь обозначена *необязательная часть*.

Отметим, что наше вычисление модуля можно написать и с неполным ветвлением, причём даже несколько короче:

```
program modulo;
var
    x: integer;
begin
```

```
    read(x);
    if x < 0 then
        x := -x;
    writeln(x)
end.
```

Здесь условие в операторе `if` мы изменили на «`x` строго меньше нуля» и в этом случае заносим в переменную `x` число, полученное из старого значения `x` заменой знака; если же `x` было неотрицательным, ничего не происходит. Затем, уже безотносительно того, ложным оказалось условие или истинным, выполняется оператор `writeln`, который печатает то, что в итоге оказалось в переменной `x`.

Обратите внимание на то, как в наших примерах расставлены отступы. Операторы, *вложенные в* ***`if`***, то есть являющиеся его частью, сдвинуты вправо *относительно того, во что они вложены*, на уже знакомые нам четыре пробела; всего при избранном нами стиле получается восемь пробелов — *два размера отступа*[17]. Сами эти операторы в нашем примере оказываются на *втором уровне вложенности*.

Отметим ещё один важный момент. Начинающие очень часто делают достаточно характерную ошибку — ставят точку с запятой перед `else`; программа после этого не проходит компиляцию. Дело в том, что в Паскале точка с запятой, как уже говорилось, *отделяет один оператор от другого*; увидев точку с запятой, компилятор считает, что очередной оператор закончился, а в данном случае в качестве такового выступает `if`. Поскольку слово `else` не имеет смысла само по себе и может фигурировать в программе только как часть оператора `if`, а этот оператор с точки зрения компилятора уже закончился, встреченное затем слово `else` приводит к ошибке. Повторим ещё раз: **перед `else` в составе оператора `if` точка с запятой не ставится!**

2.3.3. Составной оператор

В предыдущем параграфе было сказано, что действия, выполняемые оператором `if` в случае истинного или ложного значения условия, задаются *одним оператором* (в примере предыдущего параграфа это были операторы `writeln` и присваивание). Но что делать, если требуется выполнить несколько действий?

Приведём классический пример такой ситуации. Допустим, у нас в программе есть переменные `a` и `b` какого-нибудь числового типа, и нам зачем-то очень нужно сделать так, чтобы значение в переменной `a` не превосходило значение в `b`, а если всё-таки превосходит, то значения надо поменять местами. Для временного хранения нам потребуется

[17] Напомним, что избранный вами размер отступа может составлять два пробела, три, четыре или ровно один символ табуляции.

третья переменная (пусть она называется `t`); но чтобы поменять местами значения двух переменных через третью, там нужно сделать *три* присваивания, тогда как в теле `if` оператор предусмотрен только один.

Проблема решается с помощью так называемых ***операторных скобок***, в роли которых в Паскале используются уже знакомые нам ключевые слова `begin` и `end`. Заключив произвольную последовательность операторов в эти «скобки», мы превращаем всю эту последовательность вместе со скобками в *один* так называемый ***составной оператор***. С учётом этого наша задача с упорядочением значений в переменных `a` и `b` решается следующим фрагментом кода:

```
if a > b then
begin
    t := a;
    a := b;
    b := t
end
```

Подчеркнём ещё раз, что вся конструкция, состоящая из слов `begin`, `end` и всего, что между ними заключено, представляет собой **один оператор** — тоже, конечно, относящийся к «сложным», поскольку включает в себя другие операторы.

Обратите внимание на оформление фрагментов кода, содержащих составной оператор! Существует три различных допустимых способа оформления конструкции, показанной выше; в нашем примере мы снесли `begin` на строчку, следующую за заголовком оператора `if`, но сдвигать его относительно `if` не стали; что касается `end`, его мы написали в той же колонке, где *начинается* конструкция, которую этот `end` закрывает.

Второй популярный способ оформления такой конструкции отличается тем, что `begin` оставляют на одной строке с заголовком `if`:

```
if a > b then begin
    t := a;
    a := b;
    b := t
end
```

Обратите внимание, что `end` остался там же, где был! Можно считать, что он закрывает `if`; можно по-прежнему настаивать, что он закрывает `begin`, но горизонтальная позиция слова `end` в любом случае должна совпадать с позицией **строки**, где находится то (чем бы оно ни было), что закрывается данным `end`'ом. Иначе говоря, `end` должен быть снабжён в точности таким же отступом, каким снабжена строка, содержащая то, что этот `end` закрывает. Выполнение этого правила позволяет

«схватить» общую структуру программы расфокусированным взглядом, а это при работе с исходным текстом очень важно.

Существует, хотя и гораздо реже используется, третий вариант оформления, при котором слово `begin` сдвигается на отдельный уровень вложенности, а то, что вложено в составной оператор, сдвигается ещё дальше. Выглядит это примерно так:

```
if a > b then
    begin
        t := a;
        a := b;
        b := t
    end
```

Рекомендовать использование такого стиля мы не будем в силу целого ряда причин, но в принципе он допустим.

2.3.4. Логические выражения и логический тип

Коль скоро мы начали пользоваться логическими («булевскими») выражениями, попробуем обсудить их более подробно. До сих пор в примерах мы рассматривали в качестве логических только операции «больше» и «меньше», обозначаемые соответственно знаками «`>`» и «`<`»; кроме них Паскаль предусматривает операции «равно» («`=`»), «не равно» («`<>`»), «больше или равно» («`>=`»), «меньше или равно» («`<=`») и некоторые другие, которые мы рассматривать не будем.

Логические выражения, как и арифметические, *вычисляются*: если, к примеру, у нас есть переменная `x`, имеющая тип `integer`, то выражения `x + 1` и `x > 1` различаются лишь *типом значения*: первое имеет тот же тип `integer`, тогда как второе — тип `boolean`; иначе говоря, если выражение `x + 1` может дать в качестве результата произвольное целое число, то выражение `x > 1` — только одно из двух значений, обозначаемых `true` и `false`, но это значение — результат сравнения — *вычисляется*, как и результат сложения.

Как мы уже упоминали, `boolean` может выступать в роли типа переменной, то есть мы можем описать переменную, хранящую логическое значение. Простое упоминание такой переменной само по себе представляет собой *логическое выражение*, и его можно использовать, например, в качестве условия в операторе `if`; переменным этого типа можно *присваивать* значения — разумеется, логические, то есть если слева от знака присваивания мы поставим переменную типа `boolean`, то справа нам придётся написать логическое выражение.

В частности, мы могли бы переписать пример с вычислением модуля, используя логическую переменную для хранения признака того, имеем ли мы дело с отрицательным числом. Выглядеть это будет так:

```
program modulo;
var
    x: integer;
    negative: boolean;
begin
    read(x);
    negative := x < 0;
    if negative then
        x := -x;
    writeln(x)
end.
```

Здесь переменная `negative`, после присваивания будет содержать значение `true`, если введённое пользователем число (значение переменной `x`) оказалось меньше нуля, и `false` в противном случае. После этого мы используем переменную `negative` в качестве условия в операторе `if`.

Над логическими значениями язык Паскаль позволяет производить *операции*, соответствующие основным функциям алгебры логики (см. § 1.5.3). Эти операции обозначаются ключевыми словами `not` (отрицание), `and` (логическое «и», конъюнкция), `or` (логическое «или», дизъюнкция) и `xor` («исключающее или»). Например, с помощью оператора присваивания `flag := not flag` мы могли бы изменить значение логической переменной `flag` на противоположное; проверить, содержит ли целочисленная переменная `k` число, записываемое одной цифрой, можно с помощью логического выражения `(k >= 0) and (k <= 9)`. **Обратите внимание на скобки!** Дело здесь в том, что в Паскале *приоритет* логических связок, в том числе операции `and`, выше, чем приоритет операций сравнения, подобно тому как приоритет умножения и деления выше, чем приоритет сложения и вычитания; если не поставить скобки, то выражение `k >= 0 and k <= 9` компилятор Паскаля «разберёт» в соответствии с приоритетами так, как если бы мы написали `k >= (0 and k) <= 9`, что вызовет ошибку при компиляции.

Паскаль позволяет записывать сколь угодно сложные логические выражения; например, если переменная `c` имеет тип `char`, то выражение

```
((c >= 'A') and (c <= 'Z')) or ((c >= 'a') and (c <= 'z'))
```

позволит узнать, является ли её текущее значение латинской буквой. В принципе, здесь можно было бы обойтись без скобок вокруг `and`, поскольку приоритет `and` всё равно выше, чем `or`, но это тот случай, когда ликвидация избыточных скобок ничуть не добавит выражению ясности.

2.3.5. Понятие цикла; оператор `while`

Под ***циклом*** в программировании понимают некую последовательность действий, которая при работе программы выполняется (повторя-

ется) сколько-то раз подряд. Сама такая последовательность действий, представленная одним или несколькими операторами, называется ***телом цикла***, а каждое отдельное её выполнение называют ***итерацией***; можно сказать, что выполнение всего цикла состоит из некоторого количества итераций. Тело цикла бывает коротким, а бывает и достаточно длинным, количество итераций может быть совсем небольшим (две, три, одна или даже ни одной), а может достигать многих миллиардов; ещё более интересен тот факт, что на момент начала выполнения цикла количество предстоящих итераций может быть заранее известно, а может определяться в ходе выполнения цикла. Встречаются даже такие циклы, которые выполняются «до бесконечности» — точнее, до тех пор, пока программу, выполняющую такой цикл, кто-нибудь не остановит.

Обычно в программе при выполнении цикла должно что-то изменяться от итерации к итерации, в противном случае цикл никогда не кончится; иногда, впрочем, такой бесконечный цикл организуется намеренно, но это скорее исключение. В простейшем случае между итерациями изменяется значение какой-нибудь переменной, а заданное для конкретного цикла логическое выражение, определяющее, продолжать цикл или прекратить, зависит от этой переменной.

В Паскале есть три разных ***оператора цикла***; самым простым из них можно считать ***цикл*** `while`. В заголовке этого оператора указывается логическое выражение, которое будет вычисляться перед выполнением каждой итерации цикла; если результат вычисления окажется ложным, выполнение цикла будет немедленно завершено, если же получится «истина», то будет выполнено *тело цикла*, заданное одним (возможно, составным) оператором. Начало конструкции отмечается ключевым словом `while` (англ. *пока*), тело отделяется от условия ключевым словом `do`. Синтаксис оператора `while` можно представить следующим образом:

`while` <*условие*> `do` <*оператор*>

Пусть нам, к примеру, нужно выдать на экран всё ту же надпись «`Hello, world!`», но не один раз, а двадцать. Естественно, это можно и нужно сделать с помощью цикла, и цикл `while` для этой цели вполне подойдёт[18], нужно только придумать, как задать условие цикла и как изменять что-то в состоянии программы таким образом, чтобы цикл выполнился ровно двадцать раз, а на двадцать первый условие оказалось ложным. Самый простой способ добиться этого — попросту *считать*, сколько раз цикл уже выполнился, и когда он выполнится двадцать раз, больше его не выполнять. Организовать такой подсчёт можно, введя целочисленную переменную для хранения числа, равного *количеству итераций, которые уже отработали*. Первоначально в эту переменную

[18]Забегая вперёд, отметим, что как раз для таких ситуаций, когда количество итераций точно известно при входе в цикл, в Паскале предусмотрена другая управляющая конструкция — цикл `for`.

мы занесём ноль, а в конце каждой итерации будем увеличивать её на единицу; в результате в каждый момент времени эта переменная будет равна количеству выполненных к настоящему моменту итераций. Такую переменную часто называют ***счётчиком цикла***.

Кроме увеличения переменной на единицу, в теле цикла нужно сделать ещё одно действие — собственно то, ради чего всё и затевалось, то есть печать строки. Получается, что операторов в теле цикла нам нужно два, а синтаксис оператора `while` предусматривает в качестве тела цикла только один оператор; но мы уже знаем, что это не проблема — достаточно объединить все операторы, которые нам нужны, в один *составной* оператор с помощью операторных скобок `begin` и `end`. Целиком наша программа будет выглядеть так:

```
program hello20;
var
    i: integer;
begin
    i := 0;
    while i < 20 do
    begin
        writeln('Hello, world!');
        i := i + 1
    end
end.
```

Обратите внимание на то, что в теле цикла мы *сначала* расположили оператор печати, и лишь потом — оператор присваивания, увеличивающий переменную `i` на единицу. Для данной конкретной задачи ничего бы не изменилось, если бы мы поменяли их местами; но делать так всё же не следует. Как показывает практика, лучше — безопаснее в плане возможных ошибок — всегда следовать одному достаточно простому соглашению: **подготовка значений переменных для первой итерации цикла `while` должна происходить непосредственно перед циклом, а подготовка значений для *следующей* итерации должна располагаться в самом конце тела цикла**. В данном случае подготовка к первой итерации состоит в присваивании нуля счётчику цикла, а подготовка к следующей — в увеличении счётчика на единицу; оператор `i := 0` мы поставили перед самым `while`, а оператор `i := i + 1` — последним в его теле.

Можно подойти к этому вопросу по-другому. Коль скоро переменная `i` хранит число *напечатанных к данному моменту строчек* (сначала ноль, потом каждый раз на единицу больше), то вполне логично *сначала* напечатать очередную строку, и лишь потом учесть этот факт, увеличив переменную на единицу.

Значение счётчика цикла можно использовать не только в условии, как мы это сделали в программе `hello20`, но и, при необходимости, в

теле цикла. Допустим, нас заинтересовали квадраты целых чисел от 1 до 100; напечатать их можно, например, так:

```
program square100;
var
    i: integer;
begin
    i := 1;
    while i <= 100 do
    begin
        writeln(i * i);
        i := i + 1
    end
end.
```

Пользоваться результатом этой программы будет не очень удобно, поскольку она каждое число расположит на отдельной строке. Мы можем усовершенствовать её, заменив `writeln` на `write`; но если так поступить, не предприняв никаких дополнительных мер, то есть просто убрать буквы `ln` и в таком виде запустить программу, результат нас может совершенно обескуражить:

```
avst@host:~/work$ ./square100
149162536496481100121144169196225256289324361400441484529576625676729784841900 96
110241089115612251296136914441521160016811764184919362025211622092304240125002 60
127042809291630253136324933643481360037213844396940964225435644894624476149005 04
151845329547656255776592960846241640065616724688970567225739675697744792181008 28
184648649883690259216940996049801100 00avst@host:~/work$
```

Дело в том, что оператор `write` исполняет нашу волю **буквально**: если мы потребовали напечатать число, то он выдаст на печать цифры, составляющие десятичную запись этого числа, и **больше ничего** — ни пробелов, ни каких-либо других разделителей. Внимательно посмотрев на вывод, мы можем заметить, что цифры, составляющие запись чисел 1, 4, 9, 16 и т. д., никуда не делись, просто числа ничем не отделены одно от другого.

Решить эту проблему очень просто, достаточно сказать оператору `write`, что мы желаем, чтобы он после каждого числа выводил ещё и символ пробела. Кроме того, в конце программы, то есть уже после цикла, желательно добавить оператор `writeln`, чтобы программа, прежде чем завершиться, перевела строку на печати, и приглашение командной строки после её завершения появилось бы на новой строке, а не сливалось с напечатанными числами. Целиком программа будет выглядеть так:

```
program square100;
var
    i: integer;
begin
    i := 1;
    while i <= 100 do
    begin
        write(i * i, ' ');
```

```
        i := i + 1
    end;
    writeln
end.
```

Рассмотрим теперь пример такого цикла, для которого мы заранее не знаем количество итераций. Допустим, мы пишем некую программу, которая в какой-то момент должна спросить у пользователя его год рождения, и нам при этом нужно проверить, действительно ли введённое число может представлять собой год рождения. На момент написания этой книги (2016 год) на Земле оставалось всего два человека, про которых достоверно известно, что они родились раньше 1900 года; будем считать, что год рождения пользователя заведомо не может быть меньше, чем 1900. По аналогичным причинам будем считать, что год рождения пользователя не может превышать 2013, поскольку трёхлетние дети не умеют пользоваться компьютером; если к тому времени, когда вы эту книгу читаете, прошло достаточно времени, вы можете самостоятельно скорректировать эти значения.

Так или иначе, нам нужно попросить пользователя ввести его год рождения; если ввод нас не устраивает, необходимо сказать пользователю, что он, по-видимому, ошибся, и попросить повторить ввод. В секции описаний мы можем предусмотреть переменную `year`:

```
var
    year: integer;
```

Что касается самого диалога с пользователем, то реализовать его можно так:

```
    write('Please type in your birth year: ');
    readln(year);
    while (year < 1900) or (year > 2013) do
    begin
        writeln(year, ' is not a valid year!');
        write('Please try again: ');
        readln(year)
    end;
    writeln('The year ', year, ' is accepted. Thank you!')
```

С такой программой может состояться, например, следующий диалог:

```
Please type in your birth year: 1755
1755 is not a valid year!
Please try again: -500
-500 is not a valid year!
Please try again: 2050
```

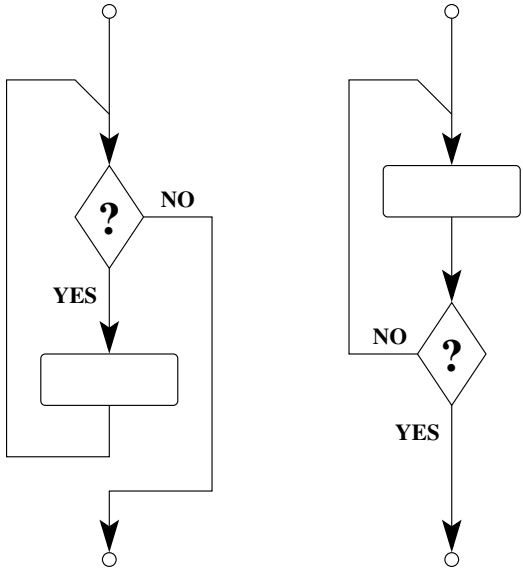

Рис. 2.2. Блок-схемы циклов с предусловием и постусловием

```
2050 is not a valid year!
Please try again: 1974
The year 1974 is accepted. Thank you!
```

Заметим, что здесь цикл может не выполниться ни одного раза — если пользователь сразу же введёт нормальный год; с другой стороны, пользователь может попасться упрямый, так что мы в действительности не можем знать, каково *максимальное* число итераций нашего цикла. Можно предположить, что, скажем, на миллиард итераций терпения пользователя всё же не хватит, но какое конкретно число указать в качестве верхней границы? Сто? Тысячу? Предположение, что на 1000 итераций пользователя хватить ещё может, а на 1001 — уже нет, выглядит достаточно нелепо, и точно так же нелепо будет выглядеть в этой роли любое другое конкретное число; проще вообще не строить на этот счёт никаких предположений.

2.3.6. Цикл с постусловием; оператор `repeat`

В операторе `while`, которому был посвящён предыдущий параграф, сначала проверяется условие, и только после этого, возможно, выполняется первая итерация. Такие конструкции в программировании называются ***циклами с предусловием***.

Кроме них, при написании программ иногда используются ***циклы с постусловием***, в которых сначала выполняется тело цикла, и только

после этого проверяется, следует ли выполнять его снова; таким образом, в циклах с постусловием тело выполняется по меньшей мере один раз. Язык Паскаль предусматривает для циклов с постусловием специальный оператор, который задаётся ключевыми словами `repeat` и `until`; операторы, составляющие тело цикла, записыватся между этими словами, причём таких операторов может быть сколько угодно, применение операторных скобок здесь не требуется; после слова `until` записывается *условие выхода из цикла* — логическое выражение, ложное значение которого указывает на необходимость продолжать цикл, а истинное — на то, что цикл пора прекращать. Блок-схемы циклов с предусловием и постусловием показаны на рис. 2.2.

Например, если бы нам по каким-то причинам не требовалось, как в примере на стр. 231, выстраивать диалог с пользователем, а просто было бы нужно вводить с клавиатуры целые числа, пока очередное из них не попадёт в отрезок от `1900` до `2013`, мы могли бы сделать это так:

```
repeat
    readln(year)
until (year >= 1900) and (year <= 2013)
```

Рассмотрим другой пример. Следующий цикл вводит числа с клавиатуры и складывает их, пока общая сумма не окажется больше `1000`:

```
sum := 0;
repeat
    readln(x);
    sum := sum + x
until sum > 1000
```

Синтаксис оператора `repeat` можно представить следующим образом:

`repeat` <*операторы*> `until` <*условие*>

Обычно в программах оператор `repeat` встречается гораздо реже, чем `while`, но знать о его существовании в любом случае необходимо.

2.3.7. Арифметические циклы и оператор `for`

Вернёмся к примерам, приведённым в начале § 2.3.5; напомним, там мы писали циклы, чтобы выдать одну и ту же надпись двадцать раз и чтобы напечатать квадраты всех целых чисел от одного до ста. Циклы, которые мы написали для решения этих простых задач, обладают одним очень важным свойством: *на момент входа в цикл точно известно, сколько раз он будет выполняться, и нужное количество итераций обеспечивается путём их подсчёта в целочисленной переменной.* Такие циклы называют ***арифметическими***.

Арифметические циклы встречаются настолько часто, что во многих языках программирования, включая Паскаль, для них предусмотрен

специальный оператор; в Паскале такой оператор называется `for`. Как это часто бывает, рассказать о нём будет проще, если сначала привести пример, а потом дать пояснения, поэтому мы начнём с того, что перепишем программы `hello20` и `square100`, используя оператор арифметического цикла. Начнём с первой:

```
program hello20_for;
var
    i: integer;
begin
    for i := 1 to 20 do
        writeln('Hello, world!')
end.
```

Конструкция `for i := 1 to 20 do` означает, что в качестве ***переменной цикла*** будет использоваться переменная `i`, её начальное значение будет `1`, финальное значение — `20`, то есть она должна пробежать все значения от `1` до `20`, и для каждого такого значения будет выполнено тело цикла. Поскольку таких значений 20, тело будет выполнено двадцать раз, что нам и требуется. Если сравнить получившуюся программу с той, что мы писали на стр. 229, можно заметить, что её текст получился гораздо компактнее; более того, для человека, который уже привык к синтаксису `for`, такой вариант существенно проще понять.

Перепишем теперь программу `square100`, взяв за основу вариант, печатающий числа через пробел. С использованием цикла `for` того же самого эффекта можно достичь так:

```
program square100_for;
var
    i: integer;
begin
    for i := 1 to 100 do
        write(i * i, ' ');
    writeln
end.
```

Как видим, в теле цикла `for` можно использовать значение переменной цикла. Отметим сразу же, что к этому значению можно обращаться, а вот **менять его ни в коем случае нельзя**. Менять переменную цикла во время выполнения цикла — это прерогатива самого оператора `for`, попытки вмешаться в его работу могут привести к непредсказуемым последствиям. Кроме того, с переменной цикла связано ещё одно ограничение: **после завершения цикла `for` значение переменной цикла считается неопределённым**, то есть мы не должны предполагать, что эта переменная будет равна какому-то конкретному числу. Конечно, *какое-то* значение там будет, но оно может зависеть от версии

компилятора и даже от того, в каком месте в программе встретился цикл. Попросту говоря, создатели компилятора не обращают никакого внимания на то, какое значение оставить в переменной цикла после его завершения, и могут оставить там всё что угодно.

В обоих наших примерах переменная цикла изменялась в сторону увеличения, но можно заставить её пробегать значения в обратном направлении, от большего к меньшему. Для этого слово `to` заменяют словом `downto`. Например, программа

```
program countdown;
var
    i: integer;
begin
    for i := 10 downto 1 do
        write(i, '... ');
    writeln('start!')
end.
```

напечатает строку

```
10... 9... 8... 7... 6... 5... 4... 3... 2... 1... start!
```

Формально синтаксис оператора `for` можно представить следующим образом:

`for` <*перем*> `:=` <*нач*> `to`|`downto` <*кон*> `do` <*оператор*>

Здесь <*перем*> — это имя целочисленной переменной, <*нач*> — начальное значение, <*кон*> — конечное значение. Оба этих значения могут задаваться не только явным образом написанными числами, как в наших примерах, но и произвольными выражениями, лишь бы результатом этих выражений было целое число. Выражения будут вычислены **один раз перед началом выполнения цикла**, так что, если в них входят переменные и значения этих переменных изменятся во время выполнения цикла, то на работу самого цикла это уже никак не повлияет.

Если после вычисления начального и конечного значений оказалось, что конечное значение *меньше* (а для `downto` — наоборот, больше) начального, то это само по себе не ошибка: цикл не выполнится ни одного раза, и в некоторых случаях это свойство можно использовать.

На самом деле переменная цикла, используемая в операторе `for`, может быть не только целочисленной; несколько позже мы введём понятие ***порядкового типа***, которое объединит все целочисленные типы, а также диапазоны, символьный тип, перечислимые типы и логический тип; оператор `for` можно использовать с любым таким типом. Конечно, тип выражений, задающих начальное и конечное значение, обязан совпадать с типом переменной цикла.

2.3.8. Вложенные циклы

Рассмотрим следующую задачу. Нужно напечатать «наклонную черту» размером во весь экран, состоящую из символов «*». Результат должен выглядеть примерно так:

```
*
 *
  *
   *
    *
     *
      *
       *
```

(для экономии места мы показали всего восемь строк, хотя их должно быть 24). Сделать это, в принципе, очень просто: нужно вывести 24 строки, причём в каждой строке сначала напечатать некоторое количество пробелов, а потом выдать «звёздочку» и перевести строку. В самой первой строке мы вообще не печатаем пробелов, сразу выдаём звёздочку; можно считать, что мы печатаем ноль пробелов. В каждой последующей строке печатается на один пробел больше, чем в предыдущей. Если считать, что номера строк у нас идут от 1 до 24, то нетрудно заметить, что в строке с номером n должно быть $n - 1$ пробелов.

Ясно, что выводить строки нужно циклом, по одной итерации на строку. Поскольку на момент входа в цикл мы точно знаем, сколько будет итераций, применить следует цикл `for`. Переменную цикла назовём `n`, её значение будет соответствовать номеру строки. Цикл должен выглядеть примерно так[19]:

```
for n := 1 to 24 do
begin
    { напечатать нужное количество пробелов }
    writeln('*')
end
```

Осталось понять, как организовать печать нужного количества пробелов. Сколько их печатать, мы уже знаем: `n - 1`. Иными словами, загадочное «`{ напечатать нужное количество пробелов }`» в нашем коде нужно заменить на что-то такое, что напечатает нам `n - 1` пробел.

[19] Фигурные скобки в Паскале означают *комментарий*, то есть такой фрагмент текста, который предназначен исключительно для читателя-человека, а компилятором должен быть полностью проигнорирован. Здесь и далее мы иногда будем писать комментарии по-русски. В учебном пособии такая вольность допустима, но в реальных программах так поступать не следует ни в коем случае: во-первых, символы кириллицы не входят в ASCII; во-вторых, всемирным языком общения программистов является английский. Если комментарии в программе вообще пишутся, то они должны быть написаны по-английски, причём по возможности без ошибок; в противном случае лучше их вообще не писать.

Как несложно догадаться, для этого тоже нужен цикл, и тоже арифметический: мы ведь знаем на момент его начала, сколько должно быть итераций. Так мы приходим к концепции ***вложенных циклов***.

Ясно, что во вложенном цикле нужно задействовать другую переменную цикла, чтобы циклы не конфликтовали; в конце концов, пока внешний цикл не завершился, никто не вправе менять его переменную, в том числе не вправе это делать и внутренний цикл. Описав для внутреннего цикла переменную `m`, мы получим следующую программу:

```
program star_slash;
var
    n, m: integer;
begin
    for n := 1 to 24 do
    begin
        for m := 1 to n - 1 do
            write(' ');
        writeln('*')
    end
end.
```

Рассмотрим задачу чуть более сложную — вывести на экран фигуру примерно такого вида:

```
     *
    * *
   *   *
  *     *
   *   *
    * *
     *
```

Эту фигуру часто называют «алмазом» (англ. *diamond*). Высоту фигуры мы в этот раз прочитаем с клавиатуры, то есть попросим пользователя сказать нам, какой высоты «алмаз» он хочет увидеть. Дальнейшее потребует некоторого анализа.

Прежде всего заметим, что высота нашей фигуры есть всегда число нечётное, так что, если пользователь введёт чётное число, то придётся попросить его повторить ввод; то же самое, по-видимому, следует сделать, если введённое число окажется отрицательным. Нечётное число, как известно, представляется в виде $2n+1$, где n — целое; верхняя часть нашей фигуры будет состоять из $n+1$ строк. Для фигуры, показанной выше, высота составляет семь строк, а n будет равно трём.

Теперь нам нужно понять, сколько пробелов и где нужно напечатать, чтобы получить искомую фигуру. Заметим для начала, что при печати самой первой строки нам придётся выдать n пробелов, при печати второй строки — $n-1$ пробел, и так далее; при печати последней,

$(n+1)$'й строки пробелов не нужно вовсе (можно считать, что мы их печатаем ноль штук).

Чуть сложнее обстоят дела с пробелами после первой звёздочки. В первой строке вообще ничего такого не нужно, там всего одна звёздочка; а вот дальше происходит довольно интересный процесс: во второй строке нужно напечатать один пробел (и после него вторую звёздочку), в третьей строке — уже три пробела, в четвёртой — пять, и так далее, каждый раз на два пробела больше. Несложно догадаться, что для строки с номером k ($k > 1$) нужное количество пробелов выражается формулой $1 + 2(k-2) = 2k - 3$. Интересно, что с некоторой натяжкой можно считать эту формулу «верной» также и для случая $k = 1$, где она даёт -1: если операцию «напечатать m пробелов» доопределить на отрицательные m как «возврат назад на соответствущее число знакомест», получится, что, напечатав первую звёздочку, мы должны будем вернуться обратно на одну позицию и вторую звёздочку напечатать точно поверх первой. Впрочем, так сделать гораздо труднее, чем просто проверить значение k, и если оно равно единице, то после первой звёздочки не печатать больше ни пробелов, ни звёздочек, а сразу перевести строку.

Полностью печать строки с номером k должна выглядеть так: сначала мы печатаем $n+1-k$ пробелов, затем звёздочку; после этого если k равно единице, просто выдаём перевод строки и считаем печать строки оконченной; в противном случае печатаем $2k-3$ пробела, звёздочку и только после этого делаем перевод строки. Всё это нам надо проделать для k от 1 до $n+1$, где n — «полувысота» нашего «алмаза».

После того, как верхняя часть фигуры будет напечатана, нам нужно будет как-то выдать ещё её нижнюю часть. Мы могли бы продолжить нумерацию строк и вывести формулы для количества пробелов в каждой строке с номером $n+1 < k \leqslant 2n+1$, что, в принципе, не так уж сложно; однако можно поступить ещё проще, заметив, что печатаемые теперь строки в точности такие же, как и в верхней части фигуры, то есть мы сначала печатаем такую же строку, как n-я, потом такую же, как $(n-1)$-я, и так далее. Самый простой способ — для каждой строки исполнить ровно такую процедуру печати, как описано параграфом выше, только в этот раз номера строк k у нас пробегут в обратном направлении все числа от n до 1.

Наша программа будет состоять из трёх основных частей: ввод числа, означающего высоту фигуры, печать верхней части фигуры, печать нижней части. Вот её текст (напомним, что слова `div` и `mod` означают деление с остатком и остаток от деления):

```
{ diamond.pas }
program diamond;
var
    n, k, h, i: integer;
```

```
begin
    { ввод числа, пока пользователь не введёт его как надо }
    repeat
        write('Enter the diamond''s height (positive odd): ');
        readln(h)
    until (h > 0) and (h mod 2 = 1);
    n := h div 2;
    { печать верхней части фигуры }
    for k := 1 to n + 1 do
    begin
        for i := 1 to n + 1 - k do
            write(' ');
        write('*');
        if k > 1 then
        begin
            for i := 1 to 2*k - 3 do
                write(' ');
            write('*')
        end;
        writeln
    end;
    { печать нижней части }
    for k := n downto 1 do
    begin
        for i := 1 to n + 1 - k do
            write(' ');
        write('*');
        if k > 1 then
        begin
            for i := 1 to 2*k - 3 do
                write(' ');
            write('*')
        end;
        writeln
    end
end.
```

Легко заметить очень серьёзный недостаток нашей программы: циклы для рисования верхней и нижней частей фигуры различаются только заголовком, а тела в них абсолютно одинаковые. Вообще-то программисты считают, что так делать нельзя: если у нас в программе содержатся две или больше копий одного и того же кода, то, если вдруг мы захотим исправить один из этих фрагментов (например, найдя в нём ошибку), скорее всего их придётся править все, а это ведёт к непроизводительным трудозатратам (что ещё полбеды) и провоцирует ошибки из-за того, что мы часть отредактировали, а часть забыли. Но справиться с этой

проблемой мы сможем, только изучив так называемые ***подпрограммы***, которым будет посвящена следующая глава.

2.4. Подпрограммы

Словом «подпрограмма» программисты называют обособленную (то есть имеющую своё начало, свой конец и даже свои собственные переменные) часть программы, предназначенную для решения какой-то части задачи. В принципе, в подпрограмму можно выделить практически любую часть главной программы или другой подпрограммы; в том месте, где раньше был код, который теперь оказался в подпрограмме, пишется так называемый ***вызов подпрограммы***, состоящий из её имени и (в большинстве случаев) списка передаваемых ей параметров. Через параметры мы можем передать подпрограмме любую информацию, необходимую для её работы.

Вынесение частей кода в подпрограммы позволяет избежать дублирования кода, поскольку одну и ту же подпрограмму можно, однажды написав, вызвать сколько угодно раз из разных частей программы. Указывая при вызове разные значения параметров, можно адаптировать одну и ту же подпрограмму для решения целого семейства схожих проблем, ещё больше сэкономив на объёме кода, который приходится писать.

Опытные программисты знают, что экономия объёма кода — это не единственная причина для применения подпрограмм. Очень часто в программах встречаются такие подпрограммы, которые вызываются только один раз, что, конечно же, не только не сокращает объём написанного кода, но даже наоборот — увеличивает его, ведь оформление подпрограммы само по себе требует написания нескольких лишних строк. Такие «одноразовые» подпрограммы пишут, чтобы *снизить сложность восприятия программы человеком.* Правильным образом выделив фрагменты кода в подпрограммы и заменив их в основной программе их названиями, мы позволяем читателю нашей программы (и, кстати, в основном — самим себе) при работе с основной программой не думать о второстепенных деталях.

Паскаль предусматривает два вида подпрограмм: ***процедуры*** и ***функции***. В принципе, такое деление во многом условно; в некоторых других языках программирования существуют только функции. Однако нам наличие процедур оказывается очень полезно: на их примере оказывается существенно проще объяснить общую идею подпрограмм для случая императивного программирования. Этим мы сейчас и займёмся.

2.4.1. Процедуры

Подпрограммы, будь то процедуры или функции, описываются в программе *между* её заголовком и главной программой, то есть в уже знакомом нам *разделе описаний*. Для создания первого впечатления о предмете приведём очень простой, хотя и странный пример: перепишем нашу самую первую программу `hello` (см. стр. 197), вынеся единственное действие, которое в ней есть, в процедуру. Выглядеть это будет так:

```
program hello_proc;

procedure SayHello;
begin
    writeln('Hello, world!')
end;

begin
    SayHello
end.
```

В этой программе сначала располагается процедура с именем `SayHello`, которая как раз и делает всю работу, а главная программа состоит из единственного действия — ***вызова*** этой процедуры. Во время исполнения программы вызов подпрограммы выглядит следующим образом. Компьютер запоминает[20] адрес того места в памяти, где в программе встретилась инструкция вызова, после чего *передаёт управление* вызываемой подпрограмме, то есть переходит к исполнению её машинного кода. Когда подпрограмма, отработав, завершается, запомненный перед её вызовом ***адрес возврата*** используется, чтобы *вернуть управление* туда, откуда произошёл вызов, точнее, на следующую после вызова инструкцию.

Наш пример иллюстрирует, *что такое* подпрограмма, но не показывает, *зачем* она нужна: в сравнении с оригиналом, с которого мы начали знакомство с Паскалем, новая программа оказалась практически вдвое длиннее и стала гораздо менее понятной, так что кажется, будто мы ничего не выиграли, а наоборот, проиграли. Это так и есть, но лишь по той причине, что мы вынесли в процедуру слишком простое действие.

Вернёмся теперь к программе `diamond` из предыдущего параграфа (см. стр. 238) и попробуем сделать её более понятной. Заметим для начала, что в программе несколько раз встречается цикл для печати заданного количества пробелов; вынесем это действие в процедуру. Поскольку процедура должна печатать любое количество пробелов,

[20] Запоминание адреса возврата из подпрограммы происходит в так называемом аппаратном (машинном) стеке; к этому мы вернёмся в следующей части нашей книги, которая посвящена архитектуре компьютера и языку ассемблера.

которое понадобится в программе, её придётся снабдить *параметром*; параметры подпрограмм в Паскале записываются в круглых скобках сразу после имени подпрограммы в виде списка, очень похожего на список описаний переменных. В нашем случае параметр будет всего один — количество пробелов, которое должно быть напечатано. Процедуру мы назовём `PrintSpaces`, а параметр — `count`; заголовок процедуры получится вот такой:

```
procedure PrintSpaces(count: integer);
```

Вспомним теперь, что для печати заданного количества пробелов нам понадобится цикл `for`, а в нём будет нужна переменная типа `integer` в роли счётчика цикла. Про эту переменную можно точно сказать, что *она никого и ничего не касается за пределами нашей процедуры*, иначе говоря, это такая деталь реализации процедуры, которую не нужно знать, если только не писать и не редактировать саму нашу процедуру. Практически все существующие языки программирования, включая Паскаль, позволяют в таких случаях описывать ***локальные переменные***, которые доступны (и вообще видны) только внутри одной подпрограммы. Для этого подпрограмма имеет свой собственный *раздел описаний*, который, как и раздел описаний главной программы, располагается между заголовком и основной частью, только теперь уже не программы, а подпрограммы.

Основная часть подпрограммы, как и основная часть главной программы, состоит из слова `begin`, последовательности операторов и слова `end`, только здесь после `end` ставится не точка, а точка с запятой. Целиком наша процедура будет выглядеть так:

```
procedure PrintSpaces(count: integer);
var
    i: integer;
begin
    for i := 1 to count do
        write(' ')
end;
```

Подчеркнём, что имена `i` и `count` в этой процедуре **локальные**, то есть они не оказывают никакого влияния на всю остальную программу: мы в любом месте можем, например, описывать переменные (и не только переменные) с такими же именами, причём, возможно, других типов, и ни к каким плохим последствиям это не приведёт.

Описав процедуру, мы можем теперь в любом месте программы вызвать её, написав что-то вроде `PrintSpaces(k)`, и будет напечатано `k` пробелов.

Прежде чем переписывать программу `diamond`, припомним замечание, сделанное сразу после её написания — о том, что тела двух циклов

в этой программе оказались совершенно одинаковыми и что делать так нельзя, но справиться с проблемой невозможно без подпрограмм. Подпрограммы теперь в нашем распоряжении есть, так что заодно исправим этот недостаток. Вспомним, что тела обоих циклов у нас печатали очередную, k-ю строку фигуры «полувысотой» n и делали это так: сначала напечатать $n+1-k$ пробелов, потом звёздочку, далее, если $k > 1$, то напечатать $2k-3$ пробела и ещё одну звёздочку, и в конце перевести строку. Как видим, для выполнения этих шагов нужно знать две величины: k и n; их мы передадим в процедуру через параметры, а саму процедуру назовём `PrintLineOfDiamond`. После замены тел циклов вызовами этой процедуры главная программа станет совсем короткой; вся программа целиком будет выглядеть так:

```
program diamond_proc; { diamondp.pas }

procedure PrintSpaces(count: integer);
var i: integer;
begin
    for i := 1 to count do
        write(' ')
end;

procedure PrintLineOfDiamond(k, n: integer);
begin
    PrintSpaces(n + 1 - k);
    write('*');
    if k > 1 then
    begin
        PrintSpaces(2*k - 3);
        write('*')
    end;
    writeln
end;

var
    n, k, h: integer;
begin
    repeat
        write('Enter the diamond''s height (positive odd): ');
        readln(h)
    until (h > 0) and (h mod 2 = 1);
    n := h div 2;
    for k := 1 to n + 1 do
        PrintLineOfDiamond(k, n);
    for k := n downto 1 do
        PrintLineOfDiamond(k, n);
end.
```

Несмотря на обилие служебных строк (при описании каждой процедуры мы тратим минимум три лишние строки — на заголовок, на слово `begin` и на слово `end`) и пустых строк, которые мы для наглядности вставляем между подпрограммами, новый вариант программы всё же получился на четыре строки короче предыдущего, если же мы сравним длину их текстов в байтах, окажется, что мы сэкономили почти четверть объёма. Впрочем, столь скромная экономия обусловлена лишь примитивностью решаемой задачи; в более сложных случаях экономия за счёт использования подпрограмм может достигать десятков, сотен, тысяч раз, так что создание серьёзных программ без их разбиения на подпрограммы оказывается просто немыслимо.

Прежде чем двигаться дальше, отметим один технический момент. Как и при описании обычных переменных, в описании параметров подпрограммы можно параметры одного типа перечислить через запятую, указав для них тип один раз; именно так мы поступили при описании процедуры `PrintLineOfDiamond`. Если же нам нужны параметры различных типов, то тип указывается для них раздельно, а между описаниями параметров разных типов ставится точка с запятой. Например, если мы захотим усовершенствовать процедуру `PrintSpaces` так, чтобы она могла печатать не только пробелы, но и любые другие символы, мы можем нужный символ передать через параметр:

```
procedure PrintChars(ch: char; count: integer);
var
    i: integer;
begin
    for i := 1 to count do
        write(ch)
end;
```

Здесь параметры `ch` и `count` имеют разные типы, так что в заголовке между их описаниями появилась точка с запятой.

2.4.2. Функции

Кроме процедур, Паскаль предусматривает также другой вид подпрограмм — так называемые ***функции***; на идейном уровне они отличаются от процедур тем, что основной их задачей является (по крайней мере, исходно предполагается) вычисление некоторого значения. В остальном функции очень похожи на процедуры: они тоже состоят из заголовка, секции описаний и основной части, точно так же в них можно описывать локальные переменные и т. д.

Рассмотрим простейший пример — функцию, которая возводит число с плавающей точкой в куб. Выглядит это так:

```
function Cube(x: real): real;
begin
    Cube := x * x * x
end;
```

Как видим, заголовок функции начинается с ключевого слова `function`, дальше, как и для процедуры, записывается имя и список параметров; после него для функции нужно (через двоеточие) указать *тип возвращаемого значения*, то есть, собственно говоря, тип того значения, для вычисления которого предназначена наша функция. В данном случае функция вычисляет число типа `real`.

В отличие от процедуры, вызов функции представляет собой *выражение*, имеющее тип, совпадающий с указанным для функции типом возвращаемого значения. Для нашего примера, в частности, «`Cube(2.7)`» есть выражение типа `real`. Например, в нашей программе может встретиться что-нибудь вроде

```
a := Cube(b+3.5) - 17.1;
```

где `a` и `b` — переменные типа `real`; в ходе вычисления выражения в правой части присваивания управление будет временно отдано функции `Cube`, причём параметр, который в ней называется `x`, получит значением результат вычисления `b+3.5`; в свою очередь, результат работы функции `Cube` будет использован в роли уменьшаемого при выполнении вычитания, а результат вычитания окажется занесён в переменную `a`.

Как и процедура, функция является *подпрограммой*, то есть фактически представляет собой обособленный фрагмент кода, которому при вызове *временно передаётся управление*; в принципе, никто не запрещает ей выполнять совершенно произвольные действия. Значение, которое функция в итоге «вернёт» (то самое значение, ради вычисления которого её вызывают), задаётся в коде функции оператором присваивания специального вида, у которого в левой части вместо имени переменной записано имя самой функции. Наша функция `Cube` из одного этого оператора и состоит, но бывают случаи более сложные. Например, хорошо известна последовательность *чисел Фибоначчи*, в которой первые два элемента равны единицам, а каждый последующий равен сумме двух предыдущих: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ Функция, вычисляющая число Фибоначчи по его номеру, могла бы выглядеть, например, так (учитывая быстрый рост этих чисел, мы будем использовать для их нумерации числа типа `integer`, а для работы с самими числами Фибоначчи — числа типа `longint`):

```
function Fibonacci(n: integer): longint;
var
    i: integer;
    p, q, r: longint;
```

```
begin
    if n <= 0 then
        Fibonacci := 0
    else
    begin
        q := 0;
        r := 1;
        for i := 2 to n do
        begin
            p := q;
            q := r;
            r := p + q
        end;
        Fibonacci := r
    end
end;
```

Здесь будут уместны некоторые пояснения. Основной алгоритм, реализованный в нашей функции, действует при условии, что номер, переданный в функцию через параметр `n`, не меньше единицы. На каждом шаге в переменных `p`, `q` и `r` хранятся два предыдущих числа Фибоначчи и текущее. Перед началом работы в `q` заносится `0`, а в `r` — число `1`, что соответствует числам Фибоначчи с номерами 0 и 1. Следует обратить внимание, что *в переменной `r` теперь находится текущее число Фибоначчи, а номер текущего числа считается первым.* Расположенный в последующих строках цикл «сдвигает» переменные `p`, `q` и `r` на одну позицию вдоль последовательности, для этого в `p` заносится то, что было в `q` (предпоследнее число становится пред-предпоследним), в `q` заносится то, что было в `r` (последнее число становится предпоследним), а в `r` заносится сумма `p` и `q`, т. е. происходит вычисление очередного числа Фибоначчи. Иначе говоря, *каждая итерация цикла увеличивает номер текущего числа на единицу, а само текущее число каждый раз оказывается в переменной `r`.* Цикл выполняется столько раз, сколько нужно, чтобы номер текущего числа сравнялся со значением параметра `n`: если `n` равно единице, цикл не выполняется вообще, если двойке — цикл отрабатывает одну итерацию, и т. д. Полученное в итоге число `r` возвращается в качестве итогового значения функции.

Легко видеть, что всё это может работать лишь для значений параметра (номера числа) от единицы и более, поэтому случай `n = 0` нам пришлось рассмотреть отдельно; именно для этого предусмотрен оператор `if`. В итоге наш пример позволяет проиллюстрировать ещё один важный момент: в теле функции может быть больше одного оператора присваивания, задающего значение, которое функция вернёт, но за каждый вызов функции сработать должен один и только один такой оператор.

Никаких формальных ограничений на действия, выполняемые в теле функции, язык Паскаль не накладывает, так что функция вполне может делать что-то ещё, кроме вычисления своего значения. **Любые следы работы функции, кроме вычисленного ею значения, которые могут быть обнаружены за пределами функции, называют *побочным эффектом функции*.**

Например, в программе рисования «алмаза» из звёздочек (см. стр. 243) мы могли бы ради дальнейшего упрощения главной части программы вынести в функцию организацию диалога с пользователем. Напомним, что нам в итоге нужно число `n`, представляющее собой *полувысоту* фигуры, но от пользователя требуется ввести высоту целиком, причём так, чтобы это было положительное нечётное число. Напишем функцию, которая проведёт весь диалог, а вернёт искомую полувысоту:

```
function NegotiateHalfSize: integer;
var
    h: integer;
begin
    repeat
        write('Enter the diamond''s height (positive odd): ');
        readln(h)
    until (h > 0) and (h mod 2 = 1);
    NegotiateHalfSize := h div 2
end;
```

Теперь первые шесть строк главной части программы можно заменить одним присваиванием:

```
    n := NegotiateHalfSize;
```

Как видим, функция в итоге вычисляет значение, но при этом она также что-то выводит на экран, что-то считывает с клавиатуры; все эти операции ввода-вывода есть ни что иное, как её побочный эффект. Забегая вперёд, отметим, что побочный эффект может быть не связан с вводом-выводом.

2.4.3. Параметры-переменные

Параметры, которые мы применяли в наших подпрограммах до сих пор, иногда называют ***параметрами-значениями***. Имя такого параметра, данное ему в заголовке подпрограммы, представляет собой фактически *локальную переменную*, в которую при вызове подпрограммы заносится *значение*, указанное в вызове (отсюда название «параметр-значение»). Например, если у нас есть процедура `p` с параметром `x`:

```
procedure p(x: integer);
begin
    { ... }
end;
```

мы можем вызвать её, указав в точке вызова в качестве значения параметра произвольное выражение, лишь бы оно дало в результате целое число (любого целочисленного типа), например:

```
a := 15;
p(2*a + 7);
```

При таком вызове сначала будет вычислено значение выражения `2*a + 7`; полученное число `37` будет занесено в локальную переменную (параметр) `x` процедуры `p`, после чего будет выполнено тело процедуры. Очевидно, что *изнутри* процедуры мы никак не можем повлиять на значение выражения `2*a + 7` и тем более на значение переменной `a`, о которой мы вообще ничего не знаем, находясь внутри процедуры. Мы можем, в принципе, присваивать новые значения переменной `x` (хотя некоторые программисты считают это плохим стилем), но по окончании выполнения процедуры переменная `x` просто исчезнет вместе со всей информацией, которую мы в неё записали.

Некоторую путаницу у начинающих вызывает тот факт, что в качестве значения параметра можно (то есть никто не мешает) указать имя переменной соответствующего типа:

```
a := 15;
p(a);
```

Этот случай ничем не отличается от предыдущего: обращение к переменной `a` есть не более чем частный случай выражения, результатом вычисления этого выражения становится число `15`, оно заносится в локальную переменную `x`, после чего выполняется тело процедуры. В этом случае можно сказать, что `x` стало *копией* `a`, но со своим оригиналом эта копия никак не связана; действия над копией никак не отражаются на оригинале, то есть мы, как и в предыдущем случае, никак не можем изнутри процедуры повлиять на значение переменной `a`.

Между тем в некоторых случаях удобно иметь возможность именно что *изнутри подпрограммы изменить значение (значения) одной или нескольких переменных, находящихся в точке вызова.* Например, это может потребоваться, если наша подпрограмма вычисляет больше одного значения, и все эти значения нужны вызывающему. Для передачи из подпрограммы к вызывающему *одного* значения мы можем описать функцию, как, например, мы это сделали для возведения в куб; но что делать, если нам хочется, к примеру, написать подпрограмму, которая для заданного числа вычисляет сразу его квадрат, куб, четвёртую и

пятую степень? В этом простом примере можно указать «лобовое» решение проблемы: написать не одну функцию, а четыре; но если мы их все последовательно вызовем, в итоге будет выполнено десять умножений, тогда как для решения задачи их достаточно сделать всего четыре. В более сложных случаях «лобового» решения проблемы может не оказаться.

На помощь здесь приходят ***параметры-переменные***, которые отличаются от параметров-значений тем, что в подпрограмму передаётся в этом случае не *значение*, а *переменная как таковая*. Имя такого параметра на время выполнения подпрограммы становится *синонимом* переменной, которая была указана в качестве параметра в точке вызова, и всё, что внутри подпрограммы делается с именем параметра, на самом деле происходит с этой переменной.

В заголовке подпрограммы перед описаниями параметров-переменных ставится слово `var`, которое действует до точки с запятой или закрывающей скобки; например, для процедуры с заголовком

```
procedure p(var x, y: integer; z: integer; var k: real);
```

параметры `x`, `y` и `k` будут параметрами-переменными, а `z` — параметром-значением.

Задачу с возведением в четыре степени можно с использованием параметров-переменных решить следующим образом:

```
procedure Powers(x: real; var quad, cube, fourth, fifth: real);
begin
    quad := x * x;
    cube := quad * x;
    fourth := cube * x;
    fifth := fourth * x
end;
```

Процедура `Powers` имеет пять параметров; при её вызове в качестве первого параметра можно указать произвольное выражение типа `real`[21], а вот остальные четыре параметра требуют указания *переменной*, притом переменной типа `real` и никакого иного. Это ограничение вполне понятно: внутри процедуры `Powers` идентификаторы `quad`, `cube`, `fourth` и `fifth` будут синонимами того, что мы укажем в точке вызова соответствующими параметрами, а работа с ними внутри процедуры ведётся как с переменными типа `real`; если попытаться в таком качестве использовать переменную любого другого типа (то есть использующую

[21] Кстати, можно даже использовать целочисленное выражение; целые числа Паскаль при необходимости молча переводит в числа с плавающей точкой, а вот для перевода в обратную сторону приходится в явном виде указать, как именно нужно выполнить перевод: с округлением или с отбрасыванием дробной части. Разговор об этом у нас пока впереди.

другое машинное представление для хранения значения), на выходе мы получим полный хаос.

Правильный вызов такой подпрограммы мог бы выглядеть так:

```
var
    p, q, r, t: real;
begin
    { ... }

    Powers(17.5, p, q, r, t);
```

В результате работы такого вызова в переменные `p`, `q`, `r` и `t` будут занесены вторая, третья, четвёртая и пятая степени числа `17.5`.

Передача информации из подпрограммы «наружу» — не единственное применение для параметров-переменных. Например, позже мы будем рассматривать переменные, имеющие достаточно большие размеры; копирование такой переменной будет происходить слишком долго по времени, так что если передавать их по значению, вся программа целиком может оказаться чересчур медленной. При передаче переменной (сколь угодно большой) в подпрограмму через параметр-переменную никакого копирования не происходит, что позволяет использовать параметры-переменные для *оптимизации*.

Кроме того, Паскаль предусматривает один особый вид переменных (так называемые файловые переменные), которые нельзя даже присваивать, не говоря уже о копировании; такие переменные можно передавать в подпрограммы исключительно через параметры-переменные и никак иначе.

2.4.4. Локальные и глобальные переменные

Как мы уже отмечали, переменная, описанная в секции описаний подпрограммы, видна только в самой этой подпрограмме и больше нигде; такие переменные называются *локальными в подпрограммах*. В противоположность этому **переменные, описанные в программе вне подпрограмм, видны начиная от того места, где они описаны, и до конца текста программы**. Если ваша переменная будет использоваться только в главной части программы, лучше всего описать её непосредственно перед началом главной части[22], чтобы только там она и была видна; такие переменные можно с некоторой натяжкой назвать *локальными переменными главной части программы*.

Если переменную описать раньше, то она будет видна во всех подпрограммах, описанных после неё. Такие переменные называются ***глобальными***. Теоретически можно использовать глобальные переменные для передачи информации в подпрограмму и из неё: например, мы можем

[22] Интересно, что исходный вариант Паскаля, предложенный Виртом, такой возможности не давал, поскольку секции описаний там имели строго фиксированный порядок, и секция описаний переменных должна была стоять раньше секции описания процедур и функций. Все современные реализации Паскаля лишены этого ограничения.

перед вызовом подпрограммы присвоить значение какой-нибудь глобальной переменной, а подпрограмма это значение использует, и наоборот, подпрограмма может занести значение в глобальную переменную, а тот, кто подпрограмму вызвал, это значение из переменной извлечёт. Больше того, через глобальные переменные можно (теоретически) организовать связь между разными подпрограммами: к примеру, мы вызываем сначала одну подпрограмму, потом другую, и первая что-то заносит в глобальные переменные, а вторая этим пользуется. Так вот, делать так не следует. **Насколько это возможно, всю связь с подпрограммами нужно осуществлять через параметры: передавать информацию в подпрограммы через параметры-значения, а из подпрограмм «наружу» всю информацию передавать в виде значений, возвращаемых функциями, и при необходимости — через параметры-переменные**.

Основная причина тут кроется, как это часто бывает, в особенностях восприятия программы человеком. Если работа процедуры или функции зависит только от её параметров, в целом гораздо легче себе представить работу программы и понять, что происходит; если же в дело вмешиваются значения глобальных переменных, то о них придётся помнить, то есть каждый раз, глядя на вызов подпрограммы с какими-либо параметрами, нужно будет принимать во внимание, что её работа будет зависеть ещё и от значений в каких-то глобальных переменных; параметры в точке вызова чётко видны, чего о глобальных переменных никак не скажешь. Кроме того, к глобальным переменным доступ может оказаться возможен из самых разных мест программы, так что, например, обнаружив во время отладки, что кто-то когда-то успел «загнать» в глобальную переменную такое значение, которого там никто не ожидал, вы можете потратить очень много времени, выясняя, в каком же месте вашей программы это произошло; в особенности это оказывается актуально при создании больших программ, над которыми работают одновременно несколько программистов.

Можно назвать ещё одну причину, по которой глобальных переменных следует по возможности избегать. Всегда существует вероятность того, что объект, который в настоящее время в вашей программе один, потребуется «размножить». Например, если вы реализуете игру и в вашей реализации имеется игровое поле, то весьма и весьма вероятно, что в будущем вам может понадобиться *два* игровых поля. Если ваша программа работает с базой данных, то можно (и нужно) предположить, что рано или поздно потребуется открыть одновременно две или больше таких баз данных (например, для изменения формата представления данных). Ряд примеров можно продолжать бесконечно. Если теперь предположить, что информация, критичная для работы с вашей базой данных (или игровым полем, или любым другим объектом) хранится в глобальной переменной и все подпрограммы завязаны на использование

этой переменной, то совершить «метапереход» от одного экземпляра объекта к нескольким у вас не получится.

2.4.5. Рекурсия

Подпрограммы, как мы уже знаем, можно вызывать друг из друга, но этим дело не ограничивается: **подпрограмма при необходимости может вызвать сама себя** — как напрямую (то есть когда в теле подпрограммы в явном виде содержится вызов самой этой подпрограммы), так и косвенно, когда одна подпрограмма вызывает другую, другая, возможно, третью и т. д., и какая-то из них снова вызывает первую.

Начинающих при упоминании рекурсии обычно беспокоит вопрос, как же будет обстоять дело с локальными переменными; оказывается, никаких проблем здесь не возникнет, потому что **локальные переменные подпрограммы создаются в момент вызова этой подпрограммы и исчезают, когда подпрограмма возвращает управление**; таким образом, при рекурсивном вызове будет создан новый комплект локальных переменных, и так каждый раз. Если в процедуре описана переменная `x` и эта процедура десять раз выполнила вызов самой себя, `x` будет существовать в одиннадцати экземплярах.

Важно понимать, что рекурсия рано или поздно должна закончиться, а для этого нужно, чтобы каждый последующий рекурсивный вызов решал пусть и ту же самую задачу, но для хотя бы чуть-чуть более простого случая. Наконец, обязательно выделить так называемый ***базис рекурсии***: настолько простой (обычно тривиальный или вырожденный) случай, при котором дальнейшие рекурсивные вызовы уже не нужны. Если этого не сделать, рекурсия окажется бесконечной; но поскольку каждый рекурсивный вызов расходует память — на хранение адреса возврата, на размещение значений параметров, на локальные переменные — то при уходе программы в бесконечную рекурсию рано или поздно доступная память кончится и произойдёт аварийное завершение программы.

Мы приведём совсем простой пример рекурсии. В § 2.4.1 мы написали процедуру `PrintChars`, которая печатала заданное количество одинаковых символов; сам символ и нужное их количество передавалось через параметры (см. стр. 244). Эту процедуру можно реализовать с использованием рекурсии вместо цикла. Для этого необходимо заметить, что, во-первых, случай, когда нужное количество символов равно нулю — вырожденный, при котором делать ничего не надо; во-вторых, если случай попался не вырожденный, то напечатать n символов — это то же самое, что напечатать сначала один символ, а потом $(n-1)$ символов, при этом задача «напечатать $(n-1)$ символов» вполне годится в качестве «чуть-чуть более простого» случая той же самой задачи. Рекурсивная реализация будет выглядеть так:

```
procedure PrintChars(ch: char; count: integer);
begin
    if count > 0 then
    begin
        write(ch);
        PrintChars(count - 1)
    end
end;
```

Как видим, для случая, когда печатать нечего, наша процедура ничего и не делает, а для всех остальных случаев она печатает один символ, так что остаётся напечатать на один символ меньше; для печати оставшихся символов процедура использует сама себя. Например, если сделать вызов `PrintChars('*', 3)`, то процедура напечатает звёздочку и сделает вызов `PrintChars('*', 2)`; «новая ипостась» процедуры напечатает звёздочку и вызовет `PrintChars('*', 1)`, которая напечатает звёздочку и вызовет `PrintChars('*', 0)`; этот последний вызов ничего делать не станет и завершится, предыдущий тоже завершится, и тот, что перед ним, тоже, и, наконец, завершится наш исходный вызов. Звёздочка, как легко видеть, будет напечатана трижды.

Часто оказывается полезным использование *обратного хода рекурсии*, когда подпрограмма сначала делает рекурсивный вызов, а потом выполняет ещё какие-то действия. Пусть, например, у нас имеется задача напечатать (разделив для наглядности пробелами) цифры, составляющие десятичное представление заданного числа. Отщепить от числа младшую цифру проблем не составляет: это остаток от деления на `10`. Все остальные цифры числа можно извлечь, если повторить тот же процесс для исходного числа, разделённого на десять с отбрасыванием остатка. Базисом рекурсии может выступить случай нуля: мы в этом случае не будем ничего печатать. Если нужно обязательно напечатать ноль, получив число `0`, то этот особый случай можно рассмотреть, написав другую процедуру, которая для нулевого аргумента будет печатать ноль, а для любого другого числа — вызывать нашу процедуру. Если написать что-то вроде

```
procedure PrintDigitsOfNumber(n: integer);
begin
    if n > 0 then
    begin
        write(n mod 10, ' ');
        PrintDigitsOfNumber(n div 10)
    end
end;
```

и вызвать её, например, так: `PrintDigitsOfNumber(7583)`, то напечатано будет не совсем то, чего мы хотим: «`3 8 7 5`». Цифры-то правильные,

только порядок обратный. Оно и понятно, мы же самой первой «отщепили» младшую цифру и сразу же её напечатали, и так далее, вот они и получились напечатанными справа налево. Но проблема решается всего одним небольшим изменением:

```
procedure PrintDigitsOfNumber(n: integer);
begin
    if n > 0 then
    begin
        PrintDigitsOfNumber(n div 10);
        write(n mod 10, ' ')
    end
end;
```

Здесь мы поменяли местами рекурсивный вызов и оператор печати. Поскольку возвраты из рекурсии происходят в порядке, обратном заходу в рекурсию, теперь цифры будут напечатаны в порядке, обратном тому, в котором их «отщепляли», то есть для того же самого вызова напечатано будет «`5 7 8 3`», что нам и требовалось.

Рекурсивными бывают не только процедуры, но и функции. Например, в следующем примере функция `ReverseNumber` вычисляет число, полученное из исходного «переворотом задом наперёд» его десятичной записи, а сама рекурсия происходит во вспомогательной функции `DoReverseNumber`:

```
function DoReverseNumber(n, m: longint): longint;
begin
    if n = 0 then
        DoReverseNumber := m
    else
        DoReverseNumber :=
            DoReverseNumber(n div 10, m * 10 + n mod 10)
end;

function ReverseNumber(n: longint): longint;
begin
    ReverseNumber := DoReverseNumber(n, 0)
end;
```

Например, `ReverseNumber(752)` вернёт число `257`, а `ReverseNumber(12345)` вернёт `54321`. Как это работает, предлагаем читателю разобраться самостоятельно.

2.5. Ещё о выражениях

Прежде чем идти дальше, попытаемся завершить обсуждение арифметических выражений; оно осталось бы неполным без побитовых опера-

ций, записи целых чисел в системах счисления, отличных от десятичной, и именованных констант.

2.5.1. Побитовые операции

Побитовые операции выполняются над целыми числами одинакового типа, но при этом числа рассматриваются не как собственно числа, а как *строки отдельных битов (т. е. двоичных цифр)*, составляющих машинное представление. Например, если в побитовой операции участвует число `75` типа `integer`, то имеется в виду битовая строка `000000001001011`.

Побитовые операции можно разделить на два вида: логические операции, выполняемые над отдельными битами (причём над всеми одновременно) и сдвиги. Так, операция `not`, применённая к целому числу (в отличие от уже знакомой нам операции `not` в применении к значению типа `boolean`) даёт в результате число, все биты которого противоположны исходному. Например, если переменные `x` и `y` имеют тип `integer`, после выполнения операторов

```
x := 75;
y := not x;
```

в переменной `y` окажется число `-76`, машинное представление которого `1111111110110100` представляет собой побитовую инверсию приведённого выше представления числа `75`. Отметим, что если бы `x` и `y` имели тип `word`, т. е. беззнаковый тип той же разрядности, то результат в переменной `y` составил бы `65460`; машинное представление этого числа в виде 16-битного беззнакового такое же, как у числа `-76` в виде 16-битного знакового.

Аналогичным образом над целыми числами работают уже знакомые нам по § 2.3.4 операции `and`, `or` и `xor`. Все эти операции являются бинарными, то есть требуют двух операндов; когда мы применяем их к целым числам, соответствующие логические операции («и», «или», «исключающее или») выполняются одновременно над первыми битами операндов, над их вторыми битами и так далее; результаты (тоже отдельные биты) соединяются в целое число того же типа и, как следствие, той же разрядности, которое и становится результатом всей операции. Например, восьмибитное беззнаковое представление (то есть представление типа `byte`) для чисел `42` и `166` будет соответственно `00101010` и `10100110`; если у нас есть переменные `x`, `y`, `p`, `q` и `r`, имеющие тип `byte`, то после присваиваний

```
x := 42;
y := 166;
p := x and y;
```

```
q := x or y;
r := x xor y;
```

переменные `p`, `q` и `r` получат соответственно значения `34` (`00100010`), `174` (`10101110`) и `140` (`10001100`).

Операции побитового сдвига, как следует из названия, *сдвигают* битовое представление на определённое число позиций влево (`shl`, от слов *shift left*) или вправо (`shr`, *shift right*). Обе операции бинарные, то есть предусматривают два операнда; слева от названия операции ставится исходное целое число, справа — количество позиций, на которое нужно сдвинуть его побитовое машинное представление. При сдвиге влево на k позиций старшие k разрядов машинного представления числа пропадают, а справа (то есть в качестве младших разрядов) дописывается k нулевых битов. Сдвиг влево на k позиций эквивалентен домножению числа на 2^k. Например, результатом выражения `1 shl 5` будет число `32`, а результатом `21 shl 3` будет `168`.

При сдвиге вправо пропадают, наоборот, k младших разрядов, что касается старших битов, то здесь операция работает по-разному для знаковых чисел и для беззнаковых. При сдвиге вправо беззнакового числа слева дописываются нулевые биты; при сдвиге знакового слева дописываются нули или единицы в зависимости от исходного значения самого старшего («знакового») бита числа, так что знак числа при сдвиге сохраняется: результатом сдвига вправо положительного числа будет всегда положительное, отрицательного — отрицательное. В частности, `21 shr 3` даст значение `2` (из представления `00010101` получится `00000010`), `(-64) shr 3` даст `-8`. Сдвиг вправо числа `-1` всегда даёт `-1`, на сколько бы позиций мы его ни сдвигали.

2.5.2. Именованные константы

Исходно словом «константа» обозначается такое выражение, значение которого всегда одно и то же. Тривиальным примером константы может послужить ***литерал*** — например, просто число, написанное в явном виде. К примеру, «`37.0`» — это литерал, который представляет собой выражение типа `real`; очевидно, что значение этого выражения всегда будет одно и то же, а именно — 37.0; следовательно, это константа. Можно привести более сложный пример константы: выражение «`6*7`». Это уже не литерал, это арифметическое выражение, а литералов тут два — это числа `6` и `7`; тем не менее, значение этого выражения тоже всегда одно и то же, так что и это — пример константы.

Среди всех констант выделяют ***константы времени компиляции*** — это такие константы, значение которых компилятор определяет во время обработки нашей программы. К таким константам относятся все литералы, что вполне естественно; кроме того, во время компиляции

компилятор может вычислять арифметические выражения, не содержащие обращений к переменным и функциям. Поэтому «6*7» — это тоже константа времени компиляции, компилятор сам вычислит, что значение здесь всегда 42, и именно число 42 поместит в машинный код; ни шестёрка, ни семёрка, ни операция умножения в коде фигурировать не будут.

Кроме констант времени компиляции, встречаются также ***константы времени исполнения*** — это выражения, которые по идее всегда имеют одно и то же значение, но компилятор это значение во время компиляции вычислить по каким-то причинам не может, так что оно становится известно только во время выполнения программы. Точная граница между этими видами констант зависит от реализации компилятора; например, Free Pascal умеет во время компиляции вычислять синусы, косинусы, квадратные корни, логарифмы и экспоненты, хотя делать всё это он совершенно не обязан, и другие версии Паскаля такого не делают.

Впрочем, можно найти ограничения и для Free Pascal: например, функции обработки строк, даже если их вызвать с константными литералами в качестве параметров, во время компиляции не вычисляются. Так, следующий фрагмент приведёт к ошибке во время компиляции, несмотря на то, что функция copy точно так же встроена в компилятор Паскаля, как и упоминавшиеся выше математические функции вроде синуса и логарифма:

```
const
    Hello = 'Hello world!';
    Part = copy(Hello, 3, 7);
```

Механизм ***именованных констант*** позволяет связать с неким постоянным значением (константой времени компиляции) некое имя, т. е. *идентификатор*, и во всём тексте программы вместо значения, записанного в явном виде, использовать этот идентификатор. Делается это в ***секции описания констант***, которую можно расположить в любом месте между заголовком и началом главной части программы, но обычно секцию констант программисты располагают как можно ближе к началу файла — например, сразу после заголовка программы. Дело тут в том, что значения некоторых констант могут оказаться (и оказываются) самой часто изменяемой частью программы, и расположение констант в самом начале программы позволяет сэкономить время и интеллектуальные усилия при их редактировании.

Для примера рассмотрим программу `hello20_for` (см. стр. 234); она выдаёт «на экран» (в стандартный поток вывода) сообщение «`Hello, world!`», причём делает это 20 раз. Эту задачу можно очевидным образом обобщить: программа *выдаёт заданное сообщение заданное число раз*. Из школьного курса физики нам известно, что практически любую задачу лучше всего решать в общем виде, а конкретные

значения подставлять в самом конце, когда уже получено общее решение. Аналогичным образом можно поступать и в программировании. В самом деле, что изменится в программе, если мы захотим изменить выдаваемое сообщение? А если мы захотим выдавать сообщение не 20 раз, а 27? Ответ очевиден: изменятся только соответствующие константы-литералы. В такой короткой программе, как `hello_20`, конечно, найти эти литералы несложно; а если программа состоит хотя бы из пятисот строк? Из пяти тысяч? И ведь это далеко не предел: в наиболее крупных и сложных компьютерных программах счёт строк идёт на *десятки миллионов*.

При этом заданные в коде константы, от которых зависит выполнение программы, в достаточной степени произвольны: на это однозначно указывает то обстоятельство, что задача очевидным образом обобщается на произвольные значения. Логично при этом будет ожидать, что нам, возможно, захочется изменить значения констант, не меняя больше ничего в программе; в этом смысле константы подобны настроечным ручкам разнообразных технических устройств. Именованные константы позволяют облегчить такую «настройку»: если без их применения литералы рассыпаны по всему коду, то давая каждой константе собственное имя, мы можем собрать все «параметры настройки» в начале текста программы, при необходимости снабдив их комментариями. Например, вместо программы `hello20_for` мы можем написать следующую программу:

```
program MessageN;
const
    message = 'Hello, world!';  { what to print }
    count = 20;                 { how many times }
var
    i: integer;
begin
    for i := 1 to count do
        writeln(message);
end.
```

Как видим, секция описаний констант состоит из ключевого слова `const`, за которым следует одно или несколько ***описаний констант***; каждое такое описание состоит из имени (идентификатора) новой константы, знака равенства, выражения, задающего значение константы (это выражение само должно быть константой времени компиляции) и точки с запятой. С того момента, как компилятор обработает такое описание, в дальнейшем тексте программы введённый этим описанием идентификатор будет заменяться на связанное с ним константное значение. Само имя константы, что вполне естественно, тоже считается константой времени компиляции; как мы увидим позже (например, когда будем изучать массивы), это обстоятельство достаточно важно.

Облегчением «настройки» программы полезность именованных констант не ограничивается. Например, часто бывает так, что одно и то же константное значение встречается в нескольких разных местах программы, причём по смыслу при изменении его в одном из мест нужно также (синхронно) изменить и все остальные места, где встречается та же самая константа. Например, если мы пишем программу, управляющую камерой хранения из отдельных автоматизированных ячеек, то нам наверняка потребуется знать, сколько ячеек у нас есть. От этого будет зависеть, например, подсчёт числа свободных ячеек, всяческие элементы пользовательского интерфейса, где необходимо выбрать одну ячейку из всех имеющихся, и многое другое. Ясно, что число, означающее общее количество ячеек, будет то и дело встречаться в разных частях программы. Если теперь инженеры вдруг решат спроектировать такую же камеру хранения, но на несколько ячеек больше, нам придётся внимательно просмотреть всю нашу программу в поисках проклятого числа, которое теперь надо везде поменять. Несложно догадаться, что такие вещи представляют собой неиссякаемый источник ошибок: если, скажем, в программе одно и то же число встречается тридцать раз, то можно быть уверенным, что с первого просмотра мы «выловим» от силы двадцать таких вхождений, а остальные упустим.

Ситуация резко осложняется, если в программе есть *два разных, не зависящих друг от друга параметра*, которые волей случая оказались равны одному и тому же числу; например, у нас имеется 26 ячеек камеры хранения, а ещё у нас есть чековый принтер, в строке которого умещается 26 символов, и оба числа встречаются прямо в тексте программы. Если один из этих параметров придётся изменить, то можно не сомневаться, что мы не только упустим часть вхождений нужного параметра, но разок-другой изменим тот параметр, который менять не требовалось.

Совсем другое дело, если в явном виде количество ячеек нашей камеры хранения встречается в программе лишь один раз — в самом её начале, а далее по тексту везде используется имя константы, например, `LockerBoxCount` или что-нибудь в этом духе. Изменить значение такого параметра очень просто, поскольку само это значение в программе написано ровно в одном месте; риск изменить что-нибудь не то при этом также исчезает.

Необходимо отметить ещё одно очень важное достоинство именованных констант: в программе, созданной с их использованием, гораздо легче разобраться. Поставьте себя, например, на место человека, который где-нибудь в дебрях длинной (скажем, в несколько тысяч строк) программы натыкается на число `80`, написанное вот прямо так, цифрами в явном виде — и, разумеется, без комментариев. Что это за «80», чему оно соответствует, откуда взялось? Может быть, это возраст дедушки автора программы? Или количество этажей в небоскрёбе на нью-йоркском

Бродвее? Или максимально допустимое количество символов в строке текста, выводимой на экран? Или комнатная температура в градусах Фаренгейта?

Потратив изрядное количество времени, читатель такой программы может заметить, что 80 составляет часть сетевого адреса, так называемый *порт*, при установлении соединения с каким-то удалённым сервером; припомнив, что порт с этим номером обычно используется для веб-серверов, можно будет догадаться, что программа что-то откуда-то пытается получить по протоколу HTTP (и, кстати, ещё не факт, что догадка окажется верна). Сколько времени уйдёт на такой анализ? Минута? Десять минут? Час? Зависит, конечно, от сложности конкретной программы; но если бы вместо числа 80 в программе стоял идентификатор `DefaultHttpPortNumber`, тратить время не пришлось бы вовсе.

В большинстве случаев используемые правила оформления программного кода попросту *запрещают* появление в программе (вне секции описания констант) чисел, написанных в явном виде, за исключением чисел 0, 1 и (иногда) -1; **всем остальным числам предписывается обязательно давать имена**. В некоторых организациях программистам запрещают использовать в глубинах программного кода не только числа, но и строки, то есть все строковые литералы, нужные в программе, требуется вынести в начало и снабдить именами, а в дальнейшем тексте использовать эти имена.

Рассмотренные нами константы называются в Паскале *нетипизированными*, поскольку при их описании не указывается их тип, он выводится уже при их использовании. Кроме них, Паскаль (во всяком случае, его диалекты, родственные знаменитому Turbo Pascal, в том числе и наш Free Pascal) предусматривает также ***типизированные константы***, для которых тип указывается в явном виде при описании. В отличие от нетипизированных констант, типизированные константы не являются константами времени компиляции; более того, в режиме, в котором компилятор работает по умолчанию, *значения таких констант разрешается изменять во время выполнения*, что вообще делает сомнительным применение названия «константы». Историю возникновения этой странной сущности мы оставляем за рамками нашей книги; заинтересованный читатель легко найдёт соответствующие материалы самостоятельно. В нашем учебном курсе типизированные константы не нужны, и рассматривать их мы не будем. Так или иначе, на случай, если вам попадутся примеры программ, использующие типизированные константы, что-то вроде

```
const
    message: string = 'Hello, world!';
    count: integer = 20;
```

помните, что это **совсем не то же самое**, что константы без указания типа, и по своему поведению похоже скорее на инициализированную переменную, чем на константу.

2.5.3. Разные способы записи чисел

До сих пор мы имели дело преимущественно с целыми числами, записанными в десятичной системе счисления, а когда нам приходилось работать с дробными числами, записывали их в простейшей форме — в виде обычной десятичной дроби, в которой роль десятичной запятой играет символ точки.

Современные версии языка Паскаль позволяет записывать целые числа в шестнадцатеричной системе счисления; для этого используется символ «$» и последовательность шестнадцатеричных цифр, причём для обозначения цифр, превосходящих девять, можно использовать как заглавные, так и строчные латинские буквы. Например, `$1A7` или `$1a7` — это то же самое, что и `423`.

Free Pascal поддерживает, кроме этого, ещё литералы в двоичной и восьмеричной системах. Восьмеричные константы начинаются с символа «&», двоичные — с символа «%». Например, число 423 можно записать также и в виде `%110100111`, и в виде `&647`. Другие версии Паскаля не поддерживают такую запись чисел; не было её и в Turbo Pascal.

Что касается чисел с плавающей точкой, то они всегда записываются в десятичной системе, но и здесь есть форма записи, отличающаяся от привычной. Мы уже сталкивались с так называемой ***научной нотацией*** (см. стр. 209), когда выводили числа с плавающей точкой на печать; напомним, что при этом печаталась ***мантисса***, то есть число, удовлетворяющее условию $1 \leqslant m < 10$, затем буква «`E`» и целое число, обозначающее ***порядок*** (степень 10, на которую нужно умножить мантиссу). Аналогичным образом можно записывать числа с плавающей точкой в тексте программы. Например, `7E3` — это то же самое, что `700.0`, а `2.5E-5` — то же, что `0.000025`.

2.6. Конструирование программ

Материал этой главы может оказаться непонятным для «совсем начинающих», поскольку при работе с короткими программами проблемы, о которых здесь пойдёт речь, просто не успевают проявиться. Пока ваша программа целиком умещается на экран, никаких трудностей с восприятием её структуры не возникает, да там, если честно, и нет никакой структуры.

Ситуация кардинально меняется, когда ваша программа вырастает хотя бы до нескольких сотен строк; удержать всю её структуру в голове сначала становится проблематично, а потом и вовсе нереально. Возможности профессионала тут несколько больше, чем возможности начинающего, то есть профессиональный и опытный программист, конечно, может ориентироваться в существенно бóльших объёмах кода, но и для профессионалов предел оказывается очень и очень близок. Перед

программистами всего мира эта проблема встала в конце 1960-х годов, когда сложность повседневно создаваемых компьютерных программ достигла такого уровня, на котором с ними перестали справляться даже лучшие из лучших.

2.6.1. Концепция структурного программирования

Языки программирования, использовавшиеся в 1960-е годы, в основной своей массе позволяли без каких-либо ограничений в любой момент передавать управление в любое место программы — делать так называемые безусловные переходы. Такие переходы запутывали управляющую структуру программы до такой степени, что разобраться в получившемся тексте часто не мог сам его автор[23].

Голландский учёный и программист Эдсгер Дейкстра в 1968 году в статье *Go to statement considered harmful*[24] предложил для повышения ясности программ отказаться от практики неконтролируемых «прыжков» между разными местами кода. За два года до этого итальянцы Коррадо Бём и Джузеппе Якопини сформулировали и доказали теорему, обычно называемую сейчас *теоремой структурного программирования*; эта теорема гласит, что любой алгоритм, представленный блок-схемой, может быть преобразован к эквивалентному алгоритму (то есть такому, который на тех же входных словах выдаёт те же выходные слова) с использованием *суперпозиции* из всего трёх «элементарных конструкций»: *прямого следования*, при котором сначала выполняется одно действие, а за ним другое; *неполного ветвления*, при котором определённое действие выполняется или не выполняется в зависимости от истинности некоторого логического выражения; и *цикла с предусловием*, при котором действие *повторяется* до тех пор, пока некоторое логическое выражение остаётся истинным. На практике обычно добавляют также полное ветвление и цикл с постусловием; все эти базовые конструкции мы уже видели на рис. 2.1 и 2.2 (см. стр. 222 и 232). Для всех перечисленных базовых конструкций можно отметить одно очень важное общее свойство: каждая из них предполагает **ровно одну точку входа и ровно одну точку выхода**.

Загадочное слово «суперпозиция» в данном случае означает, что каждый прямоугольник, обозначающий (по правилам блок-схем) некое *действие*, может быть заменён более подробным (или более формальным) описанием этого действия, то есть, в свою очередь, фрагментом блок-схемы, который тоже построен в виде одной из базовых конструк-

[23] Между прочим, не следует думать, что в современных условиях это невозможно; не будет преувеличением сказать, что едва ли не каждый программист хотя бы раз безнадёжно запутывался в собственном коде. Многие новички начинают серьёзнее относиться к структуре своего кода только после такого случая.

[24] Это название может быть приблизительно переведено с английского фразой «*Оператор go to сочтён вредоносным*».

ций. Такая замена называется ***детализацией***; обратную замену корректного фрагмента блок-схемы (то есть фрагмента, имеющего одну точку входа, одну точку выхода и построенного в виде одной из базовых конструкций) на один прямоугольник («действие») обычно называют ***генерализацией***. Собственно говоря, сама возможность производить генерализацию возникает в результате соблюдения правила об одной точке входа и одной точке выхода; прямоугольник, обозначающий на блок-схемах отдельно взятое действие, тоже имеет ровно один вход и ровно один выход, что как раз и позволяет заменить любую базовую конструкцию структурного программирования одним прямоугольником, то есть генерализовать её. Это, в свою очередь, позволяет за счёт сокрытия второстепенных деталей делать любую блок-схему всё проще и проще, пока она как единое целое не окажется достаточно простой для понимания с одного взгляда.

Если генерализация обычно требуется при изучении существующей программы, то детализация, напротив, широко используется при создании новых программ. Одна из самых популярных стратегий написания программного кода, которая называется ***нисходящей пошаговой детализацией***, состоит в том, что написание программы начинают с её главной части, причём вместо части обособленных фрагментов пишут так называемые ***заглушки***; заглушка может представлять собой либо простой комментарий вида «здесь должно происходить то-то и то-то», либо вызов подпрограммы (процедуры или функции), для которой пишется только заголовок и пустое тело (для функций, как правило, добавляют оператор, задающий какое-нибудь возвращаемое значение, в большинстве случаев бессмысленное), и при необходимости добавляется комментарий о том, что будет делать эта подпрограмма, когда будет реализована. Затем полученные заглушки постепенно заменяются рабочим кодом, при этом, естественно, появляются новые заглушки.

Легко видеть, что каждая заглушка соответствует прямоугольнику, обозначающему на блок-схеме некое сложное действие, которое в процессе детализации нужно заменить более подробным фрагментом блок-схемы. Между прочим, мы уже пользовались нисходящей пошаговой детализацией при создании программы `star_slash` (см. стр. 236).

2.6.2. Исключения из правил: операторы выхода

Структурное программирование — прекрасная концепция, но в некоторых случаях оказывается, что текст программы можно сделать понятнее, если от строгих канонов чуть-чуть отойти. Поскольку конечной целью является именно понятность программы, а не каноны структурного программирования сами по себе, между соблюдением канонов и однозначным повышением понятности кода следует, естественно, выбирать второе.

Среди всех программистских приёмов, «немножко» нарушающих концепцию структурного программирования, наиболее заметное место занимают всевозможные варианты «досрочного выхода», то есть принудительный переход откуда-то из «внутренностей» программной конструкции в её конец. Интересно отметить, что в том варианте Паскаля, который изначально был предложен Виртом, никаких специальных операторов досрочного выхода не было, но присутствовал оператор безусловного перехода (тот самый пресловутый `goto`), который мы рассмотрим в следующем параграфе; как это часто бывает, практика вносит коррективы в теорию, так что практически во всех современных вариантах Паскаля имеются специальные операторы для досрочного завершения цикла и отдельной его итерации, для немедленного выхода из подпрограммы и для принудительного завершения всей программы. Их мы сейчас и рассмотрим.

Чаще всего оказывается нужен оператор досрочного выхода из подпрограммы, который называется `exit`. Например, функцию вычисления чисел Фибоначчи, приведённую на стр. 245, мы можем с использованием оператора `exit` переписать так:

```
function Fibonacci(n: integer): longint;
var
    i: integer;
    p, q, r: longint;
begin
    if n <= 0 then
    begin
        Fibonacci := 0;
        exit
    end
    q := 0;
    r := 1;
    for i := 2 to n do
    begin
        p := q;
        q := r;
        r := p + q
    end;
    Fibonacci := r
end;
```

Если в предыдущем варианте нам пришлось разделить всё тело функции на две ветки оператора `if` и на протяжении всего тела функции об этом помнить, то здесь мы сначала обрабатываем «специальный случай» (он же в данном случае — базис рекурсии), и если имеет место именно этот случай, то мы устанавливаем возвращаемое значение и немедленно завершаем выполнение. Далее мы благополучно забываем

об уже обработанном случае; такая техника позволяет не заключать остальной код (фактически реализующий всё то, ради чего написана функция) в ветку `else`.

Выгода от использования оператора `exit` становится более очевидной с ростом количества специальных случаев. Пусть, к примеру, нам нужно написать подпрограмму, решающую квадратное уравнение. Коэффициенты уравнения она получит через параметры; поскольку вычислить (и вернуть) нужно два числа, а не одно, возврат лучше осуществить через параметры-переменные. Поскольку решение квадратного уравнения возможно не для любых коэффициентов, подпрограмму имеет смысл оформить в виде функции, возвращающей логическое значение: `true` она вернёт, если всё в порядке, ну а если по тем или иным причинам решить уравнение не получается, будет возвращаться значение `false`.

Специальных случаев тут два. Во-первых, коэффициент при второй степени может оказаться равен нулю, в этом случае уравнение не является квадратным и решать его как квадратное нельзя. Во-вторых, дискриминант может оказаться отрицательным[25]. Если не применять оператор досрочного выхода, функция будет выглядеть примерно так:

```
function quadratic(a, b, c: real; var x1, x2: real) : boolean;
var
    d: real;
begin
    if a = 0 then
        quadratic := false
    else
    begin
        d := b*b - 4*a*c;
        if d < 0 then
            quadratic := false
        else
        begin
            d := sqrt(d);
            x1 := (-b - d) / (2*a);
            x2 := (-b + d) / (2*a);
            quadratic := true
        end
    end
end;
```

Самое интересное — собственно решение квадратного уравнения — у нас оказалось «закопано» на третий уровень вложенности, да и в целом управляющая структура нашей функции выглядит довольно страшно. Используя оператор `exit`, мы можем переписать её несколько иначе:

[25] Рассматривать решение в комплексном случае мы не будем; специальный случай для нас сейчас интереснее.

```
function quadratic(a, b, c: real; var x1, x2: real) : boolean;
var
    d: real;
begin
    if a = 0 then
    begin
        quadratic := false;
        exit
    end;
    d := b*b - 4*a*c;
    if d < 0 then
    begin
        quadratic := false;
        exit
    end;
    d := sqrt(d);
    x1 := (-b - d) / (2*a);
    x2 := (-b + d) / (2*a);
    quadratic := true
end;
```

На практике встречаются подпрограммы с существенно бóльшим количеством специальных случаев — их может оказаться пять, десять, сколько угодно; при попытке написать такую подпрограмму без `exit` нам попросту не хватит ширины экрана для структурных отступов. Кроме того, организовывать обработку специальных случаев с помощью конструкции из вложенных `if`'ов не вполне правильно даже на идейном уровне: общий случай, который очевидно «главнее» всех отдельно рассматриваемых специальных, оказывается обработан где-то в глубине управляющей структуры, что отвлекает от него внимание и противоречит его главенствующей роли.

Досрочно завершить можно и всю программу целиком; это делается оператором `halt`. В принципе, этот оператор можно применить в любом месте программы, в том числе в любой из подпрограмм, но делать это стоит с известной осторожностью. Например, начинающие программисты очень любят «обрабатывать» любые ошибочные ситуации, выдавая сообщение об ошибке и немедленно завершая программу; чтобы понять, почему так делать не следует, достаточно представить себе редактор текстов, который таким вот радикальным образом будет реагировать на любое неправильное нажатие клавиш.

Версия оператора `halt`, включённая в Free Pascal, имеет две формы: обычную, когда в программе просто пишется слово `halt`, и *параметрическую* — в этом случае после слова `halt` в скобках указывается целочисленное выражение, то есть пишется что-то вроде `halt(1)`. Параметр оператора `halt`, если он указан, задаёт для нашей программы ***код завершения процесса***, что позволяет сообщить операционной системе, успешно ли, по нашему мнению, отработала наша

программа. Код 0 означает успешное завершение, коды 1, 2 и т. д. операционная система рассматривает как признак ошибки. В качестве кода завершения теоретически можно использовать любое число от 0 до 255 (однобайтовое беззнаковое), но обычно большие числа в этой роли не используют — в большинстве случаев код завершения не превышает 10.

Оператор `halt` без параметров эквивалентен оператору `halt(0)`, то есть успешному завершению.

Наконец, часто бывают полезны пришедшие в Паскаль из языка Си операторы завершения цикла (`break`) и отдельной его итерации (`continue`). На тех простых задачах, которые мы рассматривали до сих пор, проиллюстрировать эти операторы практически невозможно, они там не нужны; но уже очень скоро мы столкнёмся с более сложными задачами, в которых `break` и `continue` позволят ощутимо упростить текст программы.

2.6.3. Безусловные переходы

Оператор безусловного перехода `goto` позволяет в любой момент передать управление в другую точку программы или, если угодно, продолжить выполнение программы с другого её места. Операторы, на которые будут делаться безусловные переходы, помечаются так называемыми ***метками***, в роли которых могут выступать обычные идентификаторы, а также числа-номера; последнее компилятор поддерживает ради совместимости со старыми диалектами Паскаля. Поскольку программу с метками-идентификаторами читать заведомо проще, чем программу с метками-номерами, в наше время номера в качестве меток обычно не используются.

Метки должны быть перечислены в разделе описаний с помощью ключевого слова `label`, формирующего *раздел описания меток*; сами метки перечисляются через запятую, в конце ставится точка с запятой, например:

```
label
    Quit, AllDone, SecondStage;
```

Обычно описание меток вставляют непосредственно перед словом `begin` или перед разделом описания переменных. Надо отметить, что Паскаль запрещает «перепрыгивать» из одной подпрограммы в другую, «запрыгивать» в подпрограмму из основной программы и «выпрыгивать» в основную программу из подпрограмм, так что делать метки «глобальными» нет никакого смысла. Метки, используемые внутри подпрограммы, следует описывать в разделе описаний этой подпрограммы, а метки, потребовавшиеся в главной части программы — непосредственно перед её началом.

Чтобы пометить оператор меткой, эту метку записывают перед текстом оператора, отделяя от него двоеточием, например, так:

```
    writeln('We are before the point');
point:
    writeln('And now we are at the point');
```

Переход на такую метку делается совсем просто:

```
    goto point;
```

В литературе часто можно встретить утверждение о том, что оператор `goto` якобы «нельзя использовать», потому что он запутывает программу. В большинстве случаев это действительно так, но существуют две (не одна, не три, а именно две) ситуации, в которых применение `goto` не только допустимо, но и желательно.

Первая из двух ситуаций очень простая: выход из многократно вложенных управляющих конструкций, например циклов. С выходом из *одного* цикла справятся специально предназначенные для этого операторы `break` и `continue`, но что делать, если «выпрыгнуть» нужно, скажем, из трёх циклов, вложенных друг в друга? Конечно, обойтись без `goto`, строго говоря, можно и здесь: в условия циклов вставить проверки какого-нибудь специального флажка, в са́мом внутреннем цикле этот флажок взвести и сделать `break;`, и тогда все циклы завершатся. Отметим, что в большинстве случаев флажок придётся проверять не только в условиях циклов, но и некоторые части тел этих циклов обрамлять `if`'ами, проверяющими всё тот же флажок. Было бы по меньшей мере странно утверждать, что все эти нагромождения окажутся более ясными, нежели один оператор `goto` (конечно, при условии, что имя метки выбрано удачно и соответствует ситуации, в которой на неё делается переход).

Вторую ситуацию описать несколько сложнее, поскольку мы пока ещё не рассматривали ни файлы, ни динамическую память. Тем не менее, попробуйте представить себе, что вы пишете подпрограмму, которая в начале своей работы на время забирает себе некий ресурс, который должна потом отдать обратно. Такая схема работы встречается достаточно часто; по-английски высвобождение захваченных ресурсов перед окончанием работы называется *cleanup*, что может быть приблизительно переведено словом *очистка*. Необходимость произвести очистку перед выходом из подпрограммы не представляет никаких проблем, если точка выхода у нас одна; проблемы начинаются, если где-нибудь в середине текста подпрограммы возникает потребность досрочного её завершения с помощью `exit`. Попытка обойтись без `goto` приведёт к тому, что везде непосредственно перед завершением, т. е. перед `exit` и перед концом тела подпрограммы придётся продублировать код всех операций, проводящих очистку. Дублирование кода до добра обычно не доводит: если мы теперь изменим начало подпрограммы, добавив или убрав операции по захвату ресурса, велика вероятность того, что

из получившихся *нескольких* одинаковых фрагментов, осуществляющих очистку, мы исправим только некоторые, а про остальные забудем. Поэтому в такой ситуации обычно поступают иначе: перед операциями очистки, находящимися в конце подпрограммы, ставят метку (как правило, её называют `quit` или `cleanup`), а вместо `exit` делают `goto` на эту метку.

Следует обратить внимание, что в обоих случаях переход делается «вниз» по коду, т. е. вперёд по последовательности выполнения (то есть метка стоит в тексте программы ниже оператора `goto`) и «наружу» из управляющих конструкций. Если у вас возникло желание сделать `goto` *назад* — это означает, что вы создаёте цикл, а для циклов есть специальные операторы; попробуйте использовать `while` или `repeat/until`. Если же вам захотелось «впрыгнуть» внутрь управляющей конструкции, то, следовательно, у вас что-то пошло совсем не так, как надо, и нужно срочно понять причины возникновения таких странных желаний; отметим, что Паскаль такого просто не позволит.

2.6.4. О разбиении программы на подпрограммы

Как мы уже упоминали при обсуждении подпрограмм, грамотное разбиение программы на обособленные части позволяет эффективно бороться со сложностью её восприятия; это программистский вариант принципа «разделяй и властвуй».

Начинающие программисты (обычно школьники или студенты младших курсов) часто делают серьёзную стратегическую ошибку: пренебрегая подпрограммами, пытаются реализовать всю задачу в виде одной большой главной части. Когда размер такой программы переваливает за какую-нибудь сотню строк, навигация по коду резко затрудняется; попросту говоря, при работе с такой программой (например, при необходимости что-то в ней исправить) больше времени тратится на поиск нужного фрагмента, нежели на сами исправления. То же самое происходит, если позволить существенно «распухнуть» не главной программе, а любой из подпрограмм.

В идеале каждая обособленная часть программы, будь то подпрограмма или главная программа, должна быть настолько короткой, чтобы одного беглого взгляда на неё было достаточно для понимания её общей структуры. Опыт показывает, что идеальная подпрограмма не должна превышать 25 строк, хотя этот лимит, вообще говоря, не вполне жёсткий. Если в вашей подпрограмме нужно рассмотреть много разных возможных вариантов и действовать в соответствии с таким выбором, вполне допустимо чуть превысить указанный размер: подпрограмма в 50 строк в такой ситуации криминалом не считается, хотя и не приветствуется; с другой стороны, если подпрограмма подбирается к длине в 60, а то и 70 строк, то её необходимо немедленно, не оставляя этого

на абстрактное «потом», разбить на подзадачи. Если же подпрограмма перевалила за сотню строк, вам необходимо переосмыслить своё отношение к оформлению кода, поскольку при правильном подходе такого никогда не произойдёт.

Число 25 возникло здесь не случайно. Традиционный размер экрана алфавитно-цифрового терминала составляет 25x80 или 24x80 (24 или 25 строк по 80 знакомест в строке), и считается, что подпрограмма должна целиком умещаться на такой экран, чтобы для её анализа не приходилось прибегать к скроллингу. Вполне возможно, что лично вы предпочитаете работать с редакторами текстов, использующими графический режим, и на вашем экране умещается гораздо больше, чем 25 строк; это, на самом деле, никак не меняет ситуацию, потому что, во-первых, воспринимать текст существенно длиннее 25 строк тяжело, даже если его удалось поместить на экран; во-вторых, многие программисты при работе в графическом режиме предпочитают крупные шрифты, так что на *их* экран больше 25 строк всё-таки не влезет.

К вопросу оптимального размера подпрограммы можно подойти и с другой стороны. **Каждая подпрограмма должна решать ровно одну задачу**, причём вы должны для себя сформулировать, какую конкретно задачу будет решать эта подпрограмма. Если эта задача достаточно сложна, чтобы из неё можно было выделить подзадачи, то их *нужно* выделить — это повышает ясность программы.

Отметим, что правило «одна подпрограмма — одна задача» поможет вам также и при выборе параметров для подпрограммы. Начинающие программисты часто делают характерную ошибку, снабжая подпрограмму такими параметрами, смысл которых невозможно объяснить без объяснения принципа работы вызывающей подпрограммы. Такой стиль никуда не годится. **Ответ на вопрос «что делает эта подпрограмма» должен состоять из *одной фразы***, и из этой фразы должно быть хотя бы в первом приближении понятно, какова семантика всех параметров подпрограммы.

Коль скоро речь зашла о параметрах, отметим ещё один момент. **Подпрограмму, имеющую не более пяти параметров, использовать легко; подпрограмму с шестью параметрами использовать несколько затруднительно; подпрограммы с семью и более параметрами усложняют, а не облегчают работу с кодом**. Это обусловлено особенностями человеческого мозга. Удержать в памяти последовательность из пяти объектов нам достаточно просто, последовательность из шести объектов может удержать в памяти не каждый, ну а если объектов семь или больше, то удержать их в памяти единой картинкой попросту невозможно, приходится соответствующую последовательность *зазубривать*, а потом тратить время и силы на то, чтобы вспомнить зазубренное. Таким образом, если ваша подпрограмма имеет не больше пяти параметров, то, как правило, вы легко удержите в памяти их семантику (если это не так — скорее всего, подпрограмма неудачно спроектирована), так что сделать вызов такой подпрограммы

окажется для вас легко. Если же параметров больше, то написание каждого вызова этой подпрограммы превратится в мучительное и не всегда успешное ковыряние в памяти или, что более вероятно, в тексте программы; такие вещи неизбежно отвлекают программиста от текущей задачи, заставляя вспоминать несущественные детали кода, написанного ранее.

Можно назвать ещё одно эмпирическое правило, помогающее грамотно разбить задачу на подзадачи. Каждая выделенная подзадача должна быть такой, чтобы при работе над вызывающей подпрограммой можно было не помнить детали реализации вызываемой, и наоборот, при работе над вызываемой — никак не учитывать детали реализации вызывающей. Если это правило не выполняется, чтение кода с разбиением на подпрограммы может стать более трудным, чем до такого разбиения — ведь при анализе кода придётся постоянно «прыгать» между телами двух подпрограмм. Чтобы этого избежать, текст вызываемой подпрограммы следует сделать понятным человеку, никогда не видевшему текста вызывающей подпрограммы, и наоборот. **Если две взаимодействующие подпрограммы невозможно понять друг без друга, то такое разбиение на подпрограммы бесполезно** и даже может оказаться вредным.

2.7. Символы и их коды; текстовые данные

Как мы уже знаем из § 1.6.5, тексты в компьютере представляются в виде последовательностей чисел, каждое из которых соответствует одному символу текста, причём один и тот же символ всегда представляется одним и тем же числом; это число называется ***кодом символа***.

В этой главе мы будем рассматривать несколько упрощённую ситуацию: считать, что для хранения кода одного символа достаточно одного байта. В такую картину вполне укладывается кодировка ASCII и её многочисленные восьмибитные расширения (см. § 1.6.5); более того, многие программы, написанные в расчёте на однобайтовые коды символов, будут вполне успешно работать с UTF-8. Что касается «совсем корректной» работы с многобайтовыми символами, то это тема для отдельного разговора, причём тема достаточно сложная; подробное исследование этого вопроса отвлекло бы нас от более насущных задач, поэтому мы оставим эти проблемы за рамками нашей книги.

2.7.1. Средства работы с символами в Паскале

Как мы уже знаем, для хранения символов в Паскале предусмотрен специальный тип, который называется `char`. *Значениями* выражений этого типа являются одиночные символы, такие как `'a'`, `'7'`, `'G'`, `'!'`,

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	SPC	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рис. 2.3. Шестнадцатеричные коды ASCII

’ ’ (пробел), а также *специальные символы*, такие, например, как перевод строки, табуляция и т. п. Обсуждая в первой части нашей книги компьютерное представление текстовой информации, мы уже отмечали, что конкретный набор доступных символов зависит от используемой кодировки, но можно точно сказать, что *символы таблицы ASCII присутствуют всегда и везде.* Саму таблицу ASCII (с указанием кодов символов в десятичной системе) мы приводили на рис. 1.13, стр. 179. Для удобства на рис. 2.3 мы привели ту же таблицу с шестнадцатеричными кодами; сочетаниями из двух и трёх заглавных латинских букв обозначены специальные (управляющие) символы; пробел для наглядности обозначен «SPC». Из всего многообразия этих «хитрых» символов, которые на самом деле не совсем символы, нас могут заинтересовать разве что NUL («символ» с кодом 0, при выдаче на печать на экране ничего не меняется; часто используется в качестве ограничителя строкового буфера); BEL (*bell*, код 7, при выдаче на печать на экране ничего не меняется, но по идее должен прозвучать короткий звуковой сигнал; на телетайпах звенел звонок); BS (*backspace*, код 8, при выдаче на печать перемещает курсор на одну позицию *влево*), HT (*horizontal tabulation*, табуляция, код 9; перемещает курсор вправо в ближайшую позицию, кратную восьми или другому числу, установленному в настройках терминала), LF (*linefeed*, уже знакомый нам перевод строки с кодом 10), CR (*carriage return*, возврат каретки, код 13, курсор перемещается в начало текущей строки) и ESC (*escape*, при выводе на печать обычно обозначает начало управляющей кодовой последовательности, например, для перемещения курсора в заданную позицию).

Некоторые из управляющих символов могут быть получены программой при вводе текста с клавиатуры: это LF (при нажатии клавиши Enter), HT (клавиша Tab), BS и DEL (соответственно клавиши Backspace и Delete), ESC (клавиша Escape); кроме того, «символы» с управляющими кодами при работе с терминалом (как настоящим, так и эмулируемым программно) можно ввести, нажав комбинацию клавиш с Ctrl: Ctrl-@ (0), Ctrl-A (1), Ctrl-B (2), ..., Ctrl-Z (26), Ctrl-[(27),

`Ctrl-\` (28), `Ctrl-]` (29)., `Ctrl-^` (30), `Ctrl-_` (31), но многие из этих кодов драйвер терминала обрабатывает сам, так что работающая в терминале программа их не видит. Например, если вы нажмёте `Ctrl-C`, драйвер терминала не отдаст вашей программе символ с кодом `3`; вместо этого он отправит программе специальный ***сигнал***, который её попросту убьёт (мы обсуждали этот эффект во вводной части, см. стр. 82). При необходимости драйвер терминала может быть перенастроен, и это может, в том числе, сделать ваша программа; в частности, чуть позже мы столкнёмся с программами, которые отказываются «убиваться» при нажатии `Ctrl-C`.

В программе на Паскале символы, как мы уже знаем, обозначаются с помощью апострофов: любой[26] символ, заключённый в апострофы, обозначает сам себя. Кроме этого, можно задать символ с помощью его кода (в десятичной системе): например, `#10` означает символ перевода строки, а `#55` — это абсолютно то же самое, что `'7'` (как видно из таблицы, код символа семёрки составляет 37_{16}, то есть 55_{10}). Для задания специальных символов с кодами от 1 до 26 можно также воспользоваться так называемой «кареточной» нотацией: `^A` означает символ с кодом `1` (т. е. то же самое, что и `#1`), `^B` — то же, что и `#2`, и так далее; вместо `#26` можно написать `^Z`.

К примеру, как это нам уже хорошо известно, оператор `writeln`, напечатав все свои аргументы, в конце выдаёт *символ перевода строки*, что как раз и приводит к перемещению курсора на следующую строку. При этом никто не мешает нам выдать символ перевода строки без посредства `writeln`; в частности, вместо хорошо знакомого нам

```
writeln('Hello, world!')
```

мы могли бы написать

```
write('Hello, world!', #10)
```

или

```
write('Hello, world!', ^J)
```

(в обоих этих случаях сначала выдаётся строка, а потом отдельно символ перевода строки); забегая вперёд, отметим, что можно сделать ещё хитрее, «загнав» символ перевода строки прямо в саму строку одним из следующих способов:

```
write('Hello, world!'#10)
write('Hello, world!'^J)
```

[26] Если в вашей системе используется кодировка на основе Unicode, то вы легко можете совершить ошибку, поставив между апострофами такой символ, код которого занимает больше одного байта; компилятор с этим не справится. Универсальный рецепт здесь очень прост: в тексте программы следует использовать только символы из набора ASCII, а всё остальное при необходимости размещать в отдельных файлах.

(здесь в обоих случаях `write` печатает только одну строку, но в этой строке в конце содержится символ перевода строки).

Символ апострофа используется как ограничивающий для литералов, представляющих как одиночные символы, так и строки; если же возникает необходимость указать сам символ «`'`», его *удваивают*, тем самым сообщая компилятору, что в данном случае имеется в виду символ апострофа как таковой, а не окончание литерала. Например, фразу «*That's fine!*» в программе на Паскале задают так: «`'That''s fine!'`». Если нам потребуется не строка, а одиночный символ апострофа, то выглядеть соответствующий литерал будет так: «`''''`»; первый апостроф обозначает начало литерала, два следующих — собственно символ апострофа, последний — конец литерала.

Над символами определены операции сравнения — точно так же, как и над числами; на самом деле сравниваются попросту коды символов: например, выражение `'a' < 'z'` будет истинным, а `'7' > 'Q'` — ложным. Это, а также непрерывное расположение в ASCII-таблице символов некоторых категорий, позволяет вычислением одного логического выражения определить, относится ли символ к такой категории; так, выражение `(c >= 'a') and (c <= 'z')` есть запись на Паскале вопроса «является ли символ, находящийся в переменной `c`, строчной латинской буквой»; аналогичные выражения для заглавных букв и для цифр выглядят как `(c >= 'A') and (c <= 'Z')` и `(c >= '0') and (c <= '9')`.

Во время работы программы можно по имеющемуся символу, то есть выражению типа `char`, получить численное значение его кода; для этого используется встроенная функция `ord`[27]. Например, если у нас есть переменные

```
var
    c: char;
    n: integer;
```

то будет корректным присваивание `n := ord(c)`, при этом в переменную `n` будет занесён код символа, хранившегося в переменной `c`. Обратная операция — получение символа по заданному коду — производится встроенной функцией `chr`; присваивание `c := chr(n)` занесёт в переменную `c` символ, код которого (в виде обычного числа) находится в `n` — при условии, что это число находилось в диапазоне от `0` до `255`; результат `chr` для других значений аргумента не определён. Конечно, `chr(55)` — это то же самое, что и `#55`, и `'7'`; но, в отличие от литералов, функция `chr` позволяет нам сконструировать символ, код которого *мы не знали во время написания программы* или который во время работы программы меняется. Например, табличку, похожую на ту, что показана

[27] На самом деле `ord` умеет не только это; полностью её возможности мы рассмотрим при обсуждении обобщённого понятия порядковых типов.

на рис. 2.3 (только без управляющих символов) можно напечатать с помощью следующей программы:

```
{ print_ascii.pas }
program PrintAscii;
var
    i, j: integer;
    c: char;
begin
    write('  |');                    { первая строка заголовка }
    for c := '0' to '9' do
        write(' .', c);
    for c := 'A' to 'F' do
        write(' .', c);
    writeln;
    write('  |');                    { вторая строка заголовка }
    for i := 1 to 16 do
        write('---');
    writeln;
    for i := 2 to 7 do       { сама таблица, строка за строкой }
    begin
        write(i, '.|');
        for j := 0 to 15 do  { печать отдельно взятого символа }
            write('  ', chr(i*16 + j));
        writeln
    end
end.
```

Как видим, здесь `chr` берётся от выражения `i*16 + j`: в каждой строке у нас 16 символов, `i` содержит номер строки, `j` — номер столбца, так что это выражение как раз будет равно коду нужного символа, остаётся только превратить этот код в символ, что мы и делаем с помощью `chr`. Результат работы программы выглядит так:

```
  | .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F
  |-----------------------------------------------
2.|     !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
3.|  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
4.|  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
5.|  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
6.|  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
7.|  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

2.7.2. Посимвольный ввод информации

Рассмотрим противоположную задачу, когда нам придётся получать код символа, неизвестного на момент написания программы. Мы уже знаем, что с помощью `read` можно прочитать с клавиатуры, например, целое число; но что будет, если наша программа попытается это сделать,

а пользователь введёт какую-нибудь белиберду? Обычно при вводе чисел оператор `read` проверяет корректность пользовательского ввода, и если пользователь ошибся и ввёл нечто, не соответствующее ожиданиям программы, выдаёт сообщение об ошибке и завершает программу. Выглядит это примерно так:

```
Runtime error 106 at $080480C4
  $080480C4
  $08061F37
```

Зная, что программа написана на Free Pascal, мы можем найти (например, в Интернете) информацию о том, что́ такое `106`, но и только; если же программа предназначена для пользователя, который сам программировать не умеет, то такая диагностика для него бесполезна и может разве что испортить настроение; к тому же, как уже говорилось ранее, завершать программу из-за любой ошибки — идея неудачная.

Мы можем «перехватить инициативу» и заявить компилятору, что сами будем обрабатывать ошибки. Это делается путём вставления в текст программы довольно странно выглядящей директивы `{$I-}` («`I`» от слова *input*, «`-`» означает, что мы отключаем встроенные диагностические сообщения для ошибок пользовательского ввода). После этого мы всегда сможем узнать, успешно прошла очередная операция ввода или нет; для этого используется встроенная функция `IOResult`. Если эта функция возвращает `0` — операция прошла успешно; если же не ноль, то это свидетельствует о происшедшей ошибке. В нашем случае, если пользователь введёт белиберду вместо числа, `IOResult` вернёт вышеупомянутое `106`. Например, программа, перемножающая два целых числа, могла бы (с учётом использования `IOResult`) выглядеть так:

```
program mul;
var
    x, y: longint;
begin
    {$I-}
    read(x, y);
    if IOResult = 0 then
        writeln(x * y)
    else
        writeln('I couldn''t parse your input')
end.
```

Конечно, фраза «*I couldn't parse your input*», в переводе с английского означающая «я не смогла разобрать ваш ввод», выглядит дружественней, чем пугающее «`Runtime error 106`», но проблемы в полной мере это не решает. Мы не знаем, ошибся ли пользователь при вводе первого числа или второго, какой конкретно символ стал причиной ошибки, в какой позиции ввода это произошло — собственно говоря, мы не знаем вообще ничего, кроме того, что введено было вместо числа нечто неудобоваримое. Это лишает нас возможности выдать пользователю

диагностическое сообщение, информационная ценность которого была бы хоть немного выше сакраментального «пользователь, ты не прав».

Для тривиального случая ввода двух целых чисел это не проблема, а вот при разборе более сложных текстов, особенно всевозможных файлов, содержащих текст на каком-то формальном языке, такие возможности никуда не годятся: программа просто-таки обязана подробно объяснить пользователю, в чём конкретно состоит ошибка и в каком месте она допущена, иначе работать с такой программой будет совершенно невозможно.

Единственный вариант, при котором мы сохраняем полный контроль за происходящим и можем сделать реакцию нашей программы на ошибки сколь угодно гибкой — это вообще отказаться от услуг оператора `read` по превращению циферок, которые вводит пользователь (то есть *текстового представления числа*) собственно в число и сделать это всё самостоятельно, читая пользовательский ввод *посимвольно*.

В простейшем случае можно, например, вынести чтение целого числа в подпрограмму, которую мы оформим как *функцию*, возвращающую логическое значение: `true`, если всё в порядке, и `false`, если произошла ошибка. Само число функция будет отдавать «наружу» через параметр-переменную[28]; если пользователь допустил ошибку, функция напечатает сообщение об этом и вернёт `false`, при этом вызывающий может при желании вызвать её снова.

При формировании числа из получаемых символов мы воспользуемся двумя соображениями. Во-первых, как мы видели, *коды символов-цифр в таблице ASCII идут подряд*, начиная от 48 (код нуля) и заканчивая числом 57 (код девятки); это позволяет получить *численное значение символа-цифры* путём вычитания кода нуля из кода рассматриваемого символа. Так, `ord('5') - ord('0')` равно `5` ($53 - 48$), `ord('8') - ord('0')` таким же точно образом равно `8`, и так далее.

Во-вторых, составить численное значение десятичной записи числа из отдельных значений его цифр, просматривая эти цифры слева направо, можно, действуя по достаточно простому алгоритму. Для начала нужно завести переменную, в которой будет формироваться искомое число, и занести туда ноль. Затем, прочитав очередную цифру, мы увеличиваем уже накопленное число в десять раз, а к тому, что получилось, прибавляем численное значение свежепрочитанной цифры. Например, при чтении числа `257` у нас перед началом чтения в переменной будет находиться ноль; после прочтения цифры «`2`» новое значение числа будет $0 \cdot 10 + 2 = 2$, после прочтения цифры «`5`» получится $2 \cdot 10 + 5 = 25$, после прочтения последней цифры мы получим $25 \cdot 10 + 7 = 257$, что и требовалось.

Осталось отметить, что для посимвольного чтения можно воспользоваться уже знакомым нам оператором `read`, указав параметром пере-

[28] Если понятие «*параметр-переменная*» вызывает неуверенность, самое время перечитать § 2.4.3.

менную типа `char`. Для большей универсальности пусть наша функция работает с числами типа `longint`. Итоговый текст будет выглядеть так:

```
{ char2num.pas }
function ReadLongint(var result: longint) : boolean;
var
    c: char;
    res: longint;
    pos: integer;
begin
    res := 0;
    pos := 0;
    repeat
        read(c);
        pos := pos + 1
    until (c <> ' ') and (c <> #10);
    while (c <> ' ') and (c <> #10) do
    begin
        if (c < '0') or (c > '9') then
        begin
            writeln('Unexpected ``', c, '''''' in pos: ', pos);
            readln;
            ReadLongint := false;
            exit
        end;
        res := res*10 + ord(c) - ord('0');
        read(c);
        pos := pos + 1
    end;
    result := res;
    ReadLongint := true
end;
```

Обратите внимание, что при обнаружении ошибки мы не только выдаём сообщение об этом, но и выполняем оператор `readln`; будучи вызванным без параметров, этот оператор удалит из потока ввода все символы до ближайшего перевода строки; иначе говоря, если мы обнаружили ошибку в пользовательском вводе, мы сбрасываем целиком всю строку, в которой обнаружилась ошибка. Попробуйте поэкспериментировать с этой функцией, допуская разнообразные ошибки в различных количествах (в том числе по несколько ошибок в одной строке), сначала в том виде, в котором мы её приводим, а потом — убрав оператор `readln`; скорее всего, его предназначение станет для вас очевидным.

Для демонстрации работы с этой функцией напишем программу, которая будет запрашивать у пользователя два целых числа, считывать их с помощью `ReadLongint` и выдавать их произведение. Её главная часть может выглядеть, например, так:

```
var
    x, y: longint;
    ok: boolean;
begin
    repeat
        write('Please type the first number: ');
        ok := ReadLongint(x)
    until ok;
    repeat
        write('Please type the second number: ');
        ok := ReadLongint(y)
    until ok;
    writeln(x, ' times ', y, ' is ', x * y)
end.
```

Вообще в операционных системах семейства Unix традиции организации диалога с пользователем несколько иные; считается, что программа не должна задавать пользователю вопросы и вообще не должна ничего говорить, если только всё идёт как должно идти; что-то говорить следует только в случае ошибок. Интересно, что это позволит переписать нашу главную часть программы гораздо короче:

```
var
    x, y: longint;
begin
    while not ReadLongint(x) do
        ;
    while not ReadLongint(y) do
        ;
    writeln(x, ' times ', y, ' is ', x * y)
end.
```

Эта программа примечательна тем, что **тела обоих циклов пустые**; роль ***пустого оператора*** выполняет символ точки с запятой, точнее, он завершает оператор цикла `while`, так и оставив его с пустым телом. Как можно догадаться, это стало возможным благодаря *побочному эффекту* функции `ReadLongint`; этот побочный эффект, состоящий в операциях ввода-вывода и в занесении значения в параметр-переменную, как раз и образует настоящее тело цикла.

Приём с побочными эффектами в заголовке тела цикла очень часто используется при работе на других языках программирования, в том числе на Си, который мы будем изучать в одной из следующих частей книги. Программисты, пишущие на Паскале, такой техникой пользуются реже; в любом случае, от применения побочных эффектов в условных выражениях лучше воздержаться, пока вы не почувствуете уверенности в своих действиях.

Итог этого параграфа можно выразить одной фразой: **посимвольный ввод и анализ текстовой информации — это наиболее универсальный подход к её обработке**.

2.7.3. Чтение до конца файла и программы-фильтры

Как мы уже знаем, чтение, выполняемое с помощью `read`, на самом деле далеко не всегда происходит с клавиатуры, поскольку пользователь может запустить нашу программу с перенаправлением ввода или в составе конвейера. Данные, читаемые из файла, в отличие от вводимых с клавиатуры, имеют одно интересное свойство: *они в какой-то момент заканчиваются.* Попытаемся рассмотреть этот момент подробнее.

До сих пор мы всегда точно знали, какую информацию должен ввести пользователь и сколько её будет, так что у нас не возникало проблемы, в какой момент прекратить ввод. Но дела могут обстоять иначе: нам может потребоваться читать данные, «пока они не закончатся». Вопрос тут состоит в том, как программа узнает, что данные в стандартном потоке ввода закончились. Формально в таком случае говорят, что в потоке возникла ***ситуация «конец файла»***; Паскаль позволяет проверить это с помощью встроенной функции `eof`, название которой образовано от слов *end of file*, то есть «конец файла». При работе с потоком стандартного ввода функция `eof` используется без параметров; она возвращает значение типа `boolean`: `false`, если конец файла пока не достигнут и можно продолжать чтение, и `true`, если читать уже больше нечего.

Важно отметить, что «конец файла» — это именно *ситуация*, а не что-то иное. В литературе для начинающих, качество которой оставляет желать лучшего, встречаются такие странные понятия, как «символ конца файла», «маркер конца файла», «признак конца файла» и тому подобное; **не верьте!** Существование какого-то там «символа» конца файла — не более чем миф, который своими истоками восходит к тем далёким временам, когда «файл» представлял собой запись на магнитной ленте; в те времена на ленте действительно формировалась особая последовательность намагниченных участков, обозначавшая конец записи (впрочем, даже тогда это не имело ничего общего с «символами»). Файлы, расположенные на диске, совершенно не нуждаются ни в каких «признаках конца», поскольку операционная система хранит на диске длину каждого файла (в виде обыкновенного целого числа) и точно знает, сколько данных можно из этого файла прочитать.

Итак, запомните раз и навсегда: **в конце файла нет ни символа, ни признака, ни маркера, ни чего-либо ещё, что можно было бы «прочитать» вместо очередной порции данных**, и каждый, кто вас пытается убедить в противоположном, попросту врёт. Когда мы читаем данные из файла и файл заканчивается, мы при этом **ничего не прочитываем**; вместо этого **возникает ситуация «конец файла»**, означающая, что читать из этого файла больше нечего. Именно

это — возникла ли уже ситуация «конца файла» или нет — проверяет встроенная функция `eof`.

В системах семейства Unix традиционно выделяют целый класс программ, которые читают текст из своего стандартного потока ввода и выдают в поток стандартного вывода некий новый текст, который может быть модификацией прочитанного текста или совершенно новым текстом, содержащим какие-то результаты анализа прочитанного текста. Такие программы называются ***фильтрами***. Например, программа `grep` выделяет из введённого текста строки, удовлетворяющие определённым условиям (чаще всего — просто содержащим определённую подстроку) и только эти строки выводит, а остальные игнорирует; программа `sort` формирует выдаваемый текст из строк введённого текста, отсортированных в определённом порядке; программа `cut` позволяет выделить из каждой строки некую подстроку; программа `wc` (от слов *word count*) подсчитывает во вводимом тексте количество символов, слов и строк, а выдаёт одну строку с результатами подсчёта. Все эти программы и многие другие как раз представляют собой *фильтры*.

Умея выполнять посимвольное чтение и обнаруживать ситуацию «конец файла», мы можем сами написать на Паскале программу-фильтр. Начнём с совсем простого фильтра, который читает текст из стандартного потока ввода и в ответ на каждую введённую пользователем строку выдаёт строку «`Ok`», а когда текст кончится — фразу «`Good bye`». Для создания этого фильтра нам совершенно не нужно хранить в памяти всю введённую строку, достаточно вводить символы один за другим, и когда введён символ перевода строки, выдавать «`Ok`»; делать это следует до тех пор, пока не возникнет ситуация «конец файла». После выхода из цикла чтения останется только выдать «`Good bye`». Пишем:

```
{ filter_ok.pas }
program FilterOk;
var
    c: char;
begin
    while not eof do
    begin
        read(c);
        if c = #10 then
            writeln('Ok')
    end;
    writeln('Good bye')
end.
```

Проверить корректную работу этой программы можно не только с помощью файла (это как раз не очень интересно — она выдаст «`Ok`» столько раз, сколько в файле было строк, но не считать же их, в самом деле), но и с использованием обычного ввода с клавиатуры. При

этом нужно помнить (см. § 1.4.7), что драйверы терминалов в ОС Unix умеют *имитировать конец файла*; если терминал не перенастраивать, то он делает это при нажатии комбинации клавиш `Ctrl-D`. Если мы запустим программу `FilterOk` без перенаправления ввода и начнём вводить произвольные строки, в ответ на каждую введённую строку программа скажет «`Ok`»; когда нам надоест, мы можем нажать `Ctrl-D`, и программа корректно завершится, выдав на прощанье «`Good bye`». Конечно, мы могли бы просто «убить» программу, нажав `Ctrl-C`, а не `Ctrl-D`, но тогда никакого «`Good bye`» она бы нам не выдала.

Напишем теперь более интересный фильтр, который подсчитывает длину каждой введённой строки и выдаёт результат подсчёта, когда строка заканчивается. Как и в предыдущем случае, нам совершенно не нужно при этом хранить всю строку целиком; читать текст мы будем посимвольно, а для хранения текущего значения длины строки заведём переменную `count`; при прочтении любого символа, кроме символа конца строки, мы эту переменную будем увеличивать на единицу, а при прочтении конца строки — выдавать накопленное значение и обнулять переменную для подсчёта длины следующей строки. Ещё важно не забыть обнулить нашу переменную в самом начале программы, чтобы длина самой первой строки тоже посчиталась корректно. Всё вместе будет выглядеть так:

```
{ filter_len.pas }
program FilterLength;
var
    c: char;
    n: integer;
begin
    n := 0;
    while not eof do
    begin
        read(c);
        if c = #10 then
        begin
            writeln(n);
            n := 0
        end
        else
            n := n + 1
    end
end.
```

Рассмотрим более сложный пример. Пусть нам нужна программа-фильтр, которая выбирает из вводимого текста строки, начинающиеся с непробельных символов, и только их печатает, а строки, которые начинаются с пробела или табуляции, как и пустые строки — игнорирует.

Может возникнуть ощущение, что здесь необходимо чтение строки целиком, но и на этот раз это не так. На самом деле нам достаточно помнить, печатаем ли мы текущую строку или нет; бывает и так, что мы пока не знаем, печатать ли текущую строку — так случается, если мы ещё не прочитали ни одного символа строки. Для хранения обоих условий мы воспользуемся логическими[29] переменными: одна, которую мы назовём `know`, будет помнить, знаем ли мы, печатать текущую строку или нет, а вторая — `print` — будет использоваться лишь в том случае, если `know` «истинна», и в этом случае она будет показывать, печатаем ли мы текущую строку.

Прочитав символ, мы прежде всего проверим, не является ли он символом перевода строки. Если да, то мы для начала проверим, не выполняется ли печать текущей строки, и если да, то напечатаем символ перевода строки; после этого мы занесём в переменную `know` значение «лжи», чтобы показать, что у нас начинается следующая строка, про которую мы пока не знаем, нужно ли её печатать.

Если прочитанный символ не является символом перевода строки, то у нас есть два варианта. Если мы ещё не знаем, печатается ли текущая строка, то самое время это узнать: в зависимости от того, прочитан ли пробельный символ (или табуляция) или нет, заносим в переменную `print` значение «лжи» или «истины», после чего в переменную `know` заносим «истину», ведь теперь мы уже точно знаем, печатается текущая строка или не печатается.

Дальше, каков бы прочитанный символ ни был, мы знаем, печатать его или не печатать: в самом деле, даже если мы только что этого не знали, то уже узнали. Если нужно, печатаем символ, и на этом тело цикла заканчивается.

В начале программы нужно не забыть указать, что мы пока что не знаем, будет ли печататься следующая строка; это делается занесением «лжи» в переменную `know`. На всякий случай занесём «ложь» также и в переменную `print`, иначе компилятор выдаст предупреждение о том, что эта переменная может быть использована без инициализации (здесь это не так, но для компилятора ситуация оказалась слишком сложной). Полностью текст программы будет выглядеть так:

```
{ skip_indented.pas }
program SkipIndented;
var
    c: char;
    know, print: boolean;
begin
    know := false;
```

[29] Здесь было бы правильнее использовать переменную перечислимого типа для трёх вариантов («да», «нет», «не знаю»), но мы ещё не разбирали перечислимый тип.

```
    print := false;
    while not eof do
    begin
        read(c);
        if c = #10 then
        begin
            if know and print then
                writeln;
            know := false
        end
        else
        begin
            if not know then
            begin
                print := (c <> ' ') and (c <> #9);
                know := true;
            end;
            { к этому моменту всегда know = true }
            if print then
                write(c)
        end
    end
end.
```

Интересно будет проверить эту программу, дав ей на вход *её собственный исходный текст*:

```
avst@host:~/work$ ./skip_indented < skip_indented.pas
program SkipIndented;
var
begin
end.
avst@host:~/work$
```

В самом деле, в нашей программе только эти четыре строки написаны с крайней левой позиции, а остальные сдвинуты вправо (используются структурные отступы) и начинаются, как следствие, с пробелов.

При анализе текста нашей программы может возникнуть обманчивое впечатление, что, если текст вводить не из файла, а с клавиатуры, то выдаваемые программой символы будут перемешиваться с символами, вводимыми пользователем. На самом деле это не так: терминал отдаёт нашей программе вводимый пользователем текст не по одному символу, а целиком строками, то есть к тому моменту, когда наша программа читает первый символ строки, пользователь уже ввёл всю строку целиком, так что выдаваемые символы появятся на экране в следующей строке. Вы можете проверить это сами.

Конечно, программы-фильтры бывают гораздо более сложными, хранить в памяти зачастую приходится не только целиком строки,

но и вообще весь вводимый текст (именно так вынуждена поступать, например, программа `sort`). Позже, познакомившись с управлением динамической памятью, мы и сами научимся писать такие программы.

2.7.4. Чтение чисел до конца файла; функция `SeekEof`

В предыдущем параграфе мы обсудили ситуацию «конец файла», её возникновение в потоке стандартного ввода и показали, как организовать чтение данных до тех пор, пока они не кончатся. Когда чтение выполняется посимвольно, никаких проблем с этим не возникает; начинающие часто попадаются здесь в довольно простую и стандартную ловушку, пытаясь применить тот же самый подход к чтению последовательностей чисел.

Рассмотрим простейшую задачу: пользователь с клавиатуры вводит целые числа, а нам нужно прочитать их все и посчитать их количество и общую сумму. Количество чисел заранее неизвестно, а о том, что числа кончились, пользователь извещает нас самым правильным способом — устроив нам «конец файла», то есть нажав Ctrl-D, если ввод идёт с клавиатуры; если числа пользователь уже записал в файл или их вообще генерирует другая программа, то для возникновения «конца файла» даже нажимать ничего не придётся, всё произойдёт само собой. Будем считать, что нам достаточно разрядности типа `longint`, то есть пользователь не станет вводить числа, превышающие два миллиарда даже суммарно.

Конечно, текстовое представление числа можно прочитать по одному символу, как мы это делали в § 2.7.2, но очень уж это громоздко, так что возникает естественное желание воспользоваться возможностями, уже имеющимися в операторе `read` — он ведь умеет читать текстовое представление чисел, при этом перевод последовательности цифр в машинное представление числа он выполняет для нас сам. Начинающие программисты в такой ситуации часто пишут примерно такую программу:

```
program SimpleSum;
var
    sum, count, n: longint;
begin
    sum := 0;
    count := 0;
    while not eof do     { плохая идея! }
    begin
        read(n);
        sum := sum + n;
        count := count + 1
```

```
    end;
    writeln(count, ' ', sum)
end.
```

и с удивлением обнаруживают, что она работает «как-то не так». Нажимать `Ctrl-D` приходится несколько раз, чтобы программа, наконец, успокоилась; при этом выдаваемое программой количество чисел оказывается больше, чем мы ввели.

Понять, что тут не так, может оказаться довольно сложно, но мы попробуем; для этого нам понадобится в деталях разобраться, что же, собственно говоря, делает `read`, когда мы требуем от него прочитать число, но с этим-то как раз проблем быть не должно, ведь мы уже делали это сами в § 2.7.2. Итак, первое, что делает `read` — это пропускает пробельные символы, то есть читает из потока ввода символы по одному до тех пор, пока не найдёт первую цифру числа. Найдя эту цифру, `read` начинает читать цифры, составляющие представление числа, одну за другой, пока снова не наткнётся на пробельный символ, и в процессе этого чтения накапливает значение числа подобно тому, как это делали мы, путём последовательности умножений на десять и прибавлений значения очередной цифры.

Обратим внимание на то, что ситуация «конец файла» не обязана возникнуть немедленно после прочтения последнего числа. На самом деле она практически никогда так не возникает; вспомним, что текстовые данные представляют собой последовательность строк, а в конце каждой строки находится символ перевода строки. Если данные, поданные на вход нашей программе, представляют собой корректный текст, то после самого последнего числа в этом тексте должен стоять ещё и перевод строки (это если пользователь не оставил после числа ещё и дюжину-другую незначащих пробелов, а ему этого никто не запрещает). Получается, что в тот момент, когда `read` завершает чтение последнего числа из потока ввода, *числа уже кончились, но символы в потоке — ещё нет.* Как следствие, `eof` пока что считает, что ничего не случилось, и выдаёт «ложь»; в итоге наша программа делает ещё один `read`, который при попытке прочитать очередной символ благополучно упирается в конец файла. Его поведение в такой ситуации несколько неожиданно — не выдавая никаких ошибок, он попросту делает вид, что прочитал число `0` (почему так сделано, непонятно, но реальность именно такова). Отсюда расхождение между количеством введённых чисел, подсчитанных программой, и количеством чисел, которые реально ввёл пользователь (хотя сумма выдаётся правильная, ведь прибавление лишнего нуля её не меняет).

И это ещё не конец истории. Для случая, когда ввод идёт не из настоящего файла (который действительно может кончиться), а с клавиатуры, где ситуацию конца файла приходится имитировать нажатием

`Ctrl-D`, вполне возможно продолжение чтения после наступления ситуации конца файла; в применении к нашей ситуации это значит, что `read`, «уперевшись» в ситуацию «конец файла», *использовал* её, так что `eof` этого конца файла уже не увидит. Именно поэтому приходится нажимать Ctrl-D дважды, чтобы программа завершилась. При работе с настоящими файлами такого эффекта не наблюдается, потому что, коль скоро файл кончился, связанный с ним поток ввода остаётся в ситуации конца файла «навсегда».

Так или иначе, у нас имеется проблема, и нужно средство для её решения; Free Pascal нам такое средство предоставляет — это функция `SeekEof`. В отличие от обычного `eof`, эта функция *сначала прочитывает и игнорирует все пробельные символы*, и если она в итоге «упёрлась» в конец файла, то возвращает «истину», а если нашла непробельный символ — возвращает «ложь». При этом найденный непробельный символ «возвращается» в поток ввода, так что последующий `read` начнёт работу именно с него.

Приведённая выше «неправильная» программа превращается в правильную одним-единственным исправлением — нужно `eof` в заголовке цикла заменить на `SeekEof`, и всё будет работать именно так, как мы хотим.

На всякий случай упомянем также функцию `SeekEoln`, которая возвращает «истину», когда достигнут символ конца строки. При этом, как и `SeekEof`, она читает и отбрасывает пробельные символы. Эта функция может потребоваться, например, если формат входных данных предполагает группы чисел переменного размера, сгруппированные по разным строкам.

2.8. Система типов Паскаля

2.8.1. Встроенные типы и пользовательские типы

Переменные и выражения, которые мы использовали до сих пор, относились к различным типам (`integer` и `longint`, `real`, `boolean` и некоторым другим), но у всех этих типов есть одно общее свойство: *они встроены в язык Паскаль*, то есть нам не нужно объяснять компилятору, что это за типы, компилятор о них уже знает.

Встроенными типами дело не ограничивается; Паскаль позволяет нам самим создавать новые типы выражений и переменных. В большинстве случаев новым типам даются имена, для чего используются уже знакомые нам *идентификаторы* — точно так же, как для уже знакомых нам переменных, констант и подпрограмм; в некоторых случаях используются также ***анонимные типы***, то есть, как следует из названия, типы, которым имя не присваивается, но их использование в Паска-

ле несколько ограничено. Любые типы, вводимые программистом (то есть не являющиеся встроенными), называются ***пользовательскими типами***, потому что их вводит *пользователь компилятора*; ясно, что таким пользователем является программист, которого, конечно же, не следует путать с пользователем полученной программы.

Для описания пользовательских типов и их имён используются ***секции описания типов***, начинающиеся с ключевого слова `type`, подобно тому, как секции описания переменных начинаются со слова `var`. Как и в случае с переменными, секцию описания типов можно расположить в *глобальной области* либо *локально в подпрограмме* (между заголовком подпрограммы и её телом). Как и переменные, типы, описанные в подпрограмме, видны только в этой подпрограмме, тогда как типы, описанные вне подпрограмм (то есть в глобальной области) видны в программе с того места, где тип описан, и до конца программы.

Самый простой вариант нового типа — это синоним какого-нибудь типа, который у нас уже есть, в том числе встроенного; например, мы можем описать тип `MyNumber` как синоним типа `real`:

```
type
    MyNumber = real;
```

после чего у нас появится возможность описывать переменные типа `MyNumber`:

```
var
    x, y, z: MyNumber;
```

Делать с этими переменными можно всё то же самое, что и с переменными типа `real`; более того, их можно неограниченно смешивать в выражениях с переменными и другими значениями типа `real`, присваивать друг другу и т. п. На первый взгляд введение такого синонима может показаться бессмысленным, но иногда оно оказывается полезно. Например, мы можем во время написания какой-нибудь сравнительно сложной программы усомниться в том, какой разрядности целых чисел нам хватит для возникшей локальной задачи; если, к примеру, мы решим, что нам хватит обычных двухбайтных `integer`'ов, а потом (уже в процессе тестирования или даже эксплуатации программы) окажется, что их разрядности не хватает и программа работает с ошибками из-за переполнений, то для замены `integer`'ов на `longint`'ы или даже на `int64` нам придётся внимательно просмотреть текст программы, чтобы идентифицировать переменные, которые должны работать с числами большей разрядности, и поменять их типы; при этом мы рискуем что-нибудь упустить или, наоборот, превратить в четырёхбайтную какую-нибудь переменную, для которой достаточно и двух байт.

Некоторые программисты справляются с этой проблемой, как говорится, дёшево и сердито: попросту используют `longint` всегда и для

всего. Но при ближайшем рассмотрении такой подход оказывается не слишком удачным: во-первых, разрядности `longint` тоже может не хватить и потребуется `int64`, а во-вторых, иногда бездумное использование переменных разрядности бóльшей, чем требуется, приводит к заметному перерасходу памяти (например, при работе с большими массивами) и замедлению работы программы (если вместо `integer` использовать `int64` в 32-битной системе, скорость работы может упасть в несколько раз).

Введение типов-синонимов позволяет обойтись с этой проблемой более изящно. Допустим, мы пишем симулятор дорожного движения, в котором у каждого объекта, участвующего в нашем моделировании, имеется свой уникальный номер, а ещё бывают нужны циклы, пробегающие все такие номера; при этом мы точно не знаем, хватит нам 32767 объектов для достижения требуемых целей моделирования или не хватит. Вместо того, чтобы гадать, какой использовать тип — `integer` или `longint`[30], мы можем ввести свой собственный тип, точнее, имя типа:

```
type
    SimObjectId = integer;
```

Теперь везде, где нам по смыслу требуется хранить и обрабатывать идентификатор объекта симуляции, мы будем использовать имя типа `SimObjectId` (а не `integer`), а если вдруг мы поймём, что количество объектов в модели опасно приближается к 32000, мы сможем *в одном месте программы* — в описании типа `SimObjectId` — заменить `integer` на `longint`, остальное произойдёт автоматически. Кстати, если на протяжении программы нам ни разу не потребуется отрицательное значение идентификатора объекта, мы сможем заменить `integer` на более подходящий здесь беззнаковый `word`, и так далее.

2.8.2. Диапазоны и перечислимые типы

Рассмотренные в предыдущем параграфе имена-синонимы, вообще говоря, *никаких новых типов не создавали*, только вводили новые имена для уже существующих. Этот параграф мы посвятим двум простейшим случаям создания *действительно новых* типов.

Пожалуй, самым простым случаем, требующим наименьшего количества пояснений, является *диапазон целых чисел*. Переменная такого типа может принимать целочисленные значения, но не всякие и даже не те, которые обусловлены её разрядностью, а только значения из *заданного диапазона*. Например, мы знаем, что десятичные цифры могут иметь своим значением число от нуля до девяти; этот факт мы можем отразить, описав соответствующий тип:

[30] Тип `int64` тут точно не понадобится, это следует из специфики задачи.

```
type
    digit10 = 0..9;
var
    d: digit10;
```

Переменная d, описанная таким образом, может принимать всего десять значений: целые числа от нуля до девяти. В остальном с этой переменной можно работать точно так же, как с другими целочисленными переменными.

Уместно будет отметить, что (во всяком случае, для Free Pascal) машинное представление числа в такой переменной совпадает с машинным представлением обычного целого числа, так что размер переменной диапазонного типа оказывается таким же, как размер наименьшей переменной встроенного целого типа, способного принимать все значения заданного диапазона. Так, переменная типа digit10 будет занимать один байт; переменная типа диапазон 15000..15010 будет занимать, как ни странно, *два* байта, потому что наименьший тип, умеющий принимать все значения из этого дипапазона — это тип integer. Точно так же два байта будет занимать и переменная типа диапазон 60000..60010, поскольку все эти значения может принимать переменная типа word, и так далее. На первый взгляд это кажется несколько странным, ведь оба эти диапазона предполагают всего по 11 различных значений каждый; ясно, что для них хватило бы одного байта. Но дело в том, что, используя в таких ситуациях один байт, компилятор был бы вынужден представлять числа из диапазонов не так, как обычные целые числа, и при каждой арифметической операции над числами из диапазонов был бы вынужден вставлять в машинный код дополнительное сложение или вычитание для приведения машинного представления к виду, обрабатываемому центральным процессором. Ничего невозможного в этом нет, но в современных условиях быстродействие программы практически всегда важнее, чем объём используемой памяти.

В принципе, диапазонные типы не ограничиваются целыми числами; например, мы можем задать подмножество символов:

```
type
    LatinCaps = 'A'..'Z';
```

К этому вопросу мы вернёмся при изучении понятия *порядкового типа*. Пока отметим ещё один очень важный момент: при задании диапазона для указания его границ можно использовать только *константы времени компиляции* (см. § 2.5.2).

Ещё один простой случай пользовательского типа — так называемый ***перечислимый тип***; выражение такого типа может принимать *одно из значений, перечисленных явным образом* при описании типа. Сами эти значения задаются идентификаторами. Например, для описания цветов радуги мы могли бы задать тип

```
type
    RainbowColors =
```

```
        (red, orange, yellow, green, blue, indigo, violet);
var
    rc: RainbowColors;
```

Переменная `rc` может принимать одно из значений, перечисленных в описании типа; например, можно сделать так:

```
    rc := green;
```

Кроме присваивания, значения выражений перечислимого типа можно сравнивать (как на равенство и неравенство, так и на порядок, то есть с помощью операций <, >, <= и >=). Кроме того, для значения перечислимого типа можно узнать *предыдущее* значение и *следующее* значение. Это делается встроенными функциями `pred` и `succ`; например, при использовании типа `RainbowColors` из нашего примера выражение `pred(yellow)` имеет значение `orange`, а выражение `succ(blue)` имеет значение `indigo`. Попытки вычислить значение, предшествующее первому или следующее за последним, ни к чему хорошему не приведут, так что прежде, чем применять функции `succ` и `pred`, следует удостовериться, что выражение, используемое в качестве аргумента, не имеет своим значением, соответственно, последнее или первое значение для данного типа.

Отметим, что функции `succ` и `pred` определены отнюдь не только для перечислимых типов, но об этом позже.

Значения перечислимого типа имеют *номера*, которые можно узнать с помощью уже знакомой нам[31] встроенной функции `ord` и которые всегда начинаются с нуля, а номер следующего значения на единицу превосходит номер предыдущего.

В классическом Паскале константы, задающие перечислимый тип, равны только сами себе и ничему иному; предложение «явно задать значение константы в перечислении» с точки зрения классического Паскаля попросту бессмысленно. В то же время современные диалекты Паскаля, включая Free Pascal, под влиянием языка Си (в котором аналогичные константы являются просто целочисленными) несколько видоизменились и позволяют явно задавать номера для констант в перечислениях, порождая странный (если не сказать бредовый) тип, который считается порядковым, но при этом не допускает использования `succ` и `pred` и фактически не даёт ничего полезного, ведь (в отличие, опять же, от Си) в Паскале есть средства, специально предназначенные для описания констант времени компиляции (см. стр. 256), притом не только целочисленных. Рассматривать такие «явные значения номеров» мы не будем, как, впрочем, и многие другие средства, имеющиеся во Free Pascal.

Отметим, что константа, задающая значение перечислимого типа, может быть использована только в одном таком типе, иначе компилятор не смог бы определить тип выражения, состоящего из одной этой

[31] Напомним, что мы встречались с этой функцией, когда работали с кодами символов; см. 2.7.1.

константы. Так, описав в программе тип `RainbowColors`, а потом забыв про него и описав (например, для моделирования сигналов светофора) что-то вроде

```
type
    Signals = (red, yellow, green);
```

мы получим ошибку. Поэтому программисты часто снабжают константы перечислимых типов префиксами, мнемонически связанными с их типом. Например, мы могли бы избежать конфликта, сделав так:

```
type
    RainbowColors = (
        RcRed, RcOrange, RcYellow, RcGreen,
        RcBlue, RcIndigo, RcViolet
    );
    Signals = (SigRed, SigYellow, SigGreen);
```

2.8.3. Общее понятие порядкового типа

Под ***порядковым типом*** в Паскале понимается тип, удовлетворяющий следующим условиям:

1. всем возможным значениям этого типа сопоставлены целочисленные *номера*, причём это сделано неким естественным образом;

2. над значениями этого типа определены операции сравнения, причём элемент с *ме́ньшим* номером считается меньшим;

3. для типа возможно указать наименьшее и наибольшее значение; для всех значений, кроме набольшего, определено *следующее* значение, а для всех значений, кроме наименьшего, определено *предыдущее* значение; порядковый номер предыдущего значения на единицу меньше, а следующего значения — на единицу больше номера исходного значения.

Для вычисления порядкового номера элемента в Паскале предусмотрена встроенная функция `ord`, а для вычисления предыдущего и следующего элементов — функции `pred` и `succ` соответственно. Все эти функции нам уже знакомы, но теперь мы можем перейти от частных случаев к общему и объяснить, что́ эти функции на самом деле делают и какова их область применения.

Порядковыми типами являются:

- тип `boolean`; его наименьшее значение `false` имеет номер `0`, а наибольшее значение `true` — номер `1`;
- тип `char`; номера его элементов соответствуют кодам символов, наименьший элемент — `#0`, наибольший — `#255`;

- целочисленные типы, как знаковые, так и беззнаковые[32];
- любой диапазонный тип;
- любой перечислимый тип.

В то же время тип `real` порядковым не является; это вполне понятно, ведь числа с плавающей точкой (фактически это двоичные дроби) не допускают никакой «естественной» нумерации. Функции `ord`, `pred` и `succ` к значениям типа `real` применять нельзя, это вызовет ошибку при компиляции.

Если говорить в общем, порядковыми в Паскале не являются никакие типы, кроме перечисленных выше, то есть кроме `boolean`, `char`, целых (разрядностью до 32 бит включительно), перечислимых типов и диапазонов.

Понятие порядкового типа в Паскале очень важно, поскольку во многих ситуациях допускается использование любого порядкового типа, но не допускается использование никакого другого. Одну такую ситуацию мы уже видели: диапазон можно задать как поддиапазон *любого порядкового типа* — и никакого другого. В будущем мы столкнёмся также с другими ситуациями, в которых необходимо использовать порядковые типы.

2.8.4. Массивы

Язык Паскаль позволяет создавать *сложные* переменные, которые отличаются от простых тем, что сами *состоят из переменных* (разумеется, другого, «более простого» типа). Сложные переменные бывают двух основных видов: *массивы* и *записи*; мы начнём с массивов.

Массивом в Паскале называется **сложная переменная, состоящая из нескольких переменных одного и того же типа**, называемых ***элементами массива***. Для обращения к элементам массива служат так называемые *индексы* — значения того или иного *порядкового типа*, чаще всего — обыкновенные целые числа, но не обязательно; индекс записывается в квадратных скобках после имени массива, то есть если, например, `a` — это переменная типа «массив», для которого индексами служат целые числа, то `a[3]` — это элемент массива `a`, имеющий номер (индекс) `3`. В квадратных скобках можно указывать не только константу, но и произвольное выражение соответствующего типа, что позволяет вычислять индексы во время исполнения программы (заметим, без этой возможности массивы не имели бы никакого смысла).

Коль скоро массив — это, по сути, переменная, пусть даже и «сложная», у этой переменной должен быть тип; его можно, как и другие типы,

[32]Компилятор Free Pascal не считает порядковыми 64-битные целые типы, то есть типы `int64` и `qword`. Это ограничение введено произволом создателей компилятора и не имеет других оснований, кроме некоторого упрощения реализации.

описать и снабдить именем. Например, если мы планируем использовать в программе массивы из ста чисел типа `real`, соответствующий тип можно описать так:

```
type
    real100 = array [1..100] of real;
```

Теперь `real100` — это *имя типа*; переменная такого типа будет состоять из ста переменных типа `real` (элементов массива), которые снабжены своими номерами (индексами), причём в качестве таких индексов используются целые числа от `1` до `100`. Введя имя типа, мы можем описывать переменные этого типа, например:

```
var
    a, b: real100;
```

Описанные таким образом `a` и `b` — это и есть массивы; они состоят из элементов `a[1]`, `a[2]`, ..., `a[100]`, `b[1]`, `b[2]`, ..., `b[100]` Например, если нам зачем-то понадобится изучать поведение синуса в окрестностях нуля, то для этого мы можем для начала в элементах массива `a` сформировать последовательность чисел $1, \frac{1}{2}, \frac{1}{4}, \ldots, 2^{-99}$:

```
a[1] := 1;
for i := 2 to 100 do
    a[i] := a[i-1] / 2;
```

а затем в элементы массива `b` занести соответствующие значения синуса:

```
for i := 1 to 100 do
    b[i] := sin(a[i]);
```

и, наконец, распечатать всю полученную таблицу:

```
for i := 1 to 100 do
    writeln(a[i], ' ', b[i]);
```

Обратите внимание, что здесь мы все операции производим над *элементами* массивов, а не над самими массивами; но массив при необходимости можно рассматривать и как единое целое, ведь это же переменная. В частности, массивы можно друг другу присваивать:

```
a := b;
```

но сначала следует хорошо подумать, ведь при таком присваивании вся информация из области памяти, занятой одним массивом, *копируется* в область памяти, занятую другим массивом, и это может занять сравнительно много времени, особенно если делать это часто — скажем, в каком-нибудь цикле. Точно так же следует соблюдать осмотрительность

при передаче массивов в качестве параметра в процедуры и функции; к этому вопросу мы ещё вернёмся. Важно помнить, что **присваивать друг другу можно только массивы одного типа**; если два массива относятся к разным типам, пусть даже совершенно одинаково описанным, присваивать их будет нельзя.

Как мы уже упоминали, в принципе типу-массиву можно не давать имени, а сразу описывать переменную такого типа; например, мы могли бы поступить так:

```
var
    a, b: array [1..100] of real;
```

и в условиях простой задачи, в которой больше массивов такого типа не предполагается, мы совершенно не заметим разницы. Но если в какой-то момент, например, локально в какой-нибудь процедуре нам потребуется ещё один такой массив, и мы опишем его:

```
var
    c: array [1..100] of real;
```

то этот массив окажется *несовместим* с массивами `a` и `b`, то есть их нельзя будет присваивать друг другу, несмотря на то, что описаны они совершенно одинаково. Дело в том, что *формально* они относятся к разным типам: каждый раз, когда компилятор видит такое описание переменной, на самом деле создаётся ещё один тип, просто ему не даётся имени, поэтому в нашем примере первый такой тип оказался создан при описании массивов `a` и `b`, а второй (пусть точно такой же, но новый) — при описании массива `c`.

Отдельного внимания заслуживает указание пределов изменения индекса. Синтаксическая конструкция «`1..100`» определённо что-то напоминает: точно таким же образом описывались диапазонные типы, и это не случайно. Для индексирования массивов в Паскале можно использовать любые порядковые типы, а при описании типа-массива на самом деле **задаётся тип, которому должно принадлежать значение индекса**, чаще всего это как раз диапазон, но не обязательно. Так, тип `real100` мы могли бы описать более развёрнуто:

```
type
    from1to100 = 1..100;
    real100 = array [from1to100] of real;
```

Здесь мы сначала описываем в явном виде диапазонный тип, снабжая его именем «`from1to100`», а потом используем это имя при описании массива. Бывают массивы, индексами которых служат не диапазоны, а что-то иное. Например, если у нас в задаче имеются шарики семи цветов радуги, причём цвет мы обрабатываем с помощью перечислимого типа

`RainbowColors` (см. стр. 290) и нужно в какой-то момент посчитать, сколько есть шариков каждого цвета, то может оказаться удобным массив такого типа:

```
type
    RainbowCounters = array [RainbowColors] of integer;
```

Можно в качестве индекса массива использовать и `char`, и даже `boolean`:

```
type
    CharCounters = array [char] of integer;
    PosNegAmount = array [boolean] of real;
```

Первый из этих типов задаёт массив из 256 элементов (типа `integer`), соответствующих всем возможным значениям типа `char` (то есть всем возможным символам); такой массив может понадобиться, например, при частотном анализе текста. Второй тип предполагает два элемента типа `real`, соответствующих логическим значениям; чтобы понять, где такое странное сооружение может пригодиться, представьте себе геометрическую задачу, связанную с вычислением сумм площадей каких-нибудь фигур, причём суммирование нужно проводить раздельно по фигурам, соответствующим какому-нибудь условию, и по всем остальным.

Массивы оказываются незаменимы в таких задачах, где прямо из условия становится очевидно, что необходимо поддерживать много значений одного и того же типа, различающихся по номерам. Рассмотрим для примера следующую задачу.

> *В городе N проводили олимпиаду по информатике. Участников ожидалось достаточно много, поэтому было решено, что регистрироваться для участия в олимпиаде они будут прямо в своих школах. Поскольку школ в городе всего 67, а от каждой школы в олимпиаде участие могут принять не больше двух-трёх десятков учеников, организаторы решили устроить нумерацию карточек участников следующим образом: номер состоит из трёх или четырёх цифр, причём две младшие цифры задают номер участника среди учеников одной и той же школы, а старшие одна или две цифры — номер школы; например, в школе №5 будущим участникам олимпиады выдали карточки с номерами* 501, 502, ..., *а в школе №49 — с номерами* 4901, 4902 *и т. д.*
>
> *Школьники, приехавшие на олимпиаду, предъявляли свои карточки организаторам, а организаторы заносили в текстовый файл* `olymp.txt` *строки следующего вида: сначала*

набирали номер карточки, а потом, через пробел, — фамилию и имя участника. Естественно, участники появлялись на олимпиаде в совершенно произвольном порядке, так что файл мог содержать, например, такой фрагмент:

```
5803 Иванов Василий
401 Лаврухина Ольга
419 Горелик Оксана
2102 Борисов Андрей
```

В день олимпиады в городе проходил футбольный матч между популярными командами, в результате чего не все ученики, зарегистрировавшиеся в своих школах, в итоге приехали на олимпиаду — некоторые предпочли пойти на стадион. Надо узнать, из каких школ прибыло наибольшее количество участников.

На первый взгляд с решением этой задачи могут возникнуть сложности, ведь мы пока не разбирали, как в Паскале работать с файлами и как обрабатывать строки, но это, как ни странно, не нужно. Имена учеников мы можем проигнорировать, ведь нам нужен только номер школы, который извлекается из номера карточки участника путём целочисленного деления этого номера на 100. Что касается файлов, то недостаток знаний мы можем компенсировать умением перенаправлять поток стандартного ввода: в программе мы будем читать информацию обычными средствами — оператором `readln` (прочитав число, этот оператор сбросит всю информацию до конца строки, что нам и требуется), а запускать программу будем с перенаправлением стандартного ввода из файла `olymp.txt`. Чтение мы будем выполнять до тех пор, пока файл не закончится; как это делается, мы уже знаем из § 2.7.3.

В процессе чтения нам придётся подсчитывать количество учеников для *каждой* из 67 школ города, то есть понадобится одновременно поддерживать 67 переменных; массивы придуманы как раз для таких ситуаций. На всякий случай число `67` вынесем в начале программы в именованную константу. Точно так же в именованную константу вынесем максимально допустимое количество учеников из одной школы; оно соответствует числу, на которое нужно делить номер карточки для получения номера школы (в нашем случае это 100):

```
program OlympiadCounter;
const
    MaxSchool = 67;
    MaxGroup = 100;
```

В принципе, массив в нашей программе нужен ровно один, так что можно было бы оставить его тип анонимным, но мы так делать не будем и опишем тип нашего массива, снабдив его именем:

```
type
    CountersArray = array [1..MaxSchool] of integer;
```

В качестве переменных нам потребуется, во-первых, сам массив; во-вторых, нам нужны будут целочисленные переменные для выполнения циклов по этому массиву, для чтения номеров карточек из потока ввода и для хранения номера школы. Опишем эти переменные:

```
var
    Counters: CountersArray;
    i, c, n: integer;
```

Теперь мы можем написать главную часть программы, которую начнём с того, что обнулим все элементы массива; в самом деле, пока мы ни одной карточки участника не видели, так что количество участников из каждой школы должно быть равно нулю:

```
begin
    for i := 1 to MaxSchool do
        Counters[i] := 0;
```

Теперь нам нужно организовать цикл чтения информации. Как мы уже договорились, выполнять чтение будем до тех пор, пока функция `eof` не скажет нам, что читать больше нечего; для чтения мы воспользуемся оператором `readln`. На всякий случай мы не будем предполагать, что данные в стандартном потоке ввода корректны, ведь файл набирали люди, а людям свойственно ошибаться. Поэтому после каждого введённого числа мы будем проверять, удалось ли нашему оператору `readln` корректно преобразовать введённую цепочку символов в число. Напомним, что это делается с помощью функции `IOResult` (см. стр. 276). Чтобы это работало, нужно не забыть сообщить компилятору о нашем намерении обрабатывать ошибки ввода-вывода самостоятельно; это, как мы знаем, делается директивой `{$I-}`.

Наконец, последний важный момент состоит в том, что, прежде чем обращаться к элементу массива, пытаясь его увеличить, нужно обязательно проверить, что полученный номер школы является допустимым. Например, если оператор случайно ошибся и ввёл какой-нибудь недопустимый номер карточки вроде `20505`, то при попытке обратиться к элементу массива с номером `205` наша программа завершится аварийно; допускать этого не следует, гораздо правильнее сообщить пользователю, что мы обнаружили в файле недопустимый номер школы.

Если обнаружена любая ошибка, обработку файла на этом стоит прекратить, всё равно нам уже не получить правильных результатов; программу можно завершить, сообщив при этом операционной системе, что мы завершаемся неуспешно (см. стр. 266). Полностью наш цикл чтения будет выглядеть так:

```
{$I-}
while not eof do
begin
    readln(c);
    if IOResult <> 0 then
    begin
        writeln('Incorrect data');
        halt(1)
    end;
    n := c div MaxGroup;
    if (n < 1) or (n > MaxSchool) then
    begin
        writeln('Illegal school id: ', n, ' [', c, ']');
        halt(1)
    end;
    Counters[n] := Counters[n] + 1;
end;
```

Следующий этап обработки полученной информации — определить, какое количество учеников из одной и той же школы является «рекордным», то есть попросту определить максимальное значение среди элементов массива `Counters`. Это делается следующим образом. Для начала мы объявим «рекордной» школу № 1, сколько бы учеников оттуда ни прибыло (даже если ни одного). Для этого мы её номер, то есть число `1`, занесём в переменную `n`; эта переменная у нас будет хранить номер школы, которая к настоящему моменту (пока что) считается «рекордной». Затем мы просмотрим информацию по всем остальным школам, и каждый раз, когда число учеников из очередной школы превышает число учеников из той, которая до сей поры считалась «рекордной», мы будем присваивать переменной `n` номер новой «рекордной» школы:

```
n := 1;                          { отдаём рекорд первой школе }
for i := 2 to MaxSchool do         { просматриваем остальные }
    if Counters[i] > Counters[n] then      { новый рекорд? }
        n := i;        { обновляем номер "рекордной" школы }
```

К моменту окончания этого цикла все счётчики будут просмотрены, так что в переменной `n` окажется номер одной из тех школ, откуда приехало максимальное количество учеников; в общем случае таких школ может быть больше одной (например, на олимпиаду могло приехать по 17 участников из школ № 3, № 29 и № 51, а из всех остальных школ — меньше). Остаётся сделать, собственно, то, ради чего написана программа: напечатать номера всех школ, из которых приехало ровно столько учеников, сколько и из той, номер которой находится в переменной `n`. Это сделать совсем просто: просматриваем все школы по порядку, и если из данной школы приехало столько же учеников, сколько и из `n`'ной, то печатаем её номер:

```
    for i := 1 to MaxSchool do
        if Counters[i] = Counters[n] then
            writeln(i)
```

Остаётся только завершить программу словом end и точкой. Целиком текст программы получается такой:

```
{ olympcount.pas }
program OlympiadCounter;
const
    MaxSchool = 67;
    MaxGroup = 100;
type
    CountersArray = array [1..MaxSchool] of integer;
var
    Counters: CountersArray;
    i, c, n: integer;
begin
    for i := 1 to MaxSchool do
        Counters[i] := 0;
    {$I-}
    while not eof do
    begin
        readln(c);
        if IOResult <> 0 then
        begin
            writeln('Incorrect data');
            halt(1)
        end;
        n := c div MaxGroup;
        if (n < 1) or (n > MaxSchool) then
        begin
            writeln('Illegal school id: ', n, ' [', c, ']');
            halt(1)
        end;
        Counters[n] := Counters[n] + 1;
    end;
    n := 1;
    for i := 1 to MaxSchool do
        if Counters[i] > Counters[n] then
            n := i;
    for i := 1 to MaxSchool do
        if Counters[i] = Counters[n] then
            writeln(i)
end.
```

Отметим, что переменные-массивы при их описании можно *инициализировать*, то есть задавать начальные значения их элементов. Эти значения перечисляют в круглых скобках через запятую, например:

```
type
    arr5 = array[1..5] of integer;
var
    a: arr5 = (25, 36, 49, 64, 81);
```

2.8.5. Тип запись

Как мы уже упоминали, Паскаль поддерживает два вида *сложных типов* — массивы и записи. Если переменная типа массив, как мы видели, состоит из переменных (элементов) одинакового типа, различающихся номером (индексом), то **переменная типа запись состоит из переменных, называемых *полями*, типы которых в общем случае различны**. В отличие от элементов массива, поля записи различаются не номерами, а *именами*.

К примеру, на соревнованиях по спортивному ориентированию каждый контрольный пункт, который необходимо пройти участникам, может характеризоваться, во-первых, его номером; во-вторых, его координатами, которые можно записывать в виде дробных чисел в градусах широты и долготы (по умолчанию можно считать, например, что широты у нас северные, долготы восточные, а южные широты и западные долготы обозначать отрицательными значениями); в-третьих, некоторые контрольные пункты могут быть «скрытыми», то есть не отражёнными на картах, выдаваемых участникам соревнований; обычно местоположение таких пунктов оказывается известно при взятии других пунктов; наконец, каждому пункту сопоставляется размер штрафа за его невзятие — обычно это целое число, выраженное в минутах. Для представления информации о контрольном пункте можно создать специальный тип:

```
type
    CheckPoint = record
        n: integer;
        latitude, longitude: real;
        hidden: boolean;
        penalty: integer;
    end;
```

Здесь идентификатор `CheckPoint` — это имя нового типа, `record` — ключевое слово, обозначающее запись; далее идёт описание полей записи в таком же формате, в каком мы описываем переменные в секциях `var`: сначала имена переменных (в данном случае полей) через запятую, затем двоеточие, имя типа и точка с запятой. Переменная типа `CheckPoint` будет, таким образом, представлять собой запись с целочисленными полями `n` и `penalty`, полями `latitude` и `longitude` типа `real` и полем `hidden` логического типа. Опишем такую переменную:

```
var
    cp: CheckPoint;
```

Переменная cp занимает столько памяти, сколько нужно, чтобы разместить все её поля; сами эти поля доступны «через точку»: коль скоро cp есть имя переменной типа «запись», то cp.n, cp.latitude, cp.longitude, cp.hidden и cp.penalty — это её поля, которые тоже, конечно же, представляют собой переменные. Например, мы можем занести в переменную cp данные какого-нибудь контрольного пункта:

```
    cp.n := 70;
    cp.latitude := 54.83843;
    cp.longitude := 37.59556;
    cp.hidden := false;
    cp.penalty := 30;
```

Как и в случае с массивами, при работе с записью большинство действий, если не все, производятся над её полями; единственное, что можно сделать с самой записью как единым целым — это присваивание.

2.8.6. Конструирование сложных структур данных

Полем записи вполне может быть массив и даже другая запись (хотя этот вариант нужен сравнительно редко). Точно так же и запись может быть элементом массива; например, мы могли бы в начале программы задать константу, обозначающую общее количество контрольных пунктов:

```
const
    MaxCheckPoint = 75;
```

и после описания типа CheckPoint (как это сделано в предыдущем параграфе) описать тип для массива, в который можно занести информацию обо всех контрольных пунктах дистанции и описать переменную такого типа:

```
type
    CheckPointArray = array [1..MaxCheckPoint] of CheckPoint;
var
    track: CheckPointArray;
```

Получившаяся в переменной track структура данных соответствует нашему представлению о *таблице*. Когда мы строим таблицу на бумаге, обычно сверху мы выписываем названия её столбцов, а дальше у нас идут строки, содержащие как раз «записи». В только что построенной структуре данных роль заголовка таблицы с названиями столбцов играют имена полей — n, latitude, longitude, hidden и penalty. Строкам

таблицы соответствуют элементы массива: `track[1]`, `track[2]`, и т. д., каждый из которых представляет собой запись типа `CheckPoint`, а отдельная клетка таблицы оказывается полем соответствующей записи: `track[7].latitude`, `track[63].hidden` и т. п.

Элементом массива также может быть и другой массив. Такая ситуация встречается столь часто, что имеет и собственное название — «многомерный массив», и специальный синтаксис для её описания: при задании типа массива мы можем в квадратных скобках написать не один тип индекса, а несколько, разделив их запятой. Например, следующие описания типов эквивалентны:

```
type
    array1 = array [1..5, 1..7] of integer;
    array2 = array [1..5] of array [1..7] of integer;
```

Точно так же при обращении к элементу многомерного (в данном случае — двумерного) массива можно с одинаковым успехом написать `a[2][6]` и `a[2,6]`.

Наиболее очевидное применение многомерных массивов — для представления математического понятия *матрицы*; так, при решении системы линейных уравнений нам могла бы потребоваться какая-нибудь матрица вроде

```
type
    matrix5x5 = array [1..5, 1..5] of real;
```

Многомерные массивы тоже могут быть полями записей, как и записи могут быть их элементами. Формальных ограничений на глубину вложенности описаний типов в Паскале не существует, но увлекаться этим всё же не стоит: память компьютера отнюдь не бесконечна, и вдобавок некоторые конкретные ситуации могут накладывать дополнительные ограничения. Например, прежде чем делать огромный массив *локальной переменной* в подпрограмме, следует десять раз подумать: локальные переменные располагаются в *стековой памяти*, которой может оказаться гораздо меньше, чем вы ожидали.

2.8.7. Пользовательские типы и параметры подпрограмм

При передаче в подпрограммы значений (и переменных), имеющих пользовательские типы, следует учитывать некоторые моменты, не вполне очевидные для начинающих.

Прежде всего следует отметить, что **при описании параметров в заголовках подпрограмм запрещено использование анонимных типов**. Иначе говоря, вызовет ошибку попытка написать что-то вроде следующего:

```
procedure p1(b: 1..100);
begin
    writeln(b)
end;
```

Чтобы тип можно было использовать при передаче параметра в подпрограмму, этот тип обязательно нужно описать в секции описания типов и снабдить именем:

```
type
    MyRange = 1..100;

procedure p1(b: MyRange);
begin
    writeln(b)
end;
```

То же самое можно сказать и о возврате значения из функции: для этого годятся только типы, имеющие имена.

Второй момент, достойный упоминания, касается передачи в подпрограммы (и возврата из функций) значений сложных типов, занимающих большое количество памяти. Паскаль, в принципе, не запрещает этого делать: можно передать в процедуру большую запись или массив, и программа успешно пройдёт компиляцию и даже будет как-то работать. Необходимо только помнить две вещи. Во-первых, как уже упоминалось, *локальные переменные* (в том числе параметры-значения) располагаются в памяти в области аппаратного стека, где может быть мало места. Во-вторых, само по себе копирование больших объёмов данных *занимает ненулевое время*, что можно замечательно ощутить, если делать такие вещи часто — например, вызывать какую-нибудь подпрограмму в цикле, каждый раз передавая ей параметром по значению массив большого размера.

Поэтому по возможности следует воздерживаться от передачи в подпрограммы значений, занимающих значительный объём памяти; если же этого никак не удаётся избежать, то лучше будет использовать var-параметр, даже если вы не собираетесь в передаваемой переменной ничего менять. Дело в том, что при передаче любого значения через var-параметр никакого копирования не происходит. Мы уже говорили, что локальное имя (имя var-параметра) на время работы подпрограммы становится *синонимом* той переменной, которая указана при вызове; отметим, что на уровне машинного кода этот синоним реализуется путём передачи *адреса*, а адрес занимает не так много места — 4 байта на 32-битных системах и 8 байт на 64-битных.

Итак, **если вы работаете с переменными, имеющими значительный размер, лучше всего вообще не передавать их параметрами в подпрограммы; если это всё же приходится делать,**

то их следует передавать как var-параметры; и лишь в совсем крайнем случае (который обычно не возникает) можно попытаться передать такую переменную по значению и надеяться, что всё будет хорошо.

Конечно, возникает вопрос, какие переменные считать имеющими «значительный размер». Конечно, размер в 4 или 8 байт «значительным» не является. Можно не слишком беспокоиться по поводу копирования 16 байт и даже 32, но вот если размер вашего типа ещё больше — то передача объектов такого типа по значению сначала становится «нежелательной», потом где-то на уровне 128 байт — «крайне нежелательной», а где-нибудь на уровне 512 байт — недопустимой, несмотря на то, что компилятор возражать не станет. Если же вам придёт в голову передать по значению структуру данных, занимающую килобайт или больше, то постарайтесь хотя бы никому не показывать ваш исходный текст: велик риск, что, увидев такие выкрутасы, с вами больше не захотят иметь дела.

2.8.8. Преобразования типов

Если в программе описать переменную типа `real`, а затем присвоить ей целое число:

```
var
    r: real;
begin
    { ... }
    r := 15;
```

то, как ни странно, ничего страшного при этом не произойдёт: в переменную `r` будет занесено совершенно корректное и естественное для такой ситуации значение `15.0`. Точно так же можно переменной типа `real` присвоить значение произвольного целочисленного выражения, в том числе вычисляемого во время исполнения программы; присвоено в итоге окажется, разумеется, число с плавающей точкой, целая часть которого равна значению выражения, а дробная часть — нулю.

Начинающие зачастую не задумываются о том, что́ на самом деле происходит в такой ситуации; между тем, как мы знаем, машинные представления целого числа `15` и числа с плавающей точкой `15.0` друг на друга совершенно не похожи. Обычное присваивание, при котором в переменную заносится значение выражения точно такого же типа, сводится (на уровне машинного кода) к простому копированию информации из одного места в другое; присваивание целого числа переменной с плавающей точкой так реализовать нельзя, необходимо *преобразовать* одно представление в другое. Так мы приходим к понятию ***преобразования типов***.

Случай «магического превращения» целого в дробное относится к так называемым ***неявным преобразованиям типов***; так говорят, когда компилятор преобразует тип выражения сам, без прямых указаний со стороны программиста. Возможности неявных преобразований в Паскале довольно скромные: разрешается неявно преобразовывать числа различных целочисленных типов друг в друга и в числа с плавающей точкой, плюс можно превращать друг в друга сами числа с плавающей точкой, которых тоже есть несколько типов — `single`, `double` и `extended`, а знакомый нам `real` является синонимом одного из трёх, в нашем случае — `double`. Чуть позже мы встретимся с неявным преобразованием символа в строку, но этим всё закончится; больше компилятор ничего по своей инициативе преобразовывать не станет. Неявные преобразования встречаются не только при присваивании, но и при вызове подпрограмм: если какая-нибудь процедура ожидает параметр типа `real`, мы совершенно спокойно можем указать при вызове целочисленное выражение, и компилятор нас поймёт.

Заметим, что целое число компилятор согласен «магически превратить» в число с плавающей точкой, но делать преобразование в обратную сторону он откажется наотрез: если мы попытаемся присвоить целочисленной переменной значение, имеющее тип числа с плавающей точкой, при компиляции будет выдана ошибка. Дело здесь в том, что *преобразовать дробное в целое можно разными способами*, и компилятор отказывается делать за нас выбор такого способа; мы должны указать, *каким конкретно способом* число должно быть избавлено от дробной части. Вариантов у нас, собственно говоря, два: преобразование путём отбрасывания дробной части и путём математического округления; для первого варианта используется встроенная функция `trunc`, для второго — `round`. Например, если у нас есть три переменные:

```
var
    i, j: integer;
    r: real;
```

то после выполнения присваиваний

```
    r := 15.75;
    i := trunc(r);
    j := round(r);
```

в переменной `i` будет число `15` (результат «тупого» отбрасывания дробной части), а в переменной `j` — число `16` (результат округления к ближайшему целому). На самом деле функции `round` и `trunc` производят не преобразование типа, а некое *вычисление*, ведь в общем случае (при ненулевой дробной части) полученное значение отличается от исходного. К разговору о преобразованиях типов эти функции имеют

лишь опосредованное отношение: было бы не совсем честно заявить, что соответствующее неявное преобразование запрещено, и при этом не объяснить, что делать, если оно всё-таки потребовалось.

Отметим один немаловажный момент. Неявным преобразованим могут подвергаться значения выражений, **но не переменные**; это означает, что при передаче переменной в подпрограмму через `var`-параметр тип переменной должен точно совпадать с типом параметра, никакие вольности тут не допускаются; например, если ваша подпрограмма имеет `var`-параметр типа `integer`, то вы не сможете передать ей переменную типа `longint`, или типа `word`, или любого другого — только `integer`. Это имеет простое объяснение: машинный код подпрограммы сформирован в расчёте на переменную типа `integer` и при этом ничего не знает и не может знать о том, что на самом деле произошло в точке вызова, так что, если бы вместо `integer` компилятор допускал передачу, например, переменной типа `longint`, подпрограмма никак не могла бы узнать, что ей подсунули переменную не того типа. С преобразованиями значений (в отличие от переменных) всё гораздо проще: все преобразования компилятор делает в точке вызова, а тело подпрограммы получает уже именно тот тип, которого ждёт.

Кроме неявных преобразований, Free Pascal[33] поддерживает также ***явные преобразования типов***, причём такие преобразования возможны как для значений выражений, так и *для переменных*. Слово «явный» здесь означает, что программист сам — *в явном виде* — указывает, что вот здесь необходимо преобразовать выражение к другому типу или временно считать некую переменную имеющей другой тип. В Паскале синтаксис такого явного преобразования напоминает вызов функции с одним параметром, но вместо имени функции указывается имя типа. Например, выражения `integer('5')` и `byte('5')` оба будут иметь целочисленное значение 53, в первом случае это будет двухбайтное знаковое, во втором — однобайтовое беззнаковое. Точно так же мы можем описать переменную типа `char` и временно рассмотреть её в роли переменной типа `byte`:

```
var
    c: char;
begin
    { ... }
    byte(c) := 65;
```

Конечно, далеко не любые типы можно преобразовать друг в друга. Free Pascal разрешает явные преобразования в двух случаях: (1) *порядковые типы* можно преобразовывать друг в друга без ограничений, причём это касается не только целых чисел, но и `char`, и `boolean`, и перечислимых типов; и (2) можно преобразовать друг в друга типы, имеющие одинаковый размер машинного представления. Что касается преобразования типов переменных, то первый случай для них не работает и требуется *обязательное* совпадение размеров.

[33] Вслед за компиляторами Turbo Pascal и Delphi, но в отличие от классических вариантов Паскаля, где ничего подобного не было.

С явными преобразованиями типов следует соблюдать определённую осторожность, поскольку происходящее не всегда будет совпадать с вашими ожиданиями. Например, код символа лучше всё же получать с помощью функции `ord`, а создавать символ по заданному коду — с помощью `chr`, которые специально для этого предназначены. **Если вы не совсем уверены, что вам нужно явное преобразование типов, или знаете, как без него обойтись — то лучше вообще его не применять**. Заметим, что обойтись без таких преобразований можно *всегда*, но в некоторых сложных ситуациях возможность сделать преобразование позволяет сэкономить трудозатраты; иной вопрос, что столь заковыристые случаи вам ещё очень долго не встретятся.

2.9. Строки

2.9.1. Строковые литералы и массивы `char`'ов

Мы уже неоднократно встречались со строками в виде *строковых литералов* — заключённых в апострофы фрагментов текста; до сей поры они попадались нам только в операторах `write` и `writeln`, а писали мы их с одной-единственной целью: немедленно выдать на печать.

Между тем, как мы уже обсуждали во вводной части, *текст* — это наиболее универсальное представление для едва ли не любой информации; единственным исключением выступают данные, полученные путём оцифровки тех или иных аналоговых процессов — фотографии, звуковые и видеофайлы, которые тоже, в принципе, возможно представить в виде текста, но очень неудобно. Всё остальное, включая, например, картинки, нарисованные человеком с помощью графических редакторов, представлять в виде текста удобно и практично; очевидным следствием этого является острая необходимость уметь писать программы, которые обрабатывают текстовую информацию.

С простейшими случаями обработки текстов мы уже встречались в параграфе, посвящённом программам-фильтрам (см. § 2.7.3), где получали один текст из другого, который анализировали с помощью посимвольного чтения. На практике обработка текстов по одному символу часто оказывается неудобной и хочется рассматривать фрагменты текстов — то есть *строки* — как единое целое.

Поскольку строка — это последовательность символов, а символы можно представить с помощью значений типа `char`, логично предположить, что для обработки строк можно использовать *массивы элементов типа* `char`. Это действительно возможно; Паскаль даже допускает для таких массивов довольно странные действия, которые ни с какими иными массивами делать нельзя. Например, мы знаем, что массивы можно присваивать друг другу, только если они имеют строго один и тот же тип; но массиву, состоящему из `char`'ов, Паскаль позволяет присвоить

значение, заданное ***строковым литералом***, то есть хорошо знакомой нам последовательностью символов, заключённой в апострофы:

```
program hello_string;
var
    hello: array [1..30] of char;
begin
    hello := 'Hello, world!';
    writeln(hello)
end.
```

Эта программа успешно пройдёт компиляцию и даже, *на первый взгляд*, нормально отработает, напечатав сакраментальное «`Hello, world!`». Но лишь на первый взгляд; на самом же деле после привычной нам фразы программа «напечатает» ещё 17 символов с кодом `0` и лишь затем выдаст перевод строки и завершится; в этом можно легко убедиться самыми разными способами: перенаправить вывод в файл и посмотреть на его размер, запустить программу конвейером в связке со знакомой нам программой `wc`, применить `hexdump` (можно сразу к выводу, можно к получившемуся файлу). Символ с кодом `0`, будучи выдан на печать, никак себя на экране не проявляет, даже курсор никуда не двигается, так что этих 17 «лишних» нулей не видно — но они есть и могут быть обнаружены. Что особенно неприятно, так это то, что формально в текстовых данных этот незримый «нулевой символ» встречаться не имеет права, так что выдача нашей программы перестала, формально говоря, быть корректным текстом.

Догадаться, откуда взялись эти паразитные «нолики», несложно: фраза «`Hello, world!`» состоит из 13 символов, а массив мы объявили из 30 элементов. Оператор присваивания, скопировав 13 символов из строкового литерала в элементы массива `hello[1]`, `hello[2]`, ..., `hello[13]`, остальные элементы, за неимением лучшего, забил нулями.

Конечно, программу можно исправить и сделать корректной, например, вот так (напомним, что оператор `break` досрочно прекращает выполнение цикла):

```
program hello_string;
var
    hello: array [1..30] of char;
    i: integer;
begin
    hello := 'Hello, world!';
    for i := 1 to 30 do
    begin
        if hello[i] = #0 then
            break;
        write(hello[i])
```

```
    end;
    writeln
end.
```

но уж очень это громоздко — мы ведь в очередной раз вынуждены обрабатывать строку по одному символу!

Ситуация, когда нам нужно обрабатывать строку, не зная заранее её длины — совершенно типична. Представьте себе, что вам нужно спросить у пользователя, как его зовут; один ответит лаконичным «Вова», а другой заявит, что он не менее как Остап-Сулейман-Берта-Мария Бендер-Задунайский. Вопрос, можно ли это предугадать на этапе написания программы, следует считать риторическим. В связи с этим для работы со строками желательно иметь какое-то гибкое средство, учитывающее вот это вот фундаментальное свойство строки — *иметь непредсказуемую длину.* Кроме того, над строками очень часто выполняется операция ***конкатенации*** (присоединения одной строки к другой), и её желательно обозначить как-нибудь так, чтобы её вызов был для программиста необременителен. Между прочим, частный случай конкатенации — добавление к строке одного символа — настолько распространён, что, привыкнув к нему, вы в будущем при работе на Си (где это действие требует куда больших усилий) ещё долго будете эту возможность вспоминать с ностальгией.

Так или иначе, для решения разом большинства проблем, возникающих при работе со строками, в Паскале — точнее, в его поздних диалектах, включая Turbo Pascal и, конечно, наш Free Pascal, предусмотрено специальное семейство типов, которым будет посвящён следующий параграф.

Отметим, что в составе строкового литерала мы можем изобразить не только символы, имеющие печатное представление, но и любые другие — через их коды. В примерах нам уже встречались строки, оканчивающиеся символом перевода строки, такие как `'Hello'#10`; «хитрые» символы можно вставлять не только в начало или конец строкового литерала, но и в произвольное место строки, например, `'one'#9'two'` — здесь два слова разделены символом табуляции. Вообще в составе строкового литерала можно произвольным образом чередовать последовательности символов, заключённых в апострофы, и символы, заданные их кодами; например,

```
'first'#10#9'second'#10#9#9'third'#10#9#9#9'fourth'
```

есть валидный строковый литерал, результат выдачи которого на печать (за счёт вставленных в него табуляций и переводов строки) будет выглядеть примерно так:

```
first
```

```
        second
                third
                        fourth
```

Символы с кодами от 1 до 26 можно также изображать как ^A, ^B, ..., ^Z; это оправдывается тем, что при вводе с клавиатуры символы с соответствующими кодами, как уже упоминалось, порождаются комбинациями Ctrl-A, Ctrl-B и т. д. В частности, литерал из нашего примера можно записать и так:

```
'first'^J^I'second'^J^I^I'third'^J^I^I^I'fourth'
```

2.9.2. Тип string

Введённый в Паскале специально для работы со строками тип string фактически представляет собой частный случай массива из элементов типа char, но случай довольно нетривиальный. Прежде всего отметим, что при описании переменной типа string можно указать, а можно не указывать предельный размер строки, но «бесконечной» строка от этого не станет: максимальная её длина ограничена 255 символами. Например:

```
var
    s1: string[15];
    s2: string;
```

Переменная s1, описанная таким образом, может содержать строку длиной до 15 символов включительно, а переменная s2 — строку длиной до 255 символов. Указывать число больше 255 нельзя, это вызовет ошибку, и этому есть довольно простое объяснение: *тип string предполагает хранение длины строки в отдельном байте*, ну а байт, как мы помним, число больше 255 хранить не может.

Переменная типа string занимает на один байт больше, чем предельная длина хранимой строки: например, наши переменные s1 и s2 будут занимать соответственно 16 и 256 байт. С переменной типа string можно работать как с простым массивом элементов типа char, при этом индексация элементов, содержащих символы строки, начинается с единицы (для s1 это будут элементы с s1[1] по s1[15], для s2 — элементы s2[1] по s2[255]), но — несколько неожиданно — в этих массивах обнаруживается ещё один элемент, имеющий индекс 0. Это и есть тот самый байт, который используется для хранения длины строки; поскольку элементы массива обязаны быть одного типа, этот байт, если к нему обратиться, имеет тот же тип, что и остальные элементы — то есть char. К примеру, если выполнить присваивание

```
s1 := 'abrakadabra';
```

то выражение `s1[1]` будет равно `'a'`, выражение `s1[5]` — `'k'`; поскольку в слове «`abrakadabra`» 11 букв, последним осмысленным элементом будет `s1[11]`, он тоже равен `'a'`, значения элементов с бóльшими индексами не определены (там может содержаться всё что угодно, и особенно то, что **не** угодно). Наконец, в элементе `s1[0]` будет содержаться длина, но поскольку `s1[0]` — это выражение типа `char`, было бы неправильно сказать, что оно будет равно `11`; на самом деле оно будет равно *символу с кодом 11*, который обозначается как `#11` или `^K`, а длину строки можно получить, вычислив выражение `ord(s1[0])`, которое и даст искомое `11`. Впрочем, есть более общепринятый способ узнать длину строки: воспользоваться встроенной функцией `length`, которая специально для этого предназначена.

Вне всякой зависимости от предельной длины все переменные и выражения типа `string` в Паскале совместимы по присваиванию, то есть мы можем заставить компилятор выполнить как `s2 := s1`, так и `s1 := s2`, причём в этом втором случае строка при присваивании может оказаться обрезана; в самом деле, возможности `s2` несколько шире, никто не мешает этой переменной содержать строку, скажем, в 50 символов длиной, но в `s1` больше 15 символов загнать нельзя.

Что особенно приятно, при присваивании стрингов, как и при передаче их по значению в подпрограммы, и при возврате из функций копируется только значащая часть переменной; например, если в переменной, объявленной как `string` без указания предельной длины, хранится строка из трёх символов, то только эти три символа (плюс байт, содержащий длину) и будут копироваться, несмотря на то, что переменная целиком занимает 256 байт.

Ещё интереснее, что, если ваша подпрограмма принимает `var`-параметр типа `string`, то подать в качестве этого параметра можно *любую* переменную типа `string`, в том числе такую, для которой при описании ограничена длина. Это исключение из общего правила (о тождественном совпадении типа переменной с типом `var`-параметра) выглядит некрасиво и небезопасно, но оказывается очень удобным на практике.

Переменные типа `string` можно «складывать» с помощью символа «+», который для строк означает соединение их друг с другом. Например, программа

```
program abrakadabra;
var
    s1, s2: string;
begin
    s1 := 'abra';
    s2 := s1 + 'kadabra';
    writeln(s2)
end.
```

напечатает, как можно догадаться, слово «abrakadabra».

Строки бывают *пустыми*, то есть не содержащими ни одного символа. Длина такой строки равна нулю; литерал, обозначающий пустую строку, выглядит как «''» (два символа апострофа, поставленные рядом); этот литерал не следует путать с «' '», который обозначает символ пробела (или же строку, состоящую из одного символа — пробела).

Выражения типа char практически во всех случаях могут быть неявно преобразованы к типу string — строке, содержащей ровно один символ. Это особенно удобно в сочетании с операцией сложения. Так, программа

```
{ a_z.pas }
program a_z;
var
    s: string;
    c: char;
begin
    s := '';
    for c := 'A' to 'Z' do
        s := s + c;
    writeln(s)
end.
```

напечатает «ABCDEFGHIJKLMNOPQRSTUVWXYZ».

На самом деле Free Pascal поддерживает целый ряд типов для работы со строками, причём многие из этих типов не имеют ограничений, присущих типу string — то есть могут, например, хранить в себе текст совершенно произвольной длины, работать с многобайтовыми кодировками символов и т. п. Мы не будем их рассматривать, поскольку при освоении материала следующих частей книги всё это великолепие не потребуется; читатель, решивший сделать Free Pascal своим рабочим инструментом (в противоположность учебному пособию), может освоить эти возможности самостоятельно.

2.9.3. Встроенные функции для работы со строками

Без материала этого параграфа можно легко обойтись, ведь со строками можно работать на уровне отдельных символов, а значит, мы можем сделать с ними буквально что угодно без всяких дополнительных средств. Тем не менее, при обработке строк ряд встроенных процедур и функций может существенно облегчить жизнь, поэтому мы всё же приведём некоторые из них.

Одну функцию мы уже знаем: length принимает на вход выражение типа string и возвращает целое число — длину строки. Текущую длину строки можно изменить принудительно процедурой SetLength; например, после выполнения

```
s := 'abrakadabra';
SetLength(s, 4);
```

в переменной s будет содержаться строка «abra». Учтите, что SetLength может также и увеличить длину строки, в результате чего в конце её окажется «мусор» — непонятные символы, которые лежали в этом месте памяти до того, как там разместили строку; поэтому если вы решили увеличить длину строки с помощью SetLength, правильнее всего будет немедленно заполнить чем-нибудь осмысленным все её «новые» элементы. Учтите, что меняется только *текущая длина* строки, но никоим образом не размер области памяти, которая под эту строку выделена. В частности, если вы опишете строковую переменную на десять символов

```
s10: string[10];
```

а затем попытаетесь установить ей длину, превышающую 10, ничего хорошего у вас не получится: переменная s10 *не может* содержать строку длиннее десяти символов.

Все процедуры и функции, перечисленные ниже до конца параграфа, выполняют действия, которые необходимо уметь делать самостоятельно; мы крайне не рекомендуем пользоваться этими средствами, пока вы не научитесь свободно обращаться со строками на уровне посимвольной обработки. Каждую из перечисленных ниже функций и процедур можно начинать использовать не раньше, чем вы убедитесь, что можете сделать то же самое «вручную».

Встроенные функции LowerCase и UpCase принимают на вход выражение типа string и возвращают тоже string — такую же строку, какая содержалась в параметре, за исключением того, что латинские буквы оказываются приведены соответственно к нижнему или к верхнему регистру (иначе говоря, первая функция в строке заменяет заглавные буквы на строчные, а вторая, наоборот, строчные на заглавные).

Функция copy принимает на вход три параметра: строку, начальную позицию и количество символов, и возвращает *подстроку* заданной строки, начиная с заданной начальной позиции, длиной как заданное количество символов, либо меньше, если символов в исходной строке не хватило. Например, copy('abrakadabra', 3, 4) вернёт строку 'raka', а copy('foobar', 4, 5) — строку 'bar'.

Процедура delete тоже принимает на вход строку (на этот раз через параметр-переменную, потому что строка будет изменена), начальную позицию и количество символов, и *удаляет* из данной строки (прямо на месте, то есть в той переменной, которую вы передали параметром) заданное количество символов, начиная с заданной позиции (либо до конца строки, если символов не хватило). Например, если в переменной s содержалось всё то же «abrakadabra», то после выполнения delete(s, 5, 4) в переменной s окажется «abrabra»; а если бы мы применили delete(s, 5, 100), получилось бы просто «abra».

Встроенная процедура insert вставляет одну строку в другую. Первый параметр задаёт вставляемую строку; вторым параметром задаётся *переменная* строкового типа, в которую надлежит вставить заданную строку. Наконец, третий параметр (целое число) указывает позицию, начиная с которой следует произвести вставку. Например, если в переменную s занести строку «abcdef», а потом выполнить insert('PQR', s, 4), то после этого в переменной s будет находиться строка «abcPQRdef».

Функция pos принимает на вход две строки: первая задаёт подстроку для поиска, вторая — строку, в которой следует производить поиск. Возвращается целое число, равное позиции подстроки в строке, если таковая найдена, или 0, если не найдена. Например, pos('kada', 'abrakadabra') вернёт 5, а pos('aaa', 'abrakadabra') вернёт 0

Может оказаться очень полезной процедура val, которая строит число типа longint, integer или byte по его строковому представлению. Первым параметром в процедуру подаётся строка, в которой должно содержаться текстовое представление числа (возможно, с некоторым количеством пробелов перед ним); вторым параметром должна быть переменная типа longint, integer или byte; третьим параметром указывается ещё одна переменная, всегда имеющая тип word. Если всё прошло успешно, во второй параметр процедура занесёт полученное число, в третий — число 0; если же произошла ошибка (то есть строка не содержала корректного представления числа), то в третий параметр заносится номер позиции в строке, где перевод в число дал сбой, а второй параметр в этом случае остаётся неопределённым.

Паскаль также предоставляет средства для обратного перевода числа в строковое представление. Если представление требуется десятичное, то можно воспользоваться псевдопроцедурой str; здесь можно использовать спецификаторы количества символов аналогично тому, как мы это делали для печати в операторе write (см. стр. 209). Например, str(12.5:9:3, s) занесёт в переменную s строку « 12.500» (с тремя пробелами в начале, чтобы получилось ровно девять символов).

Для перевода в двоичную, восьмеричную и шестнадцатеричную систему также имеются встроенные средства, но на этот раз это почти обычные функции, которые называются BinStr, OctStr и HexStr. В отличие от str, они работают только с целыми числами и являются функциями, то есть полученную строку возвращают в качестве своего значения, а не через параметр. Все три зачем-то предусматривают два параметра: первый — целое число произвольного типа, второй — количество символов в результирующей строке.

2.9.4. Обработка параметров командной строки

Запуская разные программы в командной строке ОС Unix, мы часто указывали при запуске, кроме имени программы, ещё ***параметры командной строки***. Например, при запуске редактора текстов и компилятора fpc мы всегда указываем имя файла, в котором содержится или будет содержаться текст нашей программы. Ясно, что и редактор текстов, и компилятор тоже представляют собой программы и, следовательно, как-то могут обращаться к своим параметрам командной строки; может так поступить и программа, которую мы сами пишем на Паскале (да и на любом другом языке тоже).

Пусть наша программа называется «demo»; пользователь может запустить её, например, так:

```
avst@host:~/work$ ./demo abra schwabra kadabra
```

В этом случае говорят, что командная строка состоит из *четырёх* параметров: самого имени программы, слова «abra», слова «schwabra» и слова «kadabra».

В программе на Паскале эти параметры доступны с помощью встроенных функций ParamCount и ParamStr; первая, не получая никаких параметров, возвращает целое число, соответствующее *количеству параметров без учёта имени программы* (в нашем примере это будет число 3); вторая функция принимает на вход целое число и возвращает строку (string), соответствующую параметру командной строки с заданным номером. При этом имя программы считается параметром номер 0 (то есть его можно узнать с помощью выражения ParamStr(0)), а остальные нумеруются с единицы до числа, возвращённого функцией ParamCount.

Напишем для примера программу, которая печатает все элементы своей командной строки, сколько бы их ни оказалось:

```
{ cmdline.pas }
program cmdline;
var
    i: integer;
begin
    for i := 0 to ParamCount do
        writeln('[', i, ']: ', ParamStr(i))
end.
```

После компиляции мы можем попробовать эту программу в действии, например:

```
avst@host:~/work$ ./cmdline abra schwabra kadabra
[0]: /home/avst/work/cmdline
[1]: abra
[2]: schwabra
[3]: kadabra
avst@host:~/work$ ./cmdline
[0]: /home/avst/work/cmdline
avst@host:~/work$ ./cmdline "one two three"
[0]: /home/avst/work/cmdline
[1]: one two three
avst@host:~/work$
```

Следует обратить внимание на то, что фраза из трёх слов, заключённая в двойные кавычки, оказалась воспринята как один параметр; это не имеет отношения к языку Паскаль и является свойством командного интерпретатора.

2.10. Оператор выбора

Оператор выбора представляет собой в определённом смысле обобщение оператора ветвления (`if`): в отличие от ветвления, где предусмотрено всего два варианта исполнения, оператор выбора позволяет задать таких вариантов сколько угодно. Во время выполнения программы вычисляется заданное выражение, которое может иметь любой *порядковый тип* (см. § 2.8.3); в зависимости от получившегося значения выбирается одна из предусмотренных в теле оператора ветвей выполнения.

В Паскале оператор выбора начинается с ключевого слова `case`, после которого ставится произвольное арифметическое выражение, имеющее порядковый тип. Завершение выражения отмечается ключевым словом `of`; после него следует некоторое количество (не меньше одной) *ветвей*, причём каждая состоит из двух частей: набора значений, для которых данная ветвь должна выполняться (то есть ветвь будет выполняться, если результатом вычисления выражения, стоящего после `case`, стало одно из заданных значений) и оператора (возможно, составного, хотя не обязательно). Значения от оператора отделяются двоеточием. В конце тела оператора `case` можно (но не обязательно) поставить ветку `else`, которая состоит из ключевого слова `else` и оператора; этот оператор выполняется, если значение выражения в заголовке `case` не соответствует ни одной из ветвей.

Значение для ветви можно задать одно, можно перечислить несколько значений через запятую; наконец, можно задать *диапазон* значений через две точки. Следует помнить, что **значения в операторе выбора должны задаваться константами времени компиляции** (см. § 2.5.2).

Приведём пример. Следующая программа читает с клавиатуры один символ и классифицирует его как относящийся к одной из категорий:

```
program SymbolType;
var
    c: char;
begin
    read(c);
    write('The symbol is ');
    case c of
        'a'..'z', 'A'..'Z':
            writeln('a latin letter');
        '0'..'9':
            writeln('a digit');
        '+', '-', '/', '*':
            writeln('an arithmetic operation symbol');
        '<', '>', '=':
            writeln('a comparision sign');
```

```
        '.', ',', ';', ':', '!', '?':
            writeln('a punctuation symbol');
        '_', '~', '@', '#', '$', '%', '^',
        '&', '`', '|', '\':
            writeln('a special purpose sign');
        ' ':
            writeln('the space character');
        #9, #10, #13:
            writeln('a formatting code');
        #27:
            writeln('the escape code');
        '(', ')', '[', ']', '{', '}':
            writeln('a grouping symbol');
        '"', '''':
            writeln('a quoting symbol');
        else
            writeln('something strange')
    end
end.
```

Оператор выбора часто провоцирует очень серьёзную проблему, связанную со стилем написания программы. В исходниках начинающих программистов можно найти примерно такие фрагменты:

```
case nv of
    1: begin
       {...}
    end;
    2: begin
       {...}
    end;
    3: begin
       {...}
    end;
    {...}
    14: begin
       {...}
    end
end
```

Определяющим тут является то, что case-выражение имеет обыкновенный целый тип, а варианты обозначены числами 1, 2, 3 и так далее, иногда вплоть до достаточно больших чисел (автору этих строк доводилось видеть подобные конструкции на 30 и больше вариантов). Так вот, **за такое программирование увольняют с работы**, и правильно делают. В самом деле, как прикажете **это** читать? Вот что такое, к примеру, 3, то есть что оно в данном случае обозначает? Найти ответ на такой вопрос можно, перелопатив программу вдоль и поперёк, выяснив,

откуда берётся значение этой переменной `nv`, в каких случаях она принимает одни значения, в каких другие; у читателя программы уйдёт на это прорва времени. Но это ещё не самое страшное. Перепутать между собой значения такого рода может и сам автор программы, вернув по ошибке из какой-нибудь функции, скажем, число `7` вместо числа `5`. В целом программа, внутренняя логика которой основана на «номерах вариантов», на удивление быстро и лихо «отбивается от рук».

Правильно в таком случае будет применять для обозначения разных возможных вариантов не номера этих вариантов, которые никому ничего не говорят, а значения специально введённого для этой цели *перечислимого типа* (см. § 2.8.2), при этом выбрав осмысленные идентификаторы.

Необходимо отметить влияние оператора выбора на размер подпрограмм. Как уже говорилось, идеальная подпрограмма имеет длину не более 25 строк; если в вашей подпрограмме встретился оператор выбора, в большинстве случаев вы в это ограничение не уложитесь. В принципе, это как раз тот случай, когда вполне допустимо чуть превысить указанный размер, но ненамного. **Операторы выбора, вложенные друг в друга, практически всегда недопустимы**. Если в вашем операторе выбора реализация всех или некоторых альтернатив оказалась настолько сложной, следует выделить каждую альтернативу в отдельную подпрограмму, а сам оператор выбора тогда будет состоять из их вызовов.

2.11. Полноэкранные программы

В этой главе мы сделаем явный шаг в сторону от основного пути обучения. Материал, который будет рассказан здесь, совершенно не нужен для постижения последующих глав и примеров из них и никак не будет использоваться в дальнейшем тексте не только этой части, посвящённой Паскалю, но и всей книги. Тем не менее мы воздержимся от традиционной рекомендации «пропустить эту главу, если...».

Напомним то, о чём уже говорили в предисловиях. Первый — и самый, пожалуй, важный — шаг в становлении программиста делается тогда, когда будущий программист переходит от этюдов из задачника к решению задач, *поставленных самостоятельно*, и не потому, что так надо, а потому, что так интереснее. Чем шире спектр *простых* возможностей, доступных начинающему, тем больше шансы, что он придумает для себя что-то такое, что ему *хотелось бы* написать и что он при этом написать *может*.

К сожалению, до графических интерфейсов — во всяком случае, если подходить к ним всерьёз и без риска травмировать собственный мозг — нам пока ещё далеко; но и ограничиваться традиционными консольными

приложениями нас никто не заставляет. С помощью набора довольно нехитрых средств мы к концу этой главы научимся писать программы, работающие в окне эмулятора терминала, но при этом использующие его возможности полностью, не ограничиваясь «каноническим» потоковым вводом (строчка за строчкой) и таким же выводом.

Возможности полноэкранных алфавитно-цифровых пользовательских интерфейсов далеко не так скромны, как может показаться на первый взгляд; помимо знакомых нам редакторов текстов, работающих в терминале, можно отметить, например, почтовый клиент `mutt`, клиент протокола XMPP (известного также как Jabber) `mcabber` и многие другие программы; но действительно адекватное представление об открывающихся возможностях даёт игра NetHack, в которую некоторые люди играют *десятилетиями* (буквально так; известны случаи полного прохождения этой игры в течение 15 лет) и которая построена именно как полноэкранное алфавитно-цифровое приложение.

Умение работать с терминалом «на всю катушку» действительно позволяет создавать динамические игровые программы, пусть и не графические, но от этого ничуть не менее интересные; и, разумеется, этим дело не ограничивается.

Теперь мы можем, в принципе, сказать, при каких условиях эту главу можно пропустить. Вы можете перейти сразу к следующей главе, если вы *уже* придумали для себя интересную задачу, приступили к её решению и полноэкранная работа для этого вам не нужна. В этом случае, пропустив главу о полноэкранных программах, вы просто сэкономите время; во всех остальных случаях попробуйте её прочитать и освоить предлагаемые здесь средства. Вполне возможно, они вам понравятся.

2.11.1. Немного теории

Алфавитно-цифровые терминалы, начиная с самых старых моделей, имевших экран вместо традиционного принтера, поддерживали так называемые escape-последовательности для управления выводом на экран; каждая такая последовательность представляет собой набор кодовых байтов, начинающихся с псевдосимвола с кодом 27 (Escape; отсюда название). Получив такую последовательность, терминал мог, например, переместить курсор в закодированную в последовательности позицию, сменить цвет выводимого текста, произвести скроллинг вверх или вниз и т. п.

Для написания полноэкранных программ этого недостаточно, поскольку драйвер терминала по умолчанию работает с клавиатурой в так называемом *каноническом режиме*, в котором, во-первых, активная программа получает пользовательский ввод строка за строкой, то есть эффект от нажатия какой-нибудь клавиши проявится не ранее, чем пользователь нажмёт Enter, и, во-вторых, некоторые комбинации

клавиш, такие как Ctrl-C, Ctrl-D и прочее, имеют специальный смысл, так что программа не может их «прочитать» как обычную информацию и получает только уже готовый эффект — для Ctrl-C это сигнал `SIGINT`, для большинства программ фатальный, для Ctrl-D это имитация ситуации «конец файла» и т. д. К счастью, все эти особенности поведения драйвера терминала могут быть перепрограммированы, что полноэкранные программы и делают.

Если с перепрограммированием драйвера терминала всё более-менее ясно (достаточно прочитать справочную информацию по слову `termios`), то с escape-последовательностями всё оказывается сложнее. Дело в том, что терминалы когда-то выпускались весьма разнообразные, и наборы escape-последовательностей для них несколько различались; современные программы, эмулирующие терминал, такие как `xterm`, `konsole` и прочие, тоже отличаются друг от друга по своим возможностям. Все эти сложности более-менее покрывается библиотеками, предоставляющими некий набор функций для управления экраном терминала и генерирующими разные escape-последовательности в зависимости от типа используемого терминала. При программировании на языке Си для этих целей обычно используется библиотека `ncurses`, сама по себе довольно сложная.

В версиях Паскаля фирмы Борланд, популярных в эпоху MS-DOS, был предусмотрен специальный библиотечный модуль `crt`[34], который позволял создавать полноэкранные текстовые программы для MS-DOS. Конечно, это не имело ничего общего с терминалами в Unix, управление экраном производилось то через интерфейс BIOS, то вообще путём прямого доступа к видеопамяти. Сейчас всё это представляет интерес разве что археологический — за исключением того, что создатели Free Pascal поставили себе в качестве одной из целей добиться полной совместимости с Turbo Pascal'ем, включая и модуль `crt`; в результате та версия Free Pascal, которую мы с вами используем в unix-системах, содержит свою собственную реализацию модуля `crt`. Эта реализация поддерживает все те функции, которые присутствовали в MS-DOS'овском Turbo Pascal'е, но реализует их, естественно, средствами escape-последовательностей и перепрограммирования драйвера терминала.

Надо отметить, что интерфейс модуля `crt` намного проще, чем у той же библиотеки `ncurses`, и гораздо лучше подходит для начинающих. Именно этим модулем мы и воспользуемся.

Для того, чтобы возможности модуля стали доступны в программе, необходимо сообщить компилятору, что мы собираемся его использовать. Обычно это делается сразу после заголовка программы, перед всеми

[34] От слов *Cathode Ray Tube*, то есть *катодно-лучевая трубка*, она же «кинескоп». Жидкокристаллических «плоских» мониторов, в наше время уже полностью вытеснивших кинескопные, в те времена ещё не существовало.

остальными секциями, в том числе раньше секции констант (хотя не обязательно), например:

```
program tetris;
uses crt;
```

Прежде чем мы начнём обсуждение возможностей модуля, стоит сделать одно предостережение. Как только программа, написанная с использованием модуля `crt`, будет запущена, она тут же перепрограммирует терминал под свои нужды; кроме прочего, это означает, что спасительная комбинация `Ctrl-C` перестанет работать, так что если ваша программа «зависнет» или вы просто забудете предусмотреть корректный способ объяснить ей, что пора завершаться, то придётся вспоминать, как убивать процессы из соседнего окошка терминала. Возможно, стоит перечитать § 1.4.7.

Учтите, что для программ, написанных с использованием модуля `crt`, перенаправления ввода-вывода напрочь лишены смысла; всё это просто не будет работать.

2.11.2. Вывод в произвольные позиции экрана

Начнём с того, что очистим экран, чтобы текст, оставшийся от предыдущих команд, не мешал нашей полноэкранной программе. Это делается процедурой `clrscr`, название которой образовано от слов *clear screen*. Курсор при этом окажется в верхнем левом углу экрана; именно в этом месте появится текст, если сейчас его вывести с помощью обыкновенного `write` или `writeln`. Но мы этого делать не будем; гораздо интереснее *самим указать* то место на экране, куда должно быть выведено сообщение.

Текст, как известно, появляется там, где стоит курсор. Переместить курсор в произвольную позицию экрана позволяет процедура `GotoXY`, принимающая два целочисленных параметра: координату по горизонтали и координату по вертикали. Началом координат считается верхний левый угол, причём его координаты `(1, 1)`, а не `(0, 0)`, как можно было бы ожидать. Узнать ширину и высоту экрана можно, обратившись к глобальным переменным `ScreenWidth` и `SreenHeight`. Эти переменные тоже вводятся модулем `crt`; при старте программы модуль записывает в них актуальное количество доступных нам знакомест в строке и самих строк.

Имея в своём распоряжении `GotoXY`, мы уже можем сделать кое-что интересное. Напишем программу, которая показывает нашу традиционную фразу «`Hello, world!`», но делает это не в рамках нашего диалога с командным интерпретатором, как раньше, а в центре экрана, очищенного от всего постороннего текста. Выдав надпись, уберём курсор обратно в левый верхний угол, чтобы он не портил картину, подождём

пять секунд (это можно сделать с помощью процедуры `delay`, аргументом которой служит целое число, выраженное в тысячных долях секунды; она тоже предоставляется модулем `crt`), снова очистим экран и завершим работу. Длительность задержки, а также текст выдаваемого сообщения мы вынесем в начало программы в виде именованных констант.

Осталось вычислить координаты для печати сообщения. Координату по вертикали мы получим просто как половину высоту экрана (значение `ScreenHeight`, поделенное пополам); что касается координаты по горизонтали, то из ширины экрана (значения `ScreenWidth`) мы вычтем длину нашего сообщения, а оставшееся пространство, опять же, поделим пополам. При таком подходе различие между верхним и нижним полем, как и между правым и левым, не будет превышать единицы; лучшего нам всё равно не достичь, ведь вывод в алфавитно-цифровом режиме возможен только в соответствии с имеющимися знакоместами, сдвинуть текст на половину знакоместа не получится ни горизонтально, ни вертикально. Кстати, не следует забывать, что деление нам тоже потребуется целочисленное, с помощью операции `div`.

Итак, пишем:

```
{ hellocrt.pas }
program HelloCrt;
uses crt;
const
    Message = 'Hello, world!';
    DelayDuration = 5000;  { 5 seconds }
var
    x, y: integer;
begin
    clrscr;
    x := (ScreenWidth - length(Message)) div 2;
    y := ScreenHeight div 2;
    GotoXY(x, y);
    write(Message);
    GotoXY(1, 1);
    delay(DelayDuration);
    clrscr
end.
```

Отметим, что *текущие координаты курсора* можно узнать с помощью функций `WhereX` и `WhereY`; параметров эти функции не принимают. Если мы с помощью `GotoXY` попытаемся переместить курсор в *существующую* позицию, то именно в этой позиции он и окажется, тогда как если мы попробуем его переместить в позицию, которой на нашем экране нет, получившиеся текущие координаты будут какие угодно, но не те, которых мы ожидали. Кроме `GotoXY` текущую позицию курсора, естественно, меняют операции вывода (обычно совместно с модулем `crt` используется оператор `write`).

К сожалению, у описанных средств есть очень серьёзное ограничение: если пользователь изменит размеры окна, в котором работает программа, то она об этом не узнает; значения `ScreenWidth` и `ScreenHeight` останутся теми, какими их установил модуль `crt` при старте. Источник этого ограничения вполне очевиден: в те времена, когда придумывали модуль `crt`, экран *не мог изменить размер*.

2.11.3. Динамический ввод

Для организации ввода с клавиатуры по одной клавише, а также для обработки всяких «хитрых» клавиш вроде «стрелочек», `F1` — `F12` и прочего в таком духе модуль `crt` предусматривает две функции: `KeyPressed` и `ReadKey`. Обе функции не принимают параметров. Функция `KeyPressed` возвращает логическое значение: `true` — если пользователь успел нажать какую-нибудь клавишу, код которой вы пока не прочитали, и `false` — если пользователь ничего не нажимал; функция `ReadKey` позволяет получить код очередной нажатой клавиши. Если вы вызвали `ReadKey` раньше, чем пользователь что-то нажал, функция заблокируется[35] до тех пор, пока пользователь не соизволит что-нибудь нажать; если же клавиша уже была нажата, функция вернёт управление немедленно.

Тип возвращаемого значения функции `ReadKey` — обыкновенный `char`, причём если пользователь нажмёт какую-нибудь клавишу с буквой, цифрой или знаком препинания, то ровно этот символ и будет возвращён. Примерно так же обстоят дела с пробелом (`' '`), табуляцией (`#9`), клавишей Enter (`#13`; обратите внимание, что не `#10`, хотя в какой-нибудь другой версии может получиться и `#10`), Backspace (`#8`), Esc (`#27`). Комбинации `Ctrl-A`, `Crtl-B`, `Ctrl-C`, ..., `Ctrl-Z` дают коды 1, 2, 3, ..., 26, комбинации `Ctrl-[`, `Ctrl-\` и `Ctrl-]` позволяют получить следующие за ними 27, 28 и 29.

Несколько сложнее дело обстоит с остальными служебными клавишами, такими как «стрелочки», Insert, Delete, PgUp, PgDown, F1–F12. Во времена MS-DOS создатели модуля `crt` приняли довольно неочевидное и не вполне красивое решение: использовали так называемые «расширенные коды». На практике это выглядит так: пользователь нажимает клавишу, в программе функция `ReadKey` возвращает символ `#0` (символ с кодом `0`), что означает, что функцию необходимо вызвать во второй раз; символ, возвращённый функцией при этом повторном вызове (а управление она при этом возвращает немедленно), как раз идентифицирует нажатую специальную клавишу. Например, «стрелка влево» даёт коды 0–75, «стрелка вправо» — коды 0–77, «стрелка вверх»

[35] Начинающие в таком случае часто говорят «зависнет», но это неправильно: когда что-то «зависает», вывести его из этого состояния можно только чрезвычайными мерами вроде уничтожения процесса, тогда как простое ожидание события — в данном случае нажатия на клавишу — прекращается, как только это событие настанет, и называется не зависанием, а блокировкой.

и «стрелка вниз» 0–72 и 0–80 соответственно. Следующая простая программа позволит вам выяснить, каким клавишам соответствуют какие коды:

```
{ rdkey.pas }
program RdKey;
uses crt;
var
    c, cc: char;
begin
    repeat
        c := ReadKey;
        cc := c;
        if (cc < #32) or (cc > #126) then
            cc := '?';
        writeln(ord(c), ' (', cc, ')')
    until c = ' '
end.
```

Для завершения работы этой программы нужно нажать пробел.

Версия Free Pascal, имевшаяся у автора этих строк на момент написания книги, некоторые клавиши обрабатывала довольно странно, выдавая последовательность из *трёх* кодов, последним из которых был код 27 (Escape). Выделить такую последовательность очень сложно, ведь она даже не начинается с нуля. Такое поведение демонстрировали клавиша End, цифра 5 на дополнительной клавиатуре при выключенном NumLock, комбинация `Shift-Tab`. Кроме того, комбинации `Ctrl-пробел` и `Ctrl-@` выдавали код 0, за которым ничего не следовало. Всё это противоречит исходной спецификации функции `ReadKey` из турбопаскалевского модуля `crt` и нигде никак не документировано. Вполне возможно, что при использовании других оконных менеджеров и вообще другой среды исполнения проявятся ещё какие-нибудь несообразности.

Чтобы получить общее представление относительно возможностей, которые открывает динамический ввод, мы напишем для начала программу, которая, как и `hellocrt`, будет выводить в середине экрана надпись «`Hello, world!`», *но которую потом можно будет двигать клавишами стрелок*; выход из программы будет производиться по любой клавише, имеющей обычный (не расширенный) код.

В секции констант у нас останется только текст сообщения. Для выдачи сообщения и его убирания с экрана мы напишем процедуры `ShowMessage` и `HideMessage`; последняя будет выдавать в нужной позиции количество пробелов, равное длине сообщения.

Основой программы станет сравнительно короткая процедура `MoveMessage`, принимающая пять параметров: две целочисленные переменные — текущие координаты сообщения на экране; само сообщение в виде строки; два целых числа `dx` и `dy`, задающих изменение координат `x` и `y`.

В главной программе мы сделаем псевдобесконечный цикл, в котором будем читать коды клавиш. Если прочитан обычный (нерасширенный) код, цикл будет прерываться оператором `break`, в результате чего программа, очистив экран, завершится. Если же прочитан расширенный код, то, если он соответствует одной из четырёх стрелочных клавиш, будет вызвана процедура `MoveMessage` с соответствующими параметрами; остальные клавиши с расширенными кодами программа будет игнорировать. Полностью текст программы получается таким:

```
{ movehello.pas }
program MovingHello;
uses crt;
const
    Message = 'Hello, world!';

procedure ShowMessage(x, y: integer; msg: string);
begin
    GotoXY(x, y);
    write(msg);
    GotoXY(1, 1)
end;

procedure HideMessage(x, y: integer; msg: string);
var
    len, i: integer;
begin
    len := length(msg);
    GotoXY(x, y);
    for i := 1 to len do
        write(' ');
    GotoXY(1, 1)
end;

procedure MoveMessage(var x, y: integer; msg: string; dx, dy: integer);
begin
    HideMessage(x, y, msg);
    x := x + dx;
    y := y + dy;
    ShowMessage(x, y, msg)
end;

var
    CurX, CurY: integer;
    ch: char;
begin
    clrscr;
    CurX := (ScreenWidth - length(Message)) div 2;
    CurY := ScreenHeight div 2;
    ShowMessage(CurX, CurY, Message);
    while true do
    begin
        ch := ReadKey;
        if ch <> #0 then  { не расширенный код; выходим }
            break;
```

```
        ch := ReadKey;       { получаем расширенный код }
        case ch of
            #75:                        { стрелка влево }
                MoveMessage(CurX, CurY, Message, -1, 0);
            #77:                       { стрелка вправо }
                MoveMessage(CurX, CurY, Message, 1, 0);
            #72:                        { стрелка вверх }
                MoveMessage(CurX, CurY, Message, 0, -1);
            #80:                         { стрелка вниз }
                MoveMessage(CurX, CurY, Message, 0, 1);
        end
    end;
    clrscr
end.
```

У этой программы имеется серьёзный недостаток: она не отслеживает допустимые значения для координат, так что мы легко можем «вытолкать» сообщение за пределы экрана; после этого оно будет всегда появляться в левом верхнем углу. Исправить это предложим читателю в качестве упражнения.

Рассмотрим более сложный пример. Наша следующая программа выведет в середине пустого экрана символ «звёздочка» (`*`). Сначала символ будет неподвижен, но если нажать любую из четырёх стрелок, символ начнёт двигаться в заданную сторону со скоростью десять знакомест в секунду. Нажатие других стрелок изменит направление его движения, а нажатие пробела остановит. Клавиша Escape завершит программу.

В секции констант у нас снова появится `DelayDuration`, равная `100`, то есть $\frac{1}{10}$ секунды. Это интервал времени, который у нас будет проходить между двумя перемещениями звёздочки.

Учитывая опыт предыдущей программы, мы соберём все данные, задающие текущее состояние звёздочки, в одну запись, которую будем передавать в процедуры как `var`-параметр. Эти данные включают текущие координаты звёздочки, а также направление движения, заданное уже знакомыми нам значениями `dx` и `dy`. Тип для такой записи назовём просто `star`. Процедуры `ShowStar` и `HideStar`, получая единственный параметр (запись типа `star`) будут показывать звёздочку на экране и убирать её, печатая на этом месте пробел; процедура `MoveStar` будет смещать звёздочку на одну позицию в соответствии со значениями `dx` и `dy`. Для удобства опишем также процедуру `SetDirection`, которая заносит заданные значения в поля `dx` и `dy`; наконец, процедура `HandleArrowKey` будет в зависимости от переданного ей расширенного кода вызывать `SetDirection` с нужными значениями параметров. Коды пробела и Escape главная часть программы будет обрабатывать сама.

В главной части программы сначала будет производиться установка начальных значений для звёздочки и её вывод в середине экрана; после этого программа будет входить в псевдобесконечный цикл, в котором,

если пользователь не нажимал никаких клавиш (то есть `KeyPressed` вернула `false`) будет вызываться `MoveStar` и производиться задержка; поскольку в этом случае больше делать ничего не нужно, тело цикла будет досрочно завершаться оператором `continue` (напомним, что, в отличие от `break`, оператор `continue` досрочно завершает только одну итерацию выполнения тела цикла, но не весь цикл). При получении расширенного кода будет вызываться процедура `HandleArrowKey`, при получении символа пробела звёздочка будет останавливаться соответствующим вызовом `SetDirection`, и при получении `Escape` цикл будет завершаться с помощью `break`.

Всё вместе будет выглядеть так:

```
{ movingstar.pas }
program MovingStar;
uses crt;
const
    DelayDuration = 100;

type
    star = record
        CurX, CurY, dx, dy: integer;
    end;

procedure ShowStar(var s: star);
begin
    GotoXY(s.CurX, s.CurY);
    write('*');
    GotoXY(1, 1)
end;

procedure HideStar(var s: star);
begin
    GotoXY(s.CurX, s.CurY);
    write(' ');
    GotoXY(1, 1)
end;

procedure MoveStar(var s: star);
begin
    HideStar(s);
    s.CurX := s.CurX + s.dx;
    if s.CurX > ScreenWidth then
        s.CurX := 1
    else
    if s.CurX < 1 then
        s.CurX := ScreenWidth;
    s.CurY := s.CurY + s.dy;
    if s.CurY > ScreenHeight then
        s.CurY := 1
    else
    if s.CurY < 1 then
        s.CurY := ScreenHeight;
    ShowStar(s);
```

```
end;

procedure SetDirection(var s: star; dx, dy: integer);
begin
    s.dx := dx;
    s.dy := dy
end;

procedure HandleArrowKey(var s: star; extcode: char);
begin
    case extcode of
        #75: SetDirection(s, -1, 0);
        #77: SetDirection(s, 1, 0);
        #72: SetDirection(s, 0, -1);
        #80: SetDirection(s, 0, 1);
    end
end;

var
    s: star;
    ch: char;
begin
    clrscr;
    s.CurX := ScreenWidth div 2;
    s.CurY := ScreenHeight div 2;
    s.dx := 0;
    s.dy := 0;
    ShowStar(s);
    while true do
    begin
        if not KeyPressed then
        begin
            MoveStar(s);
            delay(DelayDuration);
            continue;
        end;
        ch := ReadKey;
        case ch of
            #0: begin
                ch := ReadKey;   { получаем расширенный код }
                HandleArrowKey(s, ch)
            end;
            ' ': SetDirection(s, 0, 0);
            #27: break;
        end
    end;
    clrscr
end.
```

2.11.4. Управление цветом

До сих пор все тексты, появляющиеся в окне терминала в результате работы наших программ, были одного и того же цвета — того, который указан в настройках терминальной программы; но это вполне

можно изменить. Современные эмуляторы терминалов, как и сами терминалы последних моделей (например, DEC VT340, производство которых было прекращено только во второй половине 1990-х годов), формировали на экране цветное изображение и поддерживали escape-последовательности, задающие цвет текста и цвет фона.

К сожалению, интерфейс нашего модуля `crt` раскрывает эти возможности не в полной мере; дело в том, что прообраз этого модуля из Turbo Pascal был расчитан на стандартный текстовый режим так называемых IBM-совместимых компьютеров, где всё было сравнительно просто: каждому знакоместу соответствовали две однобайтовые ячейки ***видеопамяти***, в первом байте располагался код символа, во втором — код цвета, причём четыре бита этого байта задавали цвет текста, три бита — цвет фона, что делало возможным использование всего восьми разных цветов для фона и шестнадцати — для текста; ещё один бит, если его взвести, заставлял символ мигать. Даже возможности алфавитно-цифровых терминалов, выпускавшихся в те времена, были шире, не говоря о современных программах-эмуляторах.

Поскольку модуль `crt`, представленный в Free Pascal, разрабатывался в первую очередь ради поддержки совместимости с его прообразом, его интерфейс повторяет особенности интерфейса прообраза и не предоставляет никаких более широких возможностей. В принципе, такие возможности можно было бы задействовать, используя модуль `video`, но работать с ним существенно сложнее, а задачи перед нами пока стоят исключительно учебные; пожалуй, если вы всерьёз хотите писать полноэкранные программы для алфавитно-цифрового терминала, будет правильнее изучить язык Си и воспользоваться библиотекой `ncurses`; впрочем, как вы вскоре убедитесь, возможностей модуля `crt` вполне достаточно для создания довольно интересных эффектов; в то же время освоить его гораздо проще.

Таблица 2.1. Константы для обозначения цвета в модуле `crt`

для текста и фона			только для текста	
`Black`	чёрный		`DarkGray`	тёмно-серый
`Blue`	синий		`LightBlue`	светло-синий
`Green`	зелёный		`LightGreen`	светло-зелёный
`Cyan`	голубой		`LightCyan`	светло-голубой
`Red`	красный		`LightRed`	светло-красный
`Magenta`	фиолетовый		`LightMagenta`	розовый
`Brown`	коричневый		`Yellow`	жёлтый
`LightGray`	светло-серый		`White`	белый

Основных средств у нас здесь всего два: процедура `TextColor` устанавливает цвет текста, а процедура `TextBackground` — цвет фона. Сами цвета задаются *константами*, описанными в модуле `crt`; они перечис-

лены в табл. 2.1. Следует обратить внимание, что для задания цвета текста можно использовать все 16 констант, перечисленных в таблице, тогда как для задания цвета фона можно использовать только восемь констант из левой колонки. Например, если выполнить

```
TextColor(Yellow);
TextBackground(Blue);
write('Hello');
```

то слово «Hello» будет выведено жёлтыми буквами на синем фоне, и такая комбинация будет использоваться для всего выводимого текста, пока вы снова не измените цвет текста или фона. Кроме того, вы можете заставить выводимый текст мигать; для этого при вызове TextBackground нужно добавить к её аргументу константу Blink; можно сделать это с помощью обычного сложения, хотя правильнее было бы использовать операцию *побитового «или»*. Например, текст, выводимый на экран после выполнения TextColor(Blue or Blink), будет синего цвета и мигающим.

К сожалению, необходимо отметить один фундаментальный недостаток описанных средств. **Установки цвета текста и фона сохраняют своё действие после завершения вашей программы**, при этом в модуле crt не предусмотрено средств, позволяющих узнать, какой цвет текста и фона установлен сейчас (в частности, на момент запуска вашей программы). Как следствие, восстановить цвета по умолчанию мы не можем. Как правило, после завершения нашей программы окошко терминала придётся закрыть или по крайней мере «привести в чувство» командой reset.

Следующая программа демонстрирует все возможные комбинации цветов текста и фона, заполняя экран звёздочками, причём каждая вторая звёздочка мигает. Для выхода из программы следует нажать любую клавишу. За неимением лучшего программа перед завершением устанавливает чёрный фон и тёмно-серый цвет текста; конечно, это может не соответствовать исходным настройкам терминала.

```
{ colordemo.pas }
program ColorsDemo;
uses crt;
const
    ColorCount = 16;
    BGColCount = 8;
var
    AllColors: array [1..ColorCount] of word =
    (
        Black, Blue, Green, Cyan,
        Red, Magenta, Brown, LightGray,
        DarkGray, LightBlue, LightGreen, LightCyan,
        LightRed, LightMagenta, Yellow, White
    );
procedure MakeLine(line, fgcolor: integer);
var
```

```
    i, j, w: integer;
begin
    w := ScreenWidth div BGColCount;
    for i := 1 to BGColCount do
    begin
        GotoXY((i-1)*w + 1, line);
        TextBackground(AllColors[i]);
        for j := 1 to w do
        begin
            if j mod 2 = 0 then
                TextColor(fgcolor + blink)
            else
                TextColor(fgcolor);
            write('*')
        end
    end
end;
procedure MakeScreen;
var
    i: integer;
begin
    clrscr;
    for i := 1 to ScreenHeight do
        MakeLine(i, AllColors[i mod ColorCount + 1]);
end;
begin
    MakeScreen;
    ReadKey;
    TextBackground(Black);
    TextColor(LightGray);
    clrscr
end.
```

2.12. Файлы

2.12.1. Общие сведения

Мы уже успели немного поработать с файлами через стандартные потоки ввода и вывода, полагаясь на то, что нужный файл нам «подсунет» пользователь при запуске нашей программы, перенаправив ввод или вывод средствами командного интерпретатора. Конечно, программа и сама может работать с файлами, лишь бы ей хватило на это полномочий.

Для того, чтобы работать с содержимым файла, его нужно *открыть*. При этом программа обращается к операционной системе, заявляя о намерении начать работу с файлом; обычно при этом указывается, какой файл интересует нашу программу (т. е. задаётся имя файла) и что программа собирается с ним делать (режим работы с файлом — только для чтения, только для записи, для чтения и записи, для добавления в конец). После того как файл успешно открыт, нашей программе

наряду с уже знакомыми нам *стандартными потоками ввода-вывода* становится доступен новый поток ввода или вывода, связанный с файлом на диске, указанным при его открытии. Операции, которые можно проделывать с таким потоком, в целом схожи (а на уровне операционной системы — попросту совпадают) с теми, которые мы можем выполнять со стандартными потоками: в основном это, конечно, уже известные нам чтение и запись, хотя есть и другие.

Потоки ввода-вывода, связанные с вновь открывающимися файлами, нужно как-то отличать друг от друга и от стандартных потоков. Язык Паскаль для этого предусматривает так называемые ***файловые переменные***; для описания таких переменных имеется целое семейство особых ***файловых типов***. Следует отметить, что файловый тип существенно отличается от других типов; самое заметное отличие состоит в том, что **переменные файлового типа представляют собой единственный вариант выражения файлового типа**, то есть значения файлового типа существуют только в виде «чего-то, что как-то там хранится в файловых переменных», и более никак; их даже нельзя присваивать. **Передавать файловые переменные в подпрограммы можно только через `var`-параметры**. Это может показаться непривычным, ведь раньше мы всегда говорили о *значениях* заданного типа и *выражениях*, вычисление которых даёт такие значения, а переменные того же типа рассматривали просто как хранилище соответствующего значения. С файловыми типами всё наоборот: у нас есть только файловые переменные; мы можем догадываться, что в них что-то хранится, и даже говорить, что, наверное, в них хранится «значение файлового типа», но все эти разговоры будут не более чем отвлечённой философией, поскольку никаких средств работы с такими значениями в отрыве от хранящих их переменных в Паскале нет и не предвидится.

Иначе говоря, переменные файловых типов позволяют нам различать между собой любое количество одновременно активных (открытых) потоков ввода-вывода, но и только: больше мы ни для чего файловые переменные использовать не можем.

В зависимости от того, как мы собираемся работать с файлом, мы должны выбрать конкретный тип файловой переменной. Здесь у нас есть три возможности:

- работать с файлом, предполагая, что он текстовый; для этого используется файловая переменная типа `text`;
- работать с файлом как с абстрактной последовательностью байтов, имея возможность записать и прочитать любой его фрагмент с помощью так называемых ***операций блочного чтения и блочной записи***; при этом потребуется файловая переменная, тип которой так и называется `file`;

- предположить, что файл состоит из блоков информации фиксированной длины, которые соответствуют машинному представлению в памяти значений какого-то типа; здесь нам потребуется так называемый ***типизированный файл***, для которого Паскаль поддерживает целое семейство типов, вводимых пользователем, например `file of integer`, `file of real` или (чаще) `file of myrecord`, где `myrecord` — имя описанного ранее типа-записи.

Отметим, что с одним и тем же файлом мы можем при желании работать по меньшей мере двумя, а часто и всеми тремя из перечисленных способов; избираемый нами способ работы зависит больше не от файла, а от решаемой задачи.

Вне зависимости от того, какого типа мы используем файловую переменную, перед началом работы с файлом необходимо назначить этой переменной какое-то имя файла; это делается вызовом процедуры `assign`. Например, если мы хотим работать с текстовым файлом `data.txt`, находящимся в текущей директории, нам в секции описания переменных придётся написать что-то вроде

```
var
    f1: text;
```

а где-то в программе — вызов

```
    assign(f1, 'data.txt');
```

Подчеркнём, что такой вызов просто связывает имя `'data.txt'` с файловой переменной. Процедура `assign` не пытается ни открыть файл, ни даже проверить, существует ли он (что и понятно, ведь, возможно, мы как раз собираемся создать новый файл). В качестве имени файла можно использовать, разумеется, не только константы, но и любые выражения типа `string`; если имя начинается с символа «/», то оно рассматривается как ***абсолютное имя файла*** и будет отсчитываться от корневого каталога нашей системы, если же имя начинается с любого другого символа, оно считается ***относительным*** и отсчитывается от текущей директории; впрочем, это уже обусловлено не Паскалем, а операционными системами семейства Unix.

Поскольку у начинающих часто возникает путаница между именем файла и именем файловой переменной, подчеркнём ещё раз, что это две совершенно разные, изначально никак не связанные между собой сущности. ***Имя файла*** — это то, как он (то есть файл) называется на диске, под каким именем его знает операционная система. Когда мы пишем программу, мы можем вообще не знать, каково будет имя файла во время её работы: возможно, имя файла нам укажет пользователь, или мы получим его ещё из каких-то источников, возможно даже, что

мы его прочитаем из другого файла; в реальных задачах такое часто случается.

С другой стороны, *имя файловой переменной* — это имя переменной и ничего более. Мы вольны называть свои переменные как нам вздумается; если мы переименуем переменные, но ничего больше в программе не изменим, то и поведение нашей программы не изменится, поскольку имена переменных на это поведение никак не влияют[36]. И, конечно же, выбранное нами имя файловой переменной никак не связано с тем, какой файл из хранящихся на нашем диске мы будем использовать.

Связь между именем файловой переменной и именем файла на диске начинает существовать только после вызова процедуры `assign`; больше того, никто не запрещает вызвать эту процедуру снова, разрушив старую связь и установив новую.

После того, как файловой переменной назначено имя файла, мы можем попытаться открыть файл для дальнейшей работы с ним. Если мы собираемся читать информацию из файла, то открывать его следует с помощью процедуры `reset`; в этом случае файл должен уже существовать (если его не существует, произойдёт ошибка), и работа с ним начнётся с его начальной позиции, то есть первая операция чтения извлечёт данные из самого начала файла, следующая операция извлечёт следующую порцию данных и т. д. Альтернативный вариант открытия файла — с помощью процедуры `rewrite`. В этом случае файл не обязан существовать: если его нет, он будет создан, если же он уже есть, то вся информация в нём будет уничтожена, работа начнётся «с чистого листа». Текстовые файлы, кроме того, можно открыть *на добавление*, это делается с помощью процедуры `append`; для типизированных и блочных файлов эта процедура не работает.

Нужно учесть, что операция открытия файла всегда чревата ошибками, а встроенная диагностика, которую умеет вставлять в наши программы компилятор Free Pascal, отличается феерической невнятностью; поэтому крайне желательно отключить встроенную обработку ошибок ввода-вывода, указав в программе уже знакомую нам директиву `{$I-}`, и организовать обработку ошибок самостоятельно, используя значение переменной `IOResult` (см. стр. 276).

Для чтения и записи текстовых и типизированных файлов используются уже знакомые нам операторы `read` и `write`, причём для текстовых (но не для типизированных) файлов можно использовать также и `readln`, и `writeln`; отличие здесь лишь в том, что, работая с файлами, а не со стандартными потоками ввода-вывода, первым аргументом в этих операторах мы указываем файловую переменную. Например, если

[36] Отметим, что принципиальное отсутствие влияния избранных программистом конкретных имён переменных на поведение программы — одна из ключевых особенностей *компилируемых* языков программирования, к которым относится, в числе прочих, Паскаль.

у нас есть файловая переменная `f1` и переменная `x` типа `integer`, мы можем написать что-то вроде

```
write(f1, x);
```

причём если `f1` имеет тип `text`, то в соответствующий файл будет записано текстовое представление числа, хранящегося в `x` (то есть последовательность байтов с кодами символов-цифр), тогда как если `f1` — типизированный файл, то записано будет ровно два байта — машинное представление числа типа `integer`. Точно так же применяются знакомые нам функции `eof` и `SeekEof` (последняя — только для текстовых файлов): при работе с файлами эти процедуры принимают один аргумент — файловую переменную, так что мы можем написать что-нибудь вроде «`while not eof(f1) do`».

Для работы с блочными файлами `read` и `write` не годятся, вместо них используются процедуры `BlockRead` и `BlockWrite`, которые мы рассмотрим позже в параграфе, посвящённом этому типу файлов.

Когда работа с файлом окончена, его следует закрыть вызовом процедуры `close`. Файловую переменную можно после этого использовать дальше для работы с другим (или даже тем же самым) файлом; если после закрытия файла сделать `reset` или `rewrite` с той же самой файловой переменной, будет открыт файл с тем же самым именем[37], но можно переназначить имя, повторно вызвав `assign`.

В заключение вводного параграфа приведём текст программы, записывающей всё ту же фразу `Hello, world!` в текстовый файл `hello.txt`:

```
{ hellofile.pas }
program HelloFile;
const
    message = 'Hello, world!';
    filename = 'hello.txt';
var
    f: text;
begin
    assign(f, filename);
    rewrite(f);
    writeln(f, message);
    close(f)
end.
```

После запуска такой программы в текущем каталоге появится файл `hello.txt` размером 14 байт (13 символов сообщения и перевод строки), который можно просмотреть, например, с помощью команды `cat`:

[37] Было бы ошибкой заявить, что будет открыт *тот же самый* файл: за то время, которое проходит между выполнением `close` и `reset/rewrite`, кто-то другой мог переименовать или удалить наш файл, а под его именем записать на диск совсем другой.

```
avst@host:~/work$ ./hellofile
avst@host:~/work$ ls -l hello.txt
-rw-r--r-- 1 avst avst 14 2015-07-18 18:50 hello.txt
avst@host:~/work$ cat hello.txt
Hello, world!
avst@host:~/work$
```

Вообще-то правильнее, конечно, было бы сделать чуть-чуть аккуратнее:

```
program HelloFile;
const
    message = 'Hello, world!';
    filename = 'hello.txt';
var
    f: text;
begin
    {$I-}
    assign(f, filename);
    rewrite(f);
    if IOResult <> 0 then
    begin
        writeln('Couldn''t open file ', filename);
        halt(1)
    end;
    writeln(f, message);
    if IOResult <> 0 then
    begin
        writeln('Couldn''t write to the file');
        halt(1)
    end;
    close(f)
end.
```

причём первое сообщение об ошибке гораздо важнее, чем второе: файлы очень часто не открываются по совершенно не зависящим от нас причинам, тогда как если файл открылся, то в большинстве случаев запись в него пройдёт успешно (хотя, конечно, не всегда: например, на диске может кончиться место).

Открытый файл, если это простой файл на диске, характеризуется *текущей позицией*, которая обычно при открытии устанавливается на начало файла, а при открытии текстового файла с помощью процедуры `append` — на его конец. Каждая операция ввода или вывода сдвигает текущую позицию вперёд на столько байтов, сколько было введено или выведено. Поэтому, например, последовательные операции чтения из одного и того же файла прочитают не одни и те же данные, а последовательно порции данных, находящиеся в файле, одну за другой. В некоторых случаях текущую позицию открытого файла можно изменить.

2.12.2. Текстовые файлы

Как следует из названия, ***текстовый файл*** — это файл, содержащий в себе текст, или, точнее, *данные в текстовом формате*[38]. Можно сказать, что данные к таком файле представляют собой некий *поток символов*, иногда разделяемый переводами строки; именно в таком формате мы читаем данные из стандартного потока ввода и выдаём их в стандартный поток вывода.

Для работы с текстовыми файлами, как уже говорилось, применяются файловые переменные встроенного типа `text`; только для таких потоков ввода-вывода определено понятие *строки*, что делает осмысленным применение операторов `writeln` и `readln`, а также функции `eoln` (*end of line*), которая возвращает истину, если в текущей позиции в данном файле находится конец строки. Остальные типы файлов не состоят из строк, так что ни вывод с переводом строки, ни ввод до конца строки, ни собственно конец строки для них не имеют смысла.

Важно понимать, что, если речь идёт не о символах и строках, а о данных других типов — например, о числах — то операция вывода в текстовый файл подразумевает перевод из машинного представления в текстовое, а операция ввода — перевод из текстового представления в машинное. Если это не совсем понятно, перечитайте § 1.6.6, причём прямо сейчас; если сложности не исчезли, обратитесь к кому-нибудь, кто сможет вам всё объяснить. Различие между текстовым и машинным представлением (в частности, для чисел) должно быть для вас совершенно очевидным; если это не так, дальше идти бессмысленно.

В частности, в § 1.6.6 мы рассматривали файлы, содержащие сто целых чисел от 1000 до 100099, каждое на 1001 больше предыдущего, причём файл `numbers.txt` содержал эти числа в текстовом представлении, а файл `numbers.bin` — в машинном, которое мы на тот момент называли *бинарным*. Программа, создающая первый из этих файлов, могла бы выглядеть так:

```
{ gennumtx.pas }
program GenerateNumTxt;
const
    name = 'numbers.txt';
    start = 1000;
    step = 1001;
    count = 100;
var
    f: text;
    i: integer;
    n: longint;
begin
    assign(f, name);
```

[38] См. §§ 1.6.5 и 1.6.6.

```
    rewrite(f);
    n := start;
    for i := 1 to count do
    begin
        writeln(f, n);
        n := n + step
    end;
    close(f)
end.
```

Текстовые файлы не допускают принудительного изменения текущей позиции и не предполагают попеременных операций чтения и записи; такие файлы записывают целиком, от начала к концу, иногда в несколько приёмов (в этом случае файл открывается на добавление с помощью `append`), но никогда не возвращаются при этом назад. Если потребовалось что-то изменить в начале или в середине существующего текстового файла, его перезаписывают весь целиком. Поэтому **для текстовых файлов процедура `reset` открывает файл в режиме «только чтение», а процедуры `rewrite` и `append` — в режиме «только запись»**.

Особенности текстового представления данных требуют дополнительной осторожности при выполнении чтения «до конца файла». Подробное обсуждение этого мы уже приводили в § 2.7.4 для случая стандартного потока ввода; при чтении из обычного текстового файла возникают аналогичные проблемы, решаемые с помощью той же самой функции `SeekEof`, только в этом случае она вызывается с одним параметром, в качестве которого ей передаётся файловая переменная. Напомним, что функция `SeekEof` фактически проверяет, есть ли ещё в потоке значащие (непробельные) символы; для этого она прочитывает и отбрасывает все пробельные символы, и если в процессе этого чтения/отбрасывания возникает ситуация «конец файла», функция возвращает «истину», если же найден значащий символ, этот символ возвращается обратно в поток (считается непрочитанным, чтобы его использовал последующий `read`), а сама функция при этом возвращает «ложь». Предусмотрена «файловая» версия также и для функции `SeekEoln`, которая аналогичным образом «ищет» конец строки, т. е. проверяет, можно ли из текущей строки прочитать ещё что-то значимое.

Пусть, к примеру, у нас имеется текстовый файл, имя которого мы получим через аргумент командной строки; файл состоит из строк, в каждой из которых располагаются одно или несколько чисел с плавающей точкой. Числа, находящиеся в одной строке, нам нужно будет перемножить, а результаты этих умножений — сложить и вывести результат. Например, для файла, содержащего

```
2.0  3.0     5.0
0.5  12.0
```

результатом работы должно стать число 36.0. Соответствующую программу можно написать так:

```
{ multandadd.pas }
program MultAndAdd;
var
    sum, mult, n: real;
    f: text;
begin
    {$I-}
    if ParamCount < 1 then
    begin
        writeln('Please specify the file name');
        halt
    end;
    assign(f, ParamStr(1));
    reset(f);
    if IOResult <> 0 then
    begin
        writeln('Could not open ', ParamStr(1));
        halt
    end;
    sum := 0;
    while not SeekEof(f) do
    begin
        mul := 1;
        while not SeekEoln(f) do
        begin
            read(f, n);
            mul := mul * n
        end;
        readln(f);
        sum := sum + mul
    end;
    writeln(sum:7:5)
end.
```

Обратите внимание на readln(f) после цикла чтений/умножений. Он вставлен для того, чтобы изъять из потока ввода символ перевода строки; если этот оператор убрать, программа просто «зависнет».

Ясно, что функции SeekEof и SeekEoln можно применять только для текстовых файлов; для любых других форматов данных такие функции просто *не имеют смысла*, ведь и разделение данных пробелами, и разнесение данных на разные строки — это, очевидно, явления, возможные только при работе с текстовым представлением.

Стоит отметить, что потоки стандартного ввода и стандартного вывода тоже имеют свои имена — для них Free Pascal предусматривает *глобальные переменные*, имеющие тип text. Поток стандартного вывода можно упомянуть по имени

output; например, writeln(output, 'Hello') — это то же самое, что и просто writeln('Hello'). Аналогично, поток стандартного ввода обозначается именем input, так что можно написать read(input, x) вместо просто read(x). Эти переменные могут быть удобны, например, если вы пишете какую-то подпрограмму, которая будет выдавать данные в текстовом виде, но не знаете заранее, нужно ли будет выдавать эти данные в текстовый файл или в стандартный поток; в этом случае можно предусмотреть параметр типа text, в качестве которого передавать или открытую файловую переменную, или output.

Под влиянием языка Си во Free Pascal вошли также другие названия глобальных переменных, обозначающих стандартные потоки: stdin (то же, что и input) и stdout (то же, что и output). Кроме того, Free Pascal позволяет осуществить вывод в стандартный поток выдачи сообщений об ошибках (см. § 1.4.9), также называемый потоком диагностики, который обозначается как ErrOutput или stderr.

2.12.3. Типизированные файлы

Под ***типизированным файлом*** в Паскале понимается файл, содержащий последовательность записей одинакового размера, соответствующих машинному представлению значений какого-то типа. Например, файл numbers.bin, который использовался в § 1.6.6, можно рассматривать как состоящий из 100 записей по четыре байта каждая, соответствующих машинному представлению четырёхбайтного целого (тип longint). Следующая программа создаст такой файл:

```
{ gennumbin.pas }
program GenerateNumBin;
const
    name = 'numbers.bin';
    start = 1000;
    step = 1001;
    count = 100;
var
    f: file of longint;
    i: integer;
    n: longint;
begin
    assign(f, name);
    rewrite(f);
    n := start;
    for i := 1 to count do
    begin
        write(f, n);
        n := n + step
    end;
    close(f)
end.
```

Если сравнить эту программу с программой `GenerateNumTxt` из предыдущего параграфа, можно обнаружить, что в тексте почти ничего не изменилось: поменялось название программы, суффикс в имени файла (`.bin` вместо `.txt`), используется оператор `write` вместо `writeln` и, наконец, самое главное: файловая переменная в предыдущей программе имела тип `text`, а в этой — `file of longint`.

В принципе, файл может состоять из записей, имеющих практически любой тип, нельзя только использовать файловые типы; настоятельно не рекомендуется (хотя и возможно) использовать указатели, которые мы будем рассматривать в следующей главе. Из данных любого другого типа файл состоять может; в частности, используя тип `file of char`, можно открыть в качестве типизированного текстовый файл, да и вообще любой, ведь любой файл состоит из байтов.

Очень часто в роли типа записи в типизированном файле используется, собственно, *запись* (`record`). Например, создавая какую-нибудь программу для работы с топографическими картами, мы могли бы задействовать файл, содержащий точки на местности, заданные широтой и долготой и снабжённые названиями. Для этого можно было бы описать такой тип:

```
type
    NamedPoint = record
        latitude, longitude: real;
        name: string[15];
    end;
```

и соответствующую файловую переменную:

```
var
    f: file of NamedPoint;
```

Для создания такого файла можно воспользоваться процедурой `rewrite`, для открытия существующего — процедурой `reset`. Операции открытия на добавление для типизированных файлов не предусмотрено.

В отличие от текстовых файлов, которые состоят из строк разного размера, записи типизированных файлов имеют фиксированный размер, что позволяет чередовать операции чтения и записи в любые места существующего файла. Изменить текущую позицию в открытом типизированном файле можно с помощью процедуры `seek`, которой нужно передать два параметра: файловую переменную и *номер записи* (при этом самая первая запись в файле имеет номер `0`). Например, следующие две строки:

```
    seek(f, 15);
    write(f, rec);
```

запишут запись `rec` в позицию № 15, вне зависимости от того, с какими позициями файла мы работали до этого. Это можно использовать для изменения отдельных записей уже существующего файла, что особенно важно для файлов значительного объёма, поскольку позволяет обойтись без их перезаписи. Пусть, к примеру, у нас есть файл, состоящий из записей `NamedPoint`, и нужно взять из этого файла запись с номером 705 и поменять её имя (то есть значение поля `name`) на строку `'Check12'`. Для этого мы можем прочитать эту запись в переменную типа `NamedPoint` (будем считать, что такая переменная у нас есть и называется `np`), изменить значение поля `name` и записать полученную запись на то же место:

```
seek(f, 705);
read(f, np);
np.name := 'Check12';
seek(f, 705);
write(f, np);
```

Обратите внимание, что перед записью нам пришлось опять применить `seek`; дело в том, что после операции чтения текущая позиция в открытом файле `f` стала соответствовать записи, следующей за прочитанной, то есть в данном случае — записи № 706, и пришлось это исправить.

Поскольку типизированные файлы позволяют чередовать операции чтения и записи, по умолчанию **процедуры, открывающие типизированный файл, открывают его в режиме «чтение и запись»**. Это касается и `reset`, и `rewrite`: различие между ними только в том, что `rewrite` создаёт новый файл, а если файл с таким именем уже есть, то ликвидирует его старое содержимое; `reset` не делает ни того, ни другого, а если файла с таким именем не было, выдаётся ошибка.

Такой подход может создать проблемы, например, при работе с файлом, который программа должна только прочитать, при этом для его записи у программы недостаточно полномочий. В такой ситуации попытка открыть файл для чтения и записи окончится неудачей, то есть как `reset`, так и `rewrite` выдаст ошибку. Решить проблему позволяет глобальная переменная, которая называется `filemode`; по умолчанию она содержит значение `2`, означающее, что типизированные файлы открываются на чтение и запись. Если в эту переменную записать число `0`, то файлы будут открываться в режиме «только чтение», что позволит (с помощью процедуры `reset`) успешно открыть файл, на запись которого у нас прав нет, но есть права на чтение; конечно, такой файл мы сможем только читать. Очень редко встречается ситуация, когда у нас есть права на запись в файл, но нет возможности чтения. В этом случае нужно занести в переменную `filemode` значение `1` и для открытия файла использовать `rewrite`.

2.12.4. Блочный ввод-вывод

Наряду с текстовыми и типизированными файлами Free Pascal поддерживает так называемые ***нетипизированные файлы***, позволяющие производить чтение и запись сразу большими порциями. В оригинальном Паскале такого средства не было, но именно такой вариант лучше всего соответствует возможностям, которые для работы с файлами предусмотрены в современных операционных системах.

Информацию, читаемую из нетипизированного файла, можно разместить в произвольной области памяти, то есть для этого подойдёт практически любая переменная; то же самое можно сказать и про запись в нетипизированный файл: информацию для такой записи можно взять из переменной произвольного типа. При открытии нетипизированного файла указывается ***размер блока*** в байтах, а при чтении и записи — количество блоков, которые нужно прочитать или записать. Чаще всего в качестве размера блока указывается число `1`, что позволяет производить чтение и запись совершенно произвольными фрагментами.

Файловая переменная для блочного ввода-вывода должна иметь тип `file` без указания типа элементов:

```
var
   f: file;
```

Как и для файлов других типов, для нетипизированных файлов назначается имя с помощью процедуры `assign`, а открываются они с помощью уже знакомых нам `reset` и `rewrite`, но у этих процедур при работе с нетипизированными файлами появляется второй параметр — целое число, означающее *размер блока*. Очень важно про этот параметр не забывать, поскольку «по умолчанию» (то есть если забыть указать второй параметр) размер блока будет принят равным 128 байтам, что обычно не соответствует нашим целям. Почему принято именно такое «умолчание», непонятно; как мы уже отмечали, чаще всего используется размер «блока» в один байт, как наиболее универсальный.

Точно так же, как и в случае типизированных файлов, обе процедуры `reset` и `rewrite` по умолчанию пытаются открыть файл в режиме «чтение и запись»; на это можно повлиять, изменяя значение глобальной переменной `filemode`, как это описано в предыдущем параграфе.

Для чтения из нетипизированных файлов и записи в них используются процедуры `BlockRead` и `BlockWrite`, которые очень похожи друг на друга: обе получают по четыре параметра, причём первым параметром указывается файловая переменная, вторым параметром — переменная произвольного типа и размера (за исключением файловых переменных), *в которую* будет помещена информация, прочитанная из файла, либо *из которой* будет взята информация для записи в файл (для `BlockRead` и `BlockWrite` соответственно). Третий параметр — целое число, задающее *количество блоков*, которые следует соответственно прочитать или

записать; естественно, произведение этого числа на используемый размер блока ни в коем случае не должно превосходить размер переменной, заданной вторым параметром. Наконец, в качестве четвёртого параметра процедурам передаётся *переменная* типа `longint`, `int64`, `word` или `integer`, и в эту переменную процедуры записывают количество блоков, которое им реально удалось прочитать или записать. Этот результат может в общем случае оказаться меньше того, что мы ожидали; чаще всего так происходит при чтении, когда в файле осталось меньше информации, чем мы пытаемся прочитать. Например:

```
const
    BufSize = 100;
var
    f: file;
    buf: array[1..BufSize] of char;
    res: integer;
begin
    { ... }
    BlockRead(f, buf, BufSize, res);
    { ... }
    BlockWrite(f, buf, BufSize, res);
```

Очень важен для нас один особый случай, который происходит только при использовании `BlockRead`: если в переменной, указанной последним параметром, после вызова функции оказалось значение `0`, это означает наступление ситуации «конец файла».

В принципе, четвёртый параметр можно не указывать, тогда любое несоответствие результата ожиданию вызовет ошибку. Делать так настоятельно не рекомендуется, в особенности при чтении: в самом деле, если в файле осталось недостаточно данных или если достигнут его конец, ничего ошибочного в этом нет, и вообще работа с использованием четвёртого параметра позволяет писать программы более гибко.

Для примера рассмотрим программу, которая копирует один файл в другой, получив их имена из параметров командной строки. Как для исходного, так и для целевого файлов (англ. *source* и *destination*) мы будем использовать нетипизированные файловые переменные; первый файл мы откроем в режиме «только чтение», второй — в режиме «только запись». Чтение мы будем производить фрагментами по `4096` байт (4 Kb), причём этот размер вынесем в константу; в качестве буфера, то есть переменной, в которую помещается прочитанная информация, воспользуемся массивом соответствующего размера из элементов типа `byte`.

Записывать в целевой файл мы будем на каждом шаге ровно столько информации, сколько было прочитано из исходного. Наконец, при возникновении ситуации «конец файла» мы немедленно завершим цикл чтения/записи, причём сделать это нам придётся *до выполнения записи*,

то есть из середины тела цикла; мы воспользуемся для этого оператором `break`, а сам цикл сделаем «бесконечным». После завершения цикла мы, естественно, должны будем закрыть оба файла. Поскольку мы уже знаем о существовании переменной `ErrOutput`, обозначающей поток сообщений об ошибках, все такие сообщения будем выдавать, как положено, именно в этот поток. Завершать программу после обнаружения ошибок будем с кодом 1, чтобы показать операционной системе, что у нас что-то пошло не так. Полностью программа будет выглядеть так:

```
{ block_cp.pas }
program BlockFileCopy;
const
    BufSize = 4096;
var
    src, dest: file;
    buffer: array [1..BufSize] of byte;
    ReadRes, WriteRes: longint;
begin
    {$I-}
    if ParamCount < 2 then
    begin
        writeln(ErrOutput, 'Expected: source and dest. names');
        halt(1)
    end;
    assign(src, ParamStr(1));
    assign(dest, ParamStr(2));
    filemode := 0;
    reset(src, 1);            { block size of 1 byte }
    if IOResult <> 0 then
    begin
        writeln(ErrOutput, 'Couldn''t open ', ParamStr(1));
        halt(1)
    end;
    filemode := 1;
    rewrite(dest, 1);
    if IOResult <> 0 then
    begin
        writeln(ErrOutput, 'Couldn''t open ', ParamStr(2));
        halt(1)
    end;
    while true do
    begin
        BlockRead(src, buffer, BufSize, ReadRes);
        if ReadRes = 0 then    { end of file! }
            break;
        BlockWrite(dest, buffer, ReadRes, WriteRes);
        if WriteRes <> ReadRes then
        begin
```

```
            writeln(ErrOutput, 'Error writing the file');
            break
        end
    end;
    close(src);
    close(dest)
end.
```

2.12.5. Операции над файлом как целым

Над файлом как единым объектом можно выполнить несколько операций, наиболее важные и востребованные из которых — это удаление и переименование. Для этого в Паскале предусмотрены процедуры `erase` и `rename`.

Процедура `erase` получает один параметр — файловую переменную произвольного типа, то есть здесь подойдёт и типизированный файл, и переменная типа `file` или `text`. К моменту вызова `erase` файловой переменной **должно быть назначено имя файла** с помощью `assign`, но файл при этом **не должен быть открыт**, то есть после выполнения `assign` мы должны либо вообще не вызывать `reset`, `rewrite` или `append`, либо, открыв файл и поработав с ним, закрыть его с помощью `close`. Например, следующая простая программа удаляет с диска файл, имя которого задано параметром командной строки:

```
{ erase_f.pas }
program EraseFile;
var
    f: file;
begin
    {$I-}
    if ParamCount < 1 then
    begin
        writeln(ErrOutput, 'Please specify the file to erase');
        halt(1)
    end;
    assign(f, ParamStr(1));
    erase(f);
    if IOResult <> 0 then
    begin
        writeln(ErrOutput, 'Error erasing the file');
        halt(1)
    end
end.
```

Процедура `rename` получает два параметра: файловую переменную и новое имя для файла (в виде строки). Как и в случае с `erase`, файловая переменная может иметь любой из файловых типов, а на момент вызова

`rename` ей должно быть назначено имя файла с помощью `assign`, но файл не должен быть открыт. Следующая программа получает два аргумента командной строки — старое и новое имена файла, и соответствующим образом переименовывает файл:

```
program RenameFile;
var
    f: file;
begin
    {$I-}
    if ParamCount < 2 then
    begin
        writeln(ErrOutput, 'Expected the old and new names');
        halt(1)
    end;
    assign(f, ParamStr(1));
    rename(f, ParamStr(2));
    if IOResult <> 0 then
    begin
        writeln(ErrOutput, 'Error renaming the file');
        halt(1)
    end
end.
```

2.13. Адреса, указатели и динамическая память

До сих пор мы использовали только переменные, задаваемые во время написания программы; поскольку переменная есть не более чем область памяти, это, в частности, означает, что именно во время написания программы мы полностью определяли, **сколько** памяти наша программа будет использовать во время исполнения.

Между тем, во время написания программы мы можем попросту не знать, сколько нам потребуется памяти. Учебные задачи часто специально формулируются так, чтобы проблема размера не возникала: например, нам могут сказать, что каких-нибудь чисел может быть сколько угодно, но не больше 10000, или что строки могут быть произвольной длины, но никак не длиннее 255 символов, или ещё что-нибудь в таком духе; на практике ограничения подобного рода могут оказаться неприемлемыми. Такова, например, задача, решаемая стандартной программой `sort`: эта программа прочитывает некий текст (из файла или из стандартного потока ввода), после чего выполняет сортировку и выдаёт строки прочитанного текста в отсортированном порядке. Программа не накладывает никаких ограничений ни на длину отдельной строки, ни на их общее количество; лишь бы хватило памяти. Стоит отметить, что

памяти в разных системах может быть разное количество; программа `sort` написана без каких-либо предположений о том, сколько памяти ей будет доступно, и пытается занять столько, сколько ей необходимо для сортировки конкретного текста. Если для поданного ей конкретного текста не хватило памяти в конкретной системе, программа, конечно, не сработает; но если памяти хватит, она успешно выполнит свою задачу, причём лишней памяти в ходе выполнения не займёт.

Освоение адресов и указателей как раз и позволит нам писать такие программы, которые сами *во время исполнения* определяют, сколько памяти нужно использовать. Такая возможность появляется благодаря ***динамическим переменным***, которые, в отличие от обычных переменных, не описываются в тексте программы, а создаются во время её выполнения. При этом было бы неправильно полагать, что указатели нужны только для этого; их область применения гораздо шире.

Возможно, эта глава покажется вам самой сложной; к сожалению, может получиться и так, что вы вообще не поймёте, о чём здесь идёт речь. Указатели не входят в школьную программу информатики, причём, судя по всему, причина этого не в том, что школьники не смогут их освоить, а в том, что указатели превышают возможности большинства учителей.

Между тем, позднее нам предстоит изучение языка Си, в котором, в отличие от Паскаля, практически ничего нельзя сделать, не понимая концепцию адресов и указателей. Паскаль в этом плане более лоялен к ученикам, он позволяет писать довольно сложные программы, не прибегая к указателям. Собственно говоря, мы как раз и выбрали Паскаль в качестве первого учебного языка, чтобы сначала научиться программировать, а потом, когда настанет подходящий момент, изучить заодно и указатели — и тем самым подготовиться к премудростям Си. Увы, без материала этой главы выбор Паскаля теряет всякий смысл, с тем же успехом можно было бы изучать любой другой язык. Ранее мы советовали пропустить параграф или даже целую главу, если она окажется слишком сложной, и вернуться к ней позже; для главы, посвящённой указателям, мы этого порекомендовать не можем.

Если материал, изложенный в этой главе, окажется вам не по силам, можно посоветовать либо найти кого-то, кто сможет вам помочь разобраться, либо вообще отложить на какое-то время освоение новых инструментов и попытаться писать программы с помощью всего того, что вы уже освоили. Возможно, спустя несколько месяцев и полдюжины написанных и отлаженных программ глава про указатели не покажется вам столь сложной.

Кроме того, вы можете взять следующий том, попробовать прочитать третью часть нашей книги, которая посвящена программированию на уровне машинных команд с помощью языка ассемблера, и после этого

возобновить попытки освоения указателей. Чего точно не стоит делать — это пытаться, так и не поняв указатели, изучать Си; всё равно не получится.

2.13.1. Что такое указатель

Напомним для начала несколько базовых утверждений. Во-первых, **память компьютера состоит из одинаковых ячеек, каждая из которых имеет свой *адрес***. В большинстве случаев адрес — это просто целое число, но не всегда; на некоторых архитектурах адрес может состоять из двух чисел, и теоретически можно себе представить адреса, имеющие ещё более сложную структуру. Что можно уверенно сказать, так это то, что **адрес есть некая информация, позволяющая однозначно идентифицировать ячейку памяти**.

Во-вторых, мы часто пользуемся не отдельными ячейками памяти, а некими наборами ячеек, идущих подряд, которые мы называем *областями памяти*. В частности, **любая переменная есть область памяти** (переменные некоторых типов, таких как `char` или `byte`, занимают ровно одну ячейку, что можно считать частным случаем области). **Адресом области памяти считается адрес первой ячейки этой области**.

Как и другую информацию, **адреса можно хранить в памяти**, то есть одна область памяти может содержать (хранить) адрес другой области памяти. Многие языки программирования высокого уровня, включая Паскаль, предусматривают специальный тип или семейство типов выражений, которые соответствуют адресам; переменная такого типа, то есть переменная, в которой хранится адрес (например, адрес другой переменной) как раз и называется ***указателем***.

Всю премудрость указателей и адресов можно выразить всего двумя короткими фразами:

Указатель — это переменная,
в которой хранится адрес

Утверждение вида «A указывает на B»
означает «A содержит адрес B»

На практике довольно часто допускают терминологические вольности, путая понятия адреса и указателя; например, можно услышать, что некая функция «возвращает указатель», тогда как на самом деле она возвращает, естественно, не переменную, а значение, то есть адрес, а не указатель. Точно так же довольно часто про адрес (а не про указатель) говорят, что он на что-то указывает, хотя адрес, конечно же, не указывает на переменную, а является её свойством. Хотя формально утверждения такого вида не вполне верны, никакой путаницы они, как

ни странно, не порождают: всегда остаётся понятным, что имеется в виду.

2.13.2. Указатели в Паскале

В большинстве случаев в Паскале применяются так называемые ***типизированные указатели*** — переменные, хранящие адрес, относительно которого точно известно, *переменная какого типа* расположена в соответствующей области памяти. Для обозначения указателя в оригинальном виртовском описании Паскаля использовался символ «↑», но в ASCII (а равно и на вашей клавиатуре) такого символа нет, поэтому во всех реально существующих версиях Паскаля используется символ «^» (вы легко найдёте эту «крышку» на клавише «6»). Например, если описать две переменные

```
var
    p: ^integer;
    q: ^real;
```

то `p` сможет хранить адрес переменной, имеющей тип `integer`, а `q` сможет хранить адрес переменной, имеющей тип `real`.

Адрес переменной можно получить с помощью ***операции взятия адреса***, которая обозначается знаком «`@`»[39], то есть, например, выражение `@x` даёт *адрес переменной* ***x***. В частности, если описать переменную типа `real` и переменную типа *указатель на* ***real***:

```
var
    r: real;
    p: ^real;
```

то станет возможно занести адрес первой переменной во вторую:

```
    p := @r;
```

Указатели и вообще адресные выражения были бы совершенно бесполезны, если бы нельзя было *что-то сделать с областью памяти (т. е., в большинстве случаев, с переменной), зная только её адрес*. Для этого в Паскале используется операция *dereference*, название которой на русский язык часто переводится неказистым словом «*разыменование*», хотя можно было бы использовать, например, термин «обращение по адресу». Обозначается эта операция уже знакомым нам символом «^», который ставится *после* имени указателя (или, говоря в общем, после произвольного адресного выражения, в качестве которого может

[39] В оригинальной версии Паскаля такой операции не было, что, на наш взгляд, изрядно усложняет не только работу, но и объяснения; к счастью, в современных версиях Паскаля операция взятия адреса всегда присутствует.

выступать, например, вызов функции, возвращающей адрес). Так, после присваивания адреса из вышеприведённого примера выражение `p^` будет обозначать «то, на что указывает `p`», то есть в данном случае переменную `r`. В частности, оператор

```
p^ := 25.7;
```

занесёт в память, находящуюся по адресу, хранящемуся в `p` (то есть попросту в переменную `r`) значение `25.7`, а оператор

```
writeln(p^);
```

напечатает это значение.

Отметим ещё один важный момент. В Паскале предусмотрены также ***нетипизированные указатели*** (и адреса), для которых введён встроенный тип `pointer`. Адреса этого типа компилятор рассматривает просто как абстрактные адреса ячеек памяти, не делая никаких предположений о том, значения какого типа находятся в памяти по такому адресу; указатели этого типа, соответственно, способны хранить произвольный адрес в памяти вне зависимости от того, что́ там, по этому адресу, находится. Если мы опишем переменную типа `pointer`, например

```
var
    ap: pointer;
```

то в такую переменную можно будет занести адрес любого типа; мало того, *значение этой переменной можно будет присвоить переменной любого указательного типа*, что вообще-то чревато ошибками: к примеру, можно занести в `ap` адрес какой-нибудь переменной типа `string`, а потом забыть про это и присвоить её значение переменной типа `^integer`; если теперь попробовать с такой переменной работать, ничего хорошего не выйдет, ведь на самом деле по этому адресу находится не `integer`, а `string`. Следует поэтому соблюдать крайнюю осторожность при работе с нетипизированными указателями, а лучше не использовать их вовсе, пока это всерьёз не потребуется. Отметим, что такая потребность возникнет у вас не скоро, если вообще возникнет: реальная необходимость в применении нетипизированных указателей появляется при создании нетривиальных структур данных, какие могут понадобиться только в больших и сложных программах.

Мы могли бы вообще не упоминать нетипизированные указатели в нашей книге, если бы не то обстоятельство, что *результатом операции взятия адреса является как раз нетипизированный адрес*. Почему создатели Turbo Pascal, впервые внедрившие операцию взятия адреса в этот язык, поступили именно так — остаётся непонятным (например, в языке Си точно такая же операция прекрасно справляется с формированием типизированного адреса).

Есть и ещё один важный случай нетипизированного адресного выражения — встроенная константа `nil`. Она обозначает ***недействительный адрес***, то есть такой, по которому в памяти заведомо не может находиться ни одна переменная, и присваивается переменным указательных типов, чтобы показать, что *данный указатель сейчас никуда не указывает*. Константу `nil` иногда называют ***нулевым указателем***

Если попытаться выделить из этого параграфа «сухой остаток», мы получим следующее:

- если `t` — это некий тип, то `^t` — это тип «указатель на `t`»;
- если `x` — произвольная переменная, то выражение `@x` означает «адрес переменной `x`» (причём нетипизированный);
- если `p` — это указатель (или другое адресное выражение), то `p^` обозначает «то, на что указывает `p`»;
- словом `nil` обозначается специальный «нулевой адрес», используемый, чтобы показать, что данный указатель сейчас ни на что не указывает.

2.13.3. Динамические переменные

Как мы уже отмечали, под ***динамической переменной*** понимается такая переменная (т. е. область памяти), которая создаётся во время выполнения программы. Ясно, что у такой переменной не может быть имени, ведь все имена мы задаём, когда пишем программу. Работа с динамическими переменными производится с использованием их *адресов*, хранящихся в указателях.

Память под динамические переменные выделяется из специальной области памяти, которую называют *кучей* (англ. *heap*). При уничтожении динамической переменной память возвращается обратно в «кучу» и может быть выделена под другую динамическую переменную. Если в куче не хватает места, наша программа (незаметно для нас) обращается к операционной системе с просьбой выдать больше памяти, за счёт чего увеличивает размер кучи. Следует учитывать, что *уменьшить* размер кучи невозможно, то есть если наша программа уже затребовала и получила память от операционной системы, то эта память останется в распоряжении программы до её завершения. Из этого, например, следует, что если вам надо одномоментно создать какую-то динамическую переменную, а какую-то другую уничтожить, то лучше сначала уничтожить старую переменную, а потом создавать новую. Иногда это позволяет сэкономить на общем размере кучи.

Создание динамической переменной производится с помощью псевдопроцедуры `new`, которая должна быть применена к типизированному указателю. При этом происходят две вещи: во-первых, из кучи выделяется область памяти нужного размера, в которой будет теперь размещаться свежесозданная динамическая переменная (собственно,

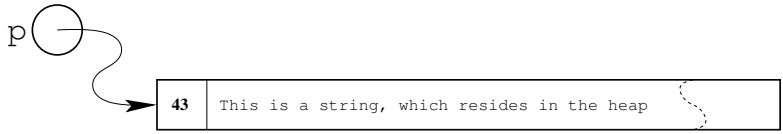

Рис. 2.4. Указатель на строку в динамической памяти

эта динамическая переменная и есть та область памяти, которую только что выделили); во-вторых, в заданный указатель заносится адрес созданной динамической переменной.

Например, если мы описали указатель

```
var
    p: ^string;
```

то теперь мы можем создать динамическую переменную типа `string`; делается это с помощью `new`:

```
    new(p);
```

При выполнении этого `new`, во-первых, из кучи будет выделена область памяти размером в 256 байт, которая как раз и станет нашей новой (динамической) переменной типа `string`; во-вторых, в переменную `p` будет занесён адрес этой области памяти. Таким образом, у нас теперь есть *безымянная* переменная типа `string`, единственный способ доступа к которой — через её адрес: выражение `p^` как раз соответствует этой переменной. Мы можем, к примеру, занести в эту переменную значение:

```
    p^ := 'This is a string, which resides in the heap';
```

Полученная в памяти программы структура может быть схематически представлена как показано на рис. 2.4.

Удаление динамической переменной, которая нам больше не нужна, производится с помощью псевдопроцедуры `dispose`; параметром ей служит адрес переменной, которую нужно ликвидировать:

```
    dispose(p);
```

При этом, как ни странно, *значение указателя* `p` *не меняется*; единственное, что происходит — это возвращение памяти, которая была занята под переменную `p^`, обратно в кучу; иначе говоря, меняется статус этой области памяти, вместо занятой она помечается как свободная и доступная к выдаче по требованию (одним из следующих `new`). Конечно, использовать значение указателя `p` после этого нельзя, ведь мы сами сообщили менеджеру кучи, что более не собираемся работать с переменной `p^`.

Важно понимать, что **динамическая переменная не привязана к конкретному указателю**. Например, если у нас есть два указателя:

```
var
    p, q: ^string;
```

то мы можем выделить память, используя один из них:

```
    new(p);
```

потом в какой-то момент занести этот адрес в другой указатель:

```
    q := p;
```

и работать с полученной переменной, обозначая её как с `q^` вместо `p^`; в самом деле, ведь адрес нашей переменной теперь есть в обоих указателях. Более того, мы можем занять указатель `p` под что-то другое, а работу с ранее выделенной переменной проводить только через `q`, и, когда придёт время, удалить эту переменную, используя, опять же, указатель `q`:

```
    dispose(q);
```

Важен здесь сам адрес (т. е. *значение адреса*), а не то, в каком из указателей этот адрес нынче лежит.

Ещё один крайне важный момент состоит в том, что при определённой небрежности можно легко потерять адрес динамической переменной. Например, если мы создадим переменную с помощью `new(p)`, поработаем с ней, а потом, не удалив эту переменную, снова выполним `new(p)`, то менеджер кучи выделит нам новую переменную и занесёт её адрес в указатель `p`; как всегда в таких случаях, старое значение переменной `p` будет потеряно, но ведь там хранился адрес первой выделенной динамической переменной, а к ней у нас нет больше никаких способов доступа!

Динамическая переменная, которую забыли освободить, но на которую больше не указывает ни один указатель, становится так называемым ***мусором*** (англ. *garbage*; не следует путать этот термин со словом *junk*, которое тоже переводится как *мусор*, но в программировании обычно обозначает бессмысленные данные, а не потерянную память). В некоторых языках программирования предусмотрена так называемая *сборка мусора*, которая обеспечивает автоматическое обнаружение таких переменных и их возвращение в кучу, но в Паскале такого нет, и это, в целом, даже хорошо. Дело в том, что механизмы сборки мусора довольно сложны и зачастую срабатывают в самый неподходящий момент, «подвешивая» на некоторое время выполнение нашей программы; например, на соревнованиях автомобилей-роботов DARPA Grand Challenge в 2005 году одна из машин, находившаяся под управлением ОС Linux и программ на языке Java, влетела на скорости около 100 км/ч в бетонную стену, причём в качестве одной из возможных причин специалисты назвали не вовремя включившийся сборщик мусора.

Так или иначе, в Паскале сборка мусора не предусмотрена, так что нам необходимо внимательно следить за удалением ненужных динамических переменных; в противном случае мы можем очень быстро исчерпать доступную память.

Кстати, процесс образования мусора программисты называют ***утечкой памяти*** (англ. *memory leaks*). Отметим, что утечка памяти свидетельствует только об одном: о безалаберности и невнимательности автора программы. Никаких оправданий утечке памяти быть не может, и если вам кто-то попытается заявить противоположное, не верьте: такой человек просто не умеет программировать.

От материала этого параграфа может остаться некоторое недоумение. В самом деле, зачем описывать указатель (например, на строку, как в нашем примере), а потом делать какой-то там `new`, если можно сразу описать переменную типа `string` и работать с ней как обычно?

Некоторый смысл работа с динамическими переменными получает, если эти переменные имеют сравнительно большой размер, например, в несколько сот килобайт (это может быть какой-нибудь массив записей, некоторые поля которых тоже являются массивами, и т. п.); описывать переменную такого размера в качестве обычной локальной попросту опасно, может не хватить стековой памяти, а вот с размещением её же в куче никаких проблем не возникнет; кроме того, если у вас много таких переменных, но не все они нужны одновременно, может оказаться полезным создавать и удалять их по мере необходимости. Однако все эти примеры, откровенно говоря, притянуты за уши; указатели почти никогда не используются таким образом. В полной мере возможности указателей раскрываются только при создании так называемых *связных динамических структур данных*, которые состоят из отдельных переменных типа «запись», причём каждая такая переменная содержит один или несколько указателей на другие переменные того же типа. Некоторые из таких структур данных мы рассмотрим в следующих параграфах.

2.13.4. Односвязные списки

Пожалуй, самой простой динамической структурой данных оказывается так называемый ***односвязный список***. Такой список строится из ***звеньев***, каждое из которых, представляя собой переменную типа «запись», в качестве одного из своих полей имеет *указатель на следующий элемент списка*; очевидно, что такое поле должно иметь тип «указатель на запись того же типа». Последний элемент списка содержит в этом поле значение `nil`, чтобы показать, что больше элементов нет. Достаточно при этом в каком-нибудь указателе хранить адрес самого первого элемента списка, и мы сможем при необходимости добраться до любого другого его элемента.

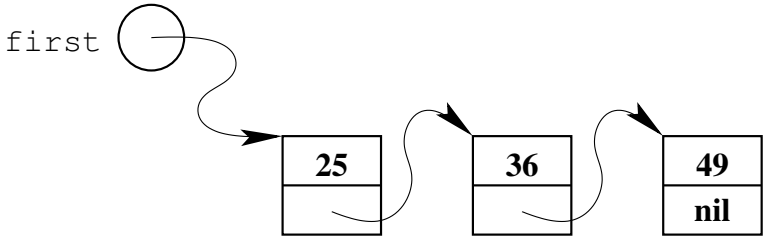

Рис. 2.5. Односвязный список из трёх элементов

Слово «односвязный» в названии этой структуры данных означает, что на каждый из его элементов указывает *один* указатель; впрочем, с тем же успехом можно отметить, что каждый элемент имеет *одно* поле, предназначенное для сохранения связности списка, и заявить, что «односвязность» предполагает именно это. Наконец, можно сказать, что в списке выстроена *одна* цепочка связей из указателей и списать загадочную «односвязность» на этот факт. Как мы увидим позже, все эти утверждения эквивалентны.

Пример односвязного списка из трёх элементов, содержащих целые числа 25, 36 и 49, показан на рис. 2.5. Для создания такого списка нам потребуется запись, состоящая из двух полей: первое будет хранить целое число, второе — адрес следующего звена списка. Здесь в Паскале имеется некая хитрость: использовать имя описываемого типа при описании его собственных полей почему-то нельзя, то есть мы не можем написать вот так:

```
type
    item = record
        data: integer;
        next: ^item;
    end;
```

При этом Паскаль позволяет описать указательный тип, используя такое имя типа, которое пока ещё не введено; как следствие, мы можем дать отдельное имя типу указателя на `item` до того, как опишем сам тип `item`, а при описании `item` воспользоваться этим именем, например:

```
type
    itemptr = ^item;
    item = record
        data: integer;
        next: itemptr;
    end;
```

Такое описание никаких возражений у компилятора не вызовет. Имея типы `item` и `itemptr`, мы можем описать указатель для работы со списком:

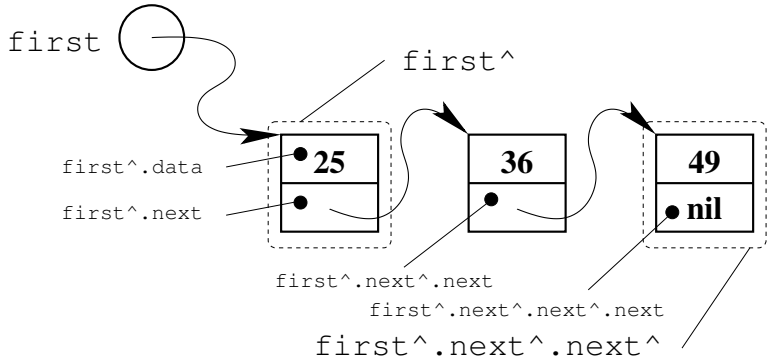

Рис. 2.6. Навигация в односвязном списке

```
var
    first: itemptr;
```

ну а сам список, показанный на рисунке, можно создать, например, так:

```
    new(first);
    first^.data := 25;
    new(first^.next);
    first^.next^.data := 36;
    new(first^.next^.next);
    first^.next^.next^.data := 49;
    first^.next^.next^.next := nil;
```

С непривычки конструкции вроде `first^.next^.data` могут напугать; это вполне нормальная реакция, но позволить себе долго бояться мы не можем, поэтому нам придётся тщательно разобраться, что тут происходит. Итак, коль скоро у нас указатель на первый элемент называется `first`, то выражение `first^` будет обозначать весь этот элемент целиком. Поскольку сам этот элемент представляет собой запись из двух полей, доступ к полям, как и для любой записи, производится через точку и имя поля; таким образом, `first^.data` — это поле первого элемента, в котором содержится целое число (на рисунке и в нашем примере это число 25), а `first^.next` — это указатель на следующий (второй) элемент списка. В свою очередь, `first^.next^` — это сам второй элемент списка, и так далее (см. рис. 2.6).

Если что-то здесь осталось непонятно, двигаться дальше нет никакого смысла! Сначала добейтесь понимания происходящего, в противном случае в дальнейшем тексте вы тем более ничего не поймёте.

Конечно, в большинстве случаев со списками работают совсем не так; пример, потребовавший устрашающего выражения

`first^.next^.next^.next`, мы привели скорее для иллюстрации. Гораздо чаще при работе с односвязным списком поступают одним из двух способов: либо заносят элементы всегда в его начало (это делается с помощью вспомогательного указателя), либо хранят два указателя — на начало и конец списка, а элементы заносят всегда в конец. Прежде чем показать, как это делается, мы предложим вашему вниманию две задачи, которые **настоятельно рекомендуется хотя бы попробовать решить самостоятельно**, причём до того, как вы прочитаете остаток этого параграфа. Дело в том, что, самостоятельно догадавшись, как это делается, вы уже никогда этого не забудете, и работа со списками никогда больше не вызовет у вас проблем; если же вы сразу начнёте с разбора примеров, которые мы приведём дальше, то понять, как всё это делать самостоятельно, окажется гораздо сложнее, потому что побороть искушение писать «по аналогии» (то есть без достижения полного понимания происходящего) часто не могут даже самые волевые люди.

Итак, вот первая задача; она потребует создания односвязного списка и добавления элементов в его начало:

> *Написать программу, которая читает целые числа из потока стандартного ввода до возникновения ситуации «конец файла», после чего печатает все введённые числа в обратном порядке. Количество чисел заранее неизвестно, вводить явные ограничения на это количество запрещается.*

Вторая задача также потребует использования односвязного списка, но добавлять новые элементы придётся в его конец, для чего нужно будет работать со списком через два указателя: первый будет хранить адрес первого элемента списка, второй — адрес последнего элемента.

> *Написать программу, которая читает целые числа из потока стандартного ввода до возникновения ситуации «конец файла», после чего* ***дважды*** *печатает все введённые числа в том порядке, в котором они были введены. Количество чисел заранее неизвестно, вводить явные ограничения на это количество запрещается.*

Учтите, что считать подобного рода задачи решёнными можно не раньше, чем у вас **на компьютере** заработает (причём заработает правильно) написанная вами программа, соответствующая условиям. Даже в этом случае задача не всегда оказывается верно решённой, поскольку вы можете упустить при её тестировании какие-то важные случаи или просто неверно истолковать получаемые результаты; но если работающей программы вообще нет, то о том, что задача решена, заведомо не может быть никакой речи.

В надежде, что вы как минимум попробовали решить предложенные задачи, мы продолжим обсуждение работы со списками. Прежде всего отметим один крайне важный момент. Если контекст решаемой задачи предполагает, что вы начинаете работу с пустого списка, то есть со списка, не содержащего ни одного элемента, **обязательно превратите ваш указатель в корректный пустой список, занеся в него значение** `nil`. Начинающие часто забывают про это, получая на выходе постоянные аварии.

Добавление элемента в начало односвязного списка производится в три этапа. Сначала, используя вспомогательный указатель, мы создаём (в динамической памяти) новый элемент списка. Затем мы заполняем этот элемент; в частности, мы делаем так, чтобы его указатель на следующий элемент стал указывать на тот элемент списка, который сейчас (пока) является первым, а после добавления нового станет вторым, т. е. как раз следующим после нового. Нужный адрес, как несложно догадаться, находится в указателе на первый элемент списка, его-то мы и присвоим полю `next` в новом элементе. Наконец, третий шаг — признать новый элемент новым первым элементом списка; это делается занесением его адреса в указатель, хранящий адрес первого элемента.

Схематически происходящее показано на рис. 2.7 на примере всё того же списка целых чисел, состоящего из элементов типа `item`, причём адрес первого элемента списка хранится в указателе `first`; сначала в этом списке находятся элементы, хранящие числа 36 и 49 (рис. 2.7, «**a)**»), и нам нужно занести в его начало новый элемент, содержащий число 25. Для этого мы вводим дополнительный указатель, который будет называться `tmp` от английского *temporary*, что означает «временный»:

```
var
    tmp: itemptr;
```

На первом шаге мы, как уже было сказано, создаём новый элемент:

```
    new(tmp);
```

Получившаяся после этого ситуация показана на рис. 2.7, «**b)**». Со списком ещё ничего не произошло, созданный новый элемент никак его не затрагивает. Сам элемент пока остаётся сильно «недоделанным»: в обоих его полях находится непонятный мусор, что показано на рисунке символом «**?!**». Пора делать второй шаг — заполнять поля нового элемента. В поле `data` нам нужно будет занести искомое число 25, тогда как поле `next` должно указывать на тот элемент, который (после включения нового элемента в список) станет следующим после нового; это, собственно, тот элемент, который сейчас пока в списке первый, то есть его адрес хранится в указателе `first`, его-то мы и присвоим полю `next`:

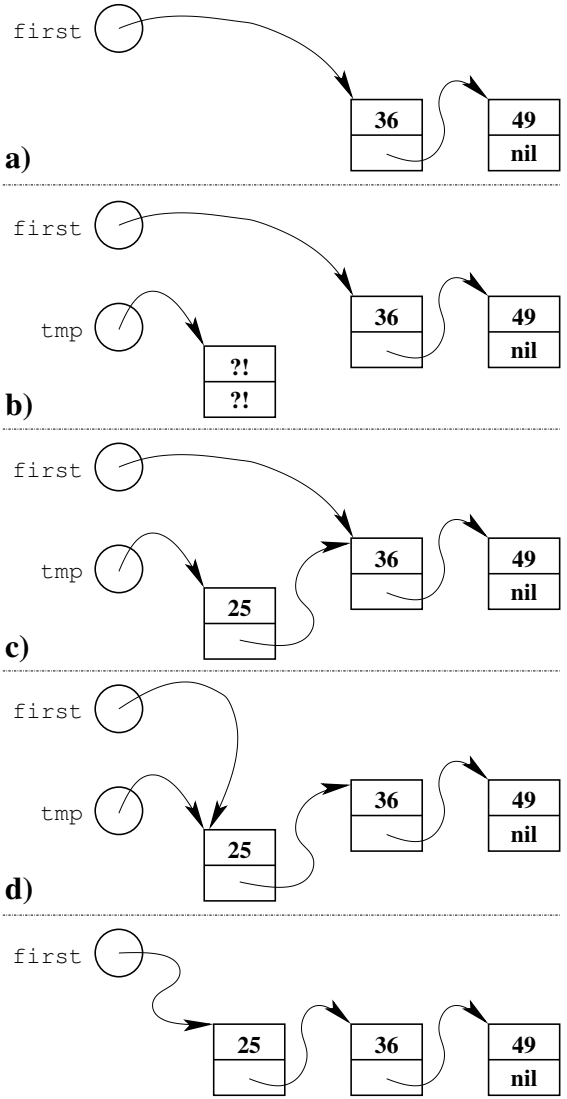

Рис. 2.7. Занесение нового элемента в начало односвязного списка

```
tmp^.data := 25;
tmp^.next := first;
```

Состояние нашей структуры данных после этих присваиваний показано на рис. 2.7, «**c**)». Осталось лишь финальным ударом объявить новый (и полностью подготовленный к своей роли) элемент первым элементом

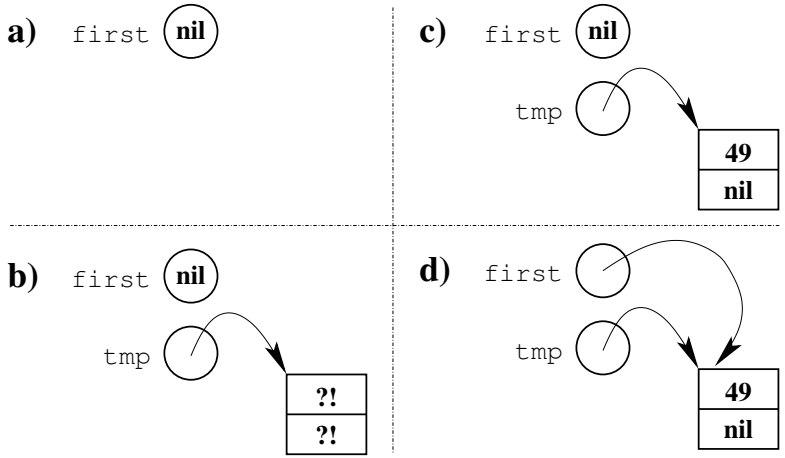

Рис. 2.8. Работа процедуры занесения в начало при исходно пустом списке

списка, присвоив его адрес указателю `first`; поскольку этот адрес хранится в `tmp`, всё оказывается совсем просто:

```
first := tmp;
```

Финальная ситуация показана на рис. 2.7, «**d)**». Забыв про наш временный указатель и про то, что первый элемент списка только что был «новым», мы получаем ровно то состояние списка, к которому стремились.

С этой трёхшаговой процедурой добавления нового элемента в начало связано одно замечательное свойство: **для случая пустого списка последовательность действий по добавлению первого элемента полностью совпадает с последовательностью действий по добавлению нового элемента в начало списка, уже содержащего элементы**. Соответствующая последовательность состояний структуры данных показана на рис. 2.8: вначале список пуст («**a)**»); на первом шаге мы создаём новый элемент («**b)**»); на втором шаге заполняем его («**c)**»); последним действием делаем этот новый элемент первым элементом списка («**d)**»). **Всё это работает только при условии, что мы не забыли перед началом работы со списком сделать его корректным, занеся значение `nil` в указатель `first`!**

Если всё время заносить элементы в начало списка, то располагаться в списке они будут в порядке, противоположном тому, в котором их туда заносили; именно это и требуется в первой из двух предложенных ранее задач. Решение будет состоять из двух циклов, в первом из них будет выполняться чтение чисел до тех пор, пока не настанет ситуация «конец

файла»[40], и после прочтения очередного числа оно будет добавляться в начало списка; после этого останется только пройти по списку из его начала до конца, печатая содержащиеся в нём числа. Программа целиком будет выглядеть так:

```
program Numbers1;
type
    itemptr = ^item;
    item = record
        data: integer;
        next: itemptr;
    end;
var
    first, tmp: itemptr;
    n: integer;
begin
    first := nil;    { делаем список корректно пустым! }
    while not SeekEof do           { цикл чтения чисел }
    begin
        read(n);
        new(tmp);                  { создали }
        tmp^.data := n;            { заполняем }
        tmp^.next := first;
        first := tmp               { включаем в список }
    end;
    tmp := first;        { проходим по списку с начала }
    while tmp <> nil do            { до конца }
    begin
        writeln(tmp^.data);
        tmp := tmp^.next  { переход к следующему эл-ту }
    end
end.
```

В этом тексте мы ни разу не использовали `dispose`; здесь высвобождение памяти не имеет большого смысла, поскольку сразу после первого (и единственного) использования построенного списка программа завершается, так что вся память, отведённая ей в системе, освобождается; разумеется, это касается в том числе и кучи. В более сложных программах, например, таких, которые строят один список за другим и так много раз, забывать про освобождение памяти ни в коем случае нельзя. Освободить память от односвязного списка можно с помощью цикла, в котором на каждом шаге из списка удаляется первый элемент, и так до тех пор, пока список не опустеет. При этом тоже есть своя хитрость. Если просто взять и сделать `dispose(first)`, первый элемент списка перестанет существовать, а значит, мы уже не сможем использовать

[40] Напомним, что при чтении чисел нужно для отслеживания ситуации «конец файла» использовать функцию `SeekEof`; мы подробно обсуждали это в § 2.7.4.

его поля, но ведь адрес второго элемента хранится именно там. Поэтому прежде чем уничтожать первый элемент, необходимо запомнить адрес следующего; после этого первый элемент уничтожается, а указателю `first` присваивается адрес следующего элемента, сохранённый во вспомогательном указателе. Соответствующий цикл выглядит так:

```
while first <> nil do
begin
    tmp := first^.next;      { запоминаем адрес следующего }
    dispose(first);            { уничтожаем первый элемент }
    first := tmp  { список теперь начинается со следующего }
end
```

Можно поступить и иначе: запомнив адрес *первого* элемента во вспомогательном указателе, исключить этот элемент из списка, соответствующим образом изменив `first`, и уже после этого удалить его:

```
while first <> nil do
begin
    tmp := first;          { запоминаем адрес первого }
    first := first^.next;   { исключаем его из списка }
    dispose(tmp)              { удаляем его из памяти }
end
```

На рис. 2.9 первый способ показан слева, второй — справа. Какой из них лучше, можете решить для себя сами; по эффективности между ними никакой разницы нет.

Как можно заметить, условие второй из предложенных раньше задач от условия первой отличается в основном порядком выдачи элементов на печать. В принципе, никто не мешает построить список, как и для первой задачи, «задом наперёд», а потом отдельным циклом «перевернуть» его (как это сделать, попробуйте придумать сами в качестве упражнения); но правильнее будет всё же сразу построить список в нужном порядке, для чего нам потребуется добавление элементов в конец списка, а не в его начало.

Если односвязный список предполагается увеличивать «с конца», обычно для этого хранят не один указатель, как в предшествующих примерах, а два: на первый элемент списка и на последний. Когда список пуст, оба указателя должны иметь значение `nil`. Процедура занесения нового элемента в такой список (в его конец) выглядит даже проще, чем только что рассмотренная процедура занесения в начало: если, к примеру, указатель на последний элемент называется `last`, то мы можем создать новый элемент сразу где надо, то есть занести его адрес в `last^.next`, затем сдвинуть указатель `last` на новоиспечённый последний элемент и уже после этого заполнить его поля; если нам нужно занести в список число `49`, это можно сделать так (см. рис. 2.10):

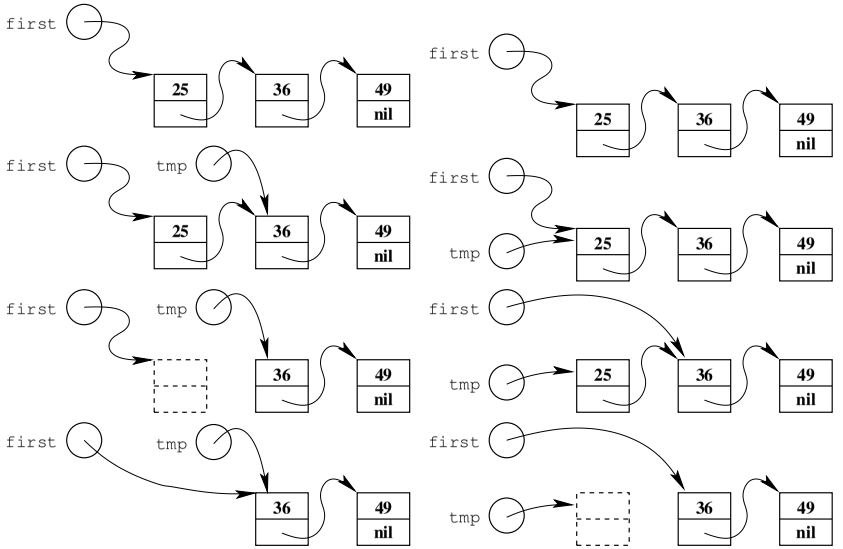

Рис. 2.9. Два способа удаления первого элемента из односвязного списка

```
new(last^.next);
last := last^.next;
last^.data := 49;
last^.next := nil;
```

Как это часто случается, всякое упрощение имеет свою цену. В отличие от занесения в начало, когда для списка хранится только указатель `first`, при наличии двух указателей **занесение первого элемента оказывается специальным случаем**, который требует совершенно иного набора действий: в самом деле, указатель `last` в этой ситуации никуда не указывает, ведь последнего элемента пока не существует (и вообще никакого не существует), так что не существует и переменной `last^.next`, которую мы столь лихо задействовали. Правильная последовательность действий для занесения числа 49 в этом случае будет такая:

```
new(first);          { создаём первый элемент списка }
last := first;          { объявляем его же последним }
last^.data := 49;                          { заполняем }
last^.next := nil;
```

Как видим, последние две строки остались точно такими же, как и раньше (мы специально постарались так сделать), тогда как первые две совершенно поменялись. Сказанное можно обобщить. Если число, которое мы заносим в список, находится в переменной `n`, причём мы не

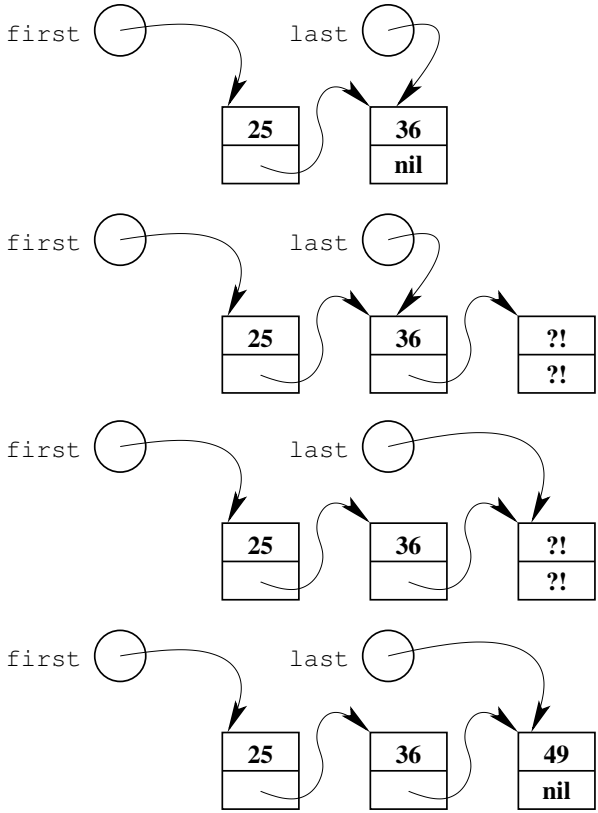

Рис. 2.10. Добавление нового элемента в конец списка

знаем, есть к этому времени в списке хотя бы один элемент или нет, то корректный код для занесения элемента в список будет таким:

```
if first = nil then
begin
    new(first);
    last := first;
end
else
begin
    new(last^.next);
    last := last^.next;
end;
last^.data := n;
last^.next := nil;
```

Полностью решение второй задачи мы не приводим; с учётом всего сказанного оно должно быть совершенно очевидно.

2.13.5. Стек и очередь

В некоторых учебниках по информатике можно встретить утверждение, что-де «стек» и «очередь» — это такие структуры данных; стеком в таких случаях обычно называют ту сущность, которую мы в предыдущем параграфе называли односвязным списком. Нам остаётся лишь призвать читателя не верить в подобные антинаучные россказни; на самом деле как стек, так и очередь могут быть реализованы совершенно разными структурами данных (списками, массивами, разнообразными комбинациями того и другого, ту же очередь часто реализуют с помощью кольцевого буфера и т. д.; больше того, значения даже не обязательно хранить в оперативной памяти, часто приходится использовать файлы на диске); сами по себе стек и очередь определяются не тем, какова реализующая их структура данных, а тем, каковы доступные пользователю *операции* и как эти операции работают.

Как стек, так и очередь представляют собой некий абстрактный «объект», обеспечивающий (неважно как) хранение значений определённого типа, причём для этого объекта определены две базовые операции: добавление нового значения и извлечение значения. Разница в том, в каком порядке происходит извлечение: из очереди значения извлекаются в том же порядке, в каком их туда занесли, а из стека — в обратном. По-английски очередь обычно обозначают аббревиатурой FIFO (*first in, first out*, т. е. «первым вошёл — первым вышел»), а стеки — аббревиатурой LIFO (*last in, first out*, «последним вошёл — первым вышел»).

Иначе говоря, очередь — это такая штука, куда можно складывать значения какого-нибудь типа (например, целочисленного), а можно их оттуда извлекать, и при извлечении мы всегда получаем значение, которое (среди тех, что были занесены, но пока что не были извлечены) в нашей очереди пролежало дольше других; со стеком ситуация обратная: первыми извлекаются самые «свежие» значения.

Самый простой вариант реализации стека нам уже известен: это односвязный список, при работе с которым мы всегда имеем дело с его началом: именно в начало добавляются новые элементы, и извлекаются элементы тоже из начала. Немного подумав, мы можем заметить, что очередь тоже очень легко реализовать на базе односвязного списка, только добавление новых элементов следует производить не в начало, а в конец (а извлекать элементы по-прежнему из начала).

Обратим внимание, что речь в обоих случаях идёт о реализации механизма, относительно которого нам важны доступные операции, а то, *как* эти операции сделаны, может остаться за кадром. Такая ситуация в программировании возникает достаточно часто; именно этот факт породил так называемое объектно-ориентированное программирование, суперпопулярное в наше время. Что это такое, мы узнаем позже; в третьем томе нашей книги этому будет посвящена отдельная часть. Пока до

ООП дело не дошло, отметим, что в любой подобной ситуации программисты стараются создать набор подпрограмм (процедур), реализующих все нужные операции, причём описать их так, чтобы даже в случае изменения принципов реализации (в нашем случае, например, при переходе от использования списка к использованию какого-нибудь массива) поменялся только текст этих процедур, а программист, использующий их, вообще ничего не заметил.

Иначе говоря, если кто-то просит нас реализовать тот же стек или какой-то другой абстрактный объект, для которого важны операции, а не его внутреннее устройство, мы обычно пишем набор процедур и функций и объясняем, как этим набором пользоваться, при этом тщательно следим, чтобы все наши объяснения никак не зависели от того, что́ мы используем внутри наших процедур в качестве механизма реализации. Такой набор процедур вместе с пояснениями, как их нужно использовать, называется *интерфейсом*[41]. Считается, что интерфейс придуман удачно, если никакие изменения в деталях реализации нашего абстрактного объекта не приводят к изменениям ни в самом интерфейсе, ни в объяснениях, как им пользоваться, то есть могут безболезненно игнорироваться теми программистами, которые станут использовать наше изделие.

Один из подходов, позволяющих создавать хорошие интерфейсы, состоит в том, чтобы сначала придумать именно интерфейс, исходя из набора операций, которых от нас ожидает пользователь, и уже потом попытатся написать какую-то реализацию, которая позволит процедурам нашего интерфейса работать в соответствии с замыслом. Поскольку о реализации мы начинаем думать уже *после* создания интерфейса, такой интерфейс с хорошей вероятностью окажется не зависящим от конкретной реализации, что нам и требуется.

Попробуем применить сказанное к случаю стека; для определённости пусть это будет стек целых чисел типа `longint`. Ясно, что, какова бы ни была его реализация, *хоть что-то*, хоть какую-то информацию нам при этом обязательно придётся хранить (при реализации в виде односвязного списка это будет указатель на первый элемент списка, но мы пока об этом думать не хотим). *Хранить информацию* мы можем только в переменных, причём если мы вспомним о существовании переменных типа «запись», то заметим, что любую информацию (разумного размера) мы можем при желании сохранить в *одной переменной*, нужно только выбрать для неё подходящий тип. Итак, какова бы ни была реализация нашего стека, нам необходимо и достаточно для каждого вводимого стека (которых может быть сколько угодно) описать переменную некоего типа; не зная, какова будет реализация, мы не можем

[41] Часто встречается английская аббревиатура API, образованная от слов *application programming interface*, то есть «интерфейс прикладного программирования».

сказать, какой конкретно это будет тип, но это не помешает нам дать ему имя. Поскольку речь идёт о стеке чисел типа `longint`, назовём этот тип `StackOfLongints`.

Основные операции над стеком по традиции обозначаются словами *push* (добавление значения в стек) и *pop* (извлечение значения). Обе эти операции мы реализуем в виде подпрограмм; в принципе, было бы логично соответствующие подпрограммы назвать как-то вроде `StackOfLongintsPush` и `StackOfLongintsPop`, но, посмотрев внимательно на получившиеся длинные имена, мы можем (вполне естественно) усомниться в том, что пользоваться ими будет удобно. Поэтому везде, кроме собственно имени типа, мы громоздкое `StackOfLongints` заменим на короткое `SOL`, а подпрограммы назовём, соответственно, `SOLPush` и `SOLPop`. Заметим ещё, что занесение нового элемента в стек всегда проходит удачно[42], тогда как извлечение из стека, если он пуст, успешным считаться не может. Поэтому `SOLPush` у нас будет процедурой, получающей на вход два параметра — сам стек и число, которое в него нужно занести (причём стек будет передаваться как параметр-переменная, а число — как простой параметр), тогда как `SOLPop` будет оформлена в виде логической функции, возвращающей истину в случае успеха и ложь при неудаче; параметров у неё тоже будет два — переменная-стек и переменная типа `longint`, в которую (в случае успеха) будет записываться результат.

Собственно говоря, мы почти закончили придумывать интерфейс. Добавим для удобства ещё функцию `SOLIsEmpty`, получающую в качестве параметра стек и возвращающую истину, если он пуст, и ложь в противном случае. Наконец, вспомним, что переменная может, если ей ничего не присвоить, содержать произвольный мусор, и добавим процедуру, превращающую свежеописанную переменную в корректный пустой стек; назовём её `SOLInit`. Всё, интерфейс готов. Вспомнив наш подход к *пошаговой детализации*, при котором для начала пишутся пустые подпрограммы, мы можем зафиксировать наш интерфейс в виде фрагмента программы:

```
type
    StackOfLongints = ;

procedure SOLInit(var stack: StackOfLongints);
begin
end;

procedure SOLPush(var stack: StackOfLongints; n: longint);
begin
```

[42] Кроме случая, когда не хватит памяти, но современные операционные системы таковы, что в этом случае наша программа о возникшей проблеме уже не узнает: её просто прибьют, причём автоматически.

```
end;

function SOLPop(var stack: StackOfLongints; var n: longint)
                                                    : boolean;
begin
end;

function SOLIsEmpty(var stack: StackOfLongints) : boolean;
begin
end;
```

Конечно, такой фрагмент даже не откомпилируется, поскольку мы не указали, что такое `StackOfLongints`; но ведь надо же с чего-то начинать. Теперь, когда интерфейс полностью зафиксирован, мы можем приступить к реализации. Поскольку мы хотели использовать односвязный список, нам потребуется тип для его звена:

```
type
    LongItemPtr = ^LongItem;
    LongItem = record
        data: longint;
        next: LongItemPtr;
    end;
```

Теперь мы можем начать обращивать «мясом» наши «заглушки» интерфейсных подпрограмм. Для начала заметим, что для работы со стеком нужен только указатель на начало списка, так что в качестве переменной типа `StackOfLongints` можно использовать переменную типа `LongItemPtr`. Отразим этот факт:

```
type
    StackOfLongints = LongItemPtr;
```

Теперь берём наши четыре подпрограммы и реализуем их тела:

```
procedure SOLInit(var stack: StackOfLongints);
begin
    stack := nil         { корректный пустой стек }
end;

procedure SOLPush(var stack: StackOfLongints; n: longint);
var
    tmp: LongItemPtr;
begin
    new(tmp);
    tmp^.data := n;
    tmp^.next := stack;
    stack := tmp
```

```
end;

function SOLPop(var stack: StackOfLongints; var n: longint)
                                                    : boolean;
var
    tmp: LongItemPtr;
begin
    if stack = nil then
    begin
        SOLPop := false;
        exit
    end;
    n := stack^.data;
    tmp := stack;
    stack := stack^.next;
    dispose(tmp);
    SOLPop := true
end;

function SOLIsEmpty(var stack: StackOfLongints) : boolean;
begin
    SOLIsEmpty := stack = nil
end;
```

С использованием этих процедур решение задачи о выдаче введённой последовательности чисел в обратном порядке, которую мы рассматривали в предыдущем параграфе, может стать гораздо изящнее:

```
var
    s: StackOfLongints;
    n: longint;
begin
    SOLInit(s);
    while not eof do
    begin
        readln(n);
        SOLPush(s, n)
    end;
    while not SOLEmpty(s) do
    begin
        SOLPop(s, n);
        writeln(n)
    end
end.
```

Второй цикл можно записать ещё короче:

```
    while SOLPop(s, n) do
        writeln(n)
```

Попробуем применить ту же методику к очереди. Ясно, что *какая-то переменная* нам для организации очереди точно понадобится; назовём её тип `QueueOfLongints`. Основные операции над очередью, в отличие от стека, обычно называют *put* и *get*; как и для стека, для очереди будут нужны процедура инициализации и функция, позволяющая узнать, не пуста ли очередь. Для фиксации интерфейса можно написать следующие заглушки:

```
type
    QueueOfLongints = ;

procedure QOLInit(var queue: QueueOfLongints);
begin
end;

procedure QOLPut(var queue: QueueOfLongints; n: longint);
begin
end;

function QOLGet(var queue: QueueOfLongints; var n: longint)
                                                  : boolean;
begin
end;

function QOLIsEmpty(var queue: QueueOfLongints) : boolean;
begin
end;
```

Для реализации используем список из элементов того же типа `LongItem`, но заметим, что нам нужно для представления очереди *два* указателя типа `LongItemPtr`; чтобы объединить их в одну переменную, воспользуемся записью:

```
type
    QueueOfLongints = record
        first, last: LongItemPtr;
    end;
```

Реализация интерфейсных процедур будет выглядеть так:

```
procedure QOLInit(var queue: QueueOfLongints);
begin
    queue.first := nil;
    queue.last := nil
end;

procedure QOLPut(var queue: QueueOfLongints; n: longint);
begin
```

```
    if queue.first = nil then
    begin
        new(queue.first);
        queue.last := queue.first
    end
    else
    begin
        new(queue.last^.next);
        queue.last := queue.last^.next
    end;
    queue.last^.data := n;
    queue.last^.next := nil
end;

function QOLGet(var queue: QueueOfLongints; var n: longint)
                                                  : boolean;
var
    tmp: LongItemPtr;
begin
    if queue.first = nil then
    begin
        QOLGet := false;
        exit
    end;
    n := queue.first^.data;
    tmp := queue.first;
    queue^.first := queue.first^.next;
    if queue^.first = nil then
        queue^.last := nil;
    dispose(tmp);
    QOLGet := true
end;

function QOLIsEmpty(var queue: QueueOfLongints) : boolean;
begin
    QOLIsEmpty := queue^.first = nil
end;
```

2.13.6. Проход по списку указателем на указатель

Рассмотрим довольно простую задачу: пусть у нас есть всё тот же список целых чисел и нам нужно удалить из него все элементы, содержащие отрицательные значения. Сложность здесь в том, что для удаления элемента из списка *необходимо изменить тот указатель, который на него указывает*; если удаляемый элемент в списке не первый, то изменять нужно значение указателя `next` в *предыдущем* элементе списка, если же он первый, то изменять нужно указатель `first`.

Сходу становится ясно, что выполнять проход по списку так, как мы это делали, например, для его печати, в этой задаче не получится. В самом деле, выполняя цикл по знакомой нам схеме

```
tmp := first;
while tmp <> nil do
begin
    { действия с элементом tmp^ }
    tmp := tmp^.next
end
```

мы на момент работы с элементом *уже не помним, где находится в памяти предыдущий элемент*, и не можем изменить значение его поля `next`. Можно попытаться с этой проблемой справиться, храня в переменной цикла как раз адрес *предыдущего* элемента:

```
tmp := first;
while tmp^.next <> nil do
begin
    { действия с элементом tmp^.next^ }
    tmp := tmp^.next
end
```

но тогда *первый* элемент списка придётся обрабатывать отдельно; это, в свою очередь, повлечёт специальную обработку для случая пустого списка. В итоге для удаления всех отрицательных значений мы получим что-то вроде следующего фрагмента:

```
if first <> nil then
begin
    while first^.data < 0 do  { удаление из начала }
    begin
        tmp := first;
        first := first^.next;
        dispose(tmp)
    end;
    tmp := first;
    while tmp^.next <> nil do
    begin
        if tmp^.next^.data < 0 then
        begin
            tmp2 := tmp^.next;
            tmp^.next := tmp^.next^.next;
            dispose(tmp2)
        end
        else
            tmp := tmp^.next
    end
end
```

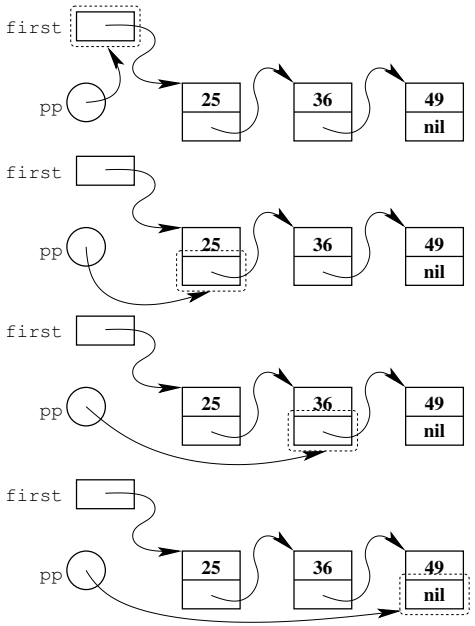

Рис. 2.11. Указатель на указатель в качестве переменной цикла

Полученное решение трудно назвать красивым: фрагмент получился громоздким из-за наличия двух циклов и объемлющего `if`'а, его достаточно трудно читать. Между тем, на самом деле *при удалении первого элемента и любого другого действия выполняются абсолютно одинаковые*, только в первом случае они выполняются над указателем `first`, а во втором — над `tmp^.next`.

Сократить размеры и сделать решение понятнее позволяет нетривиальный приём, предполагающий задействование *указателя на указатель*. Если тип звена списка у нас называется, как и раньше, `item`, а тип указателя на него — `itemptr`, то такой указатель на указатель описывается следующим образом:

```
var
    pp: ^itemptr;
```

Полученная переменная `pp` может содержать адрес *указателя на* `item`, то есть она с одинаковым успехом может хранить в себе как адрес нашего `first`, так и адрес поля `next`, расположенного в любом из элементов списка. Именно этого мы и добиваемся; цикл прохода по списку с удалением отрицательных элементов мы организуем с помощью `pp` в качестве переменной цикла, причём в `pp` сначала будет содержаться адрес `first`, затем — адрес `next` из первого звена списка, из второго

звена и так далее (см. рис. 2.11). В целом pp будет в каждый момент времени *указывать на указатель, в котором располагается адрес текущего (рассматриваемого) звена списка.* Начальное значение переменная pp получит в результате очевидного присваивания pp := @first, переход к рассмотрению следующего элемента будет выглядеть несколько сложнее:

```
pp := @(pp^^.next);
```

Если две «крышки» после pp кажутся вам чем-то слишком сложным, вспомните, что pp^ есть то, на что указывает pp, то есть, как мы договорились, указатель на текущий элемент; следовательно, pp^^ есть сам этот текущий элемент, pp^^.next — это его поле next (то есть как раз указатель на следующий элемент), от него мы берём адрес, этот адрес заносим в pp — и после этого мы готовы к работе со следующим элементом.

Отметим, что переход к следующему следует выполнять только в том случае, если текущий элемент мы не удалили; если же элемент был удалён, то переход к рассмотрению следующего за ним произойдёт сам собой, ведь хотя адрес в pp у нас не изменится, но изменится адрес, хранящийся в указателе pp^ (будь то first или очередное поле next, неважно).

Осталось понять, каково должно быть условие такого цикла. Закончить работу следует, когда очередного элемента нет, то есть когда указатель на очередной элемент содержит nil. Это опять-таки может оказаться как указатель first (если список пуст), так и поле next последнего элемента в списке (если в нём есть хотя бы один элемент). Проверить такое условие можно выражением pp^ <> nil. Окончательно (в предположении, что переменная tmp имеет, как и раньше, тип itemptr, а переменная pp имеет тип ^itemptr) наш цикл примет следующий вид:

```
pp := @first;
while pp^ <> nil do
begin
    if pp^^.data < 0 then
    begin
        tmp := pp^;
        pp^ := pp^^.next;
        dispose(tmp)
    end
    else
        pp := @(pp^^.next)
end
```

Сравнивая это решение с предыдущим, можно заметить, что оно практически дословно повторяет цикл, который в том решении был внутренним,

но при этом у нас исчез объемлющий `if` и отдельный цикл для удаления элементов из начала. Применение указателя на указатель позволило нам *обобщить* основной цикл, сняв тем самым необходимость в обработке особых случаев.

Применение указателя на указатель позволяет не только удалять элементы из произвольного места списка, но и вставлять элементы в любую его позицию — точнее, в то место, куда сейчас указывает указатель, *на который* указывает рабочий указатель. Если в рабочем указателе находится адрес указателя `first`, вставка будет выполняться *в начало* списка, если же там адрес поля `next` одного из звеньев списка, то вставка произойдёт *после* этого звена, т. е. перед следующим звеном. Это может оказаться и вставка в конец списка, если только в рабочем указателе будет находиться адрес поля `next` из последнего звена списка. Например, если рабочий указатель у нас по-прежнему называется `pp`, вспомогательный указатель типа `itemptr` называется `tmp`, а число, которое нужно добавить в список, хранится в переменной `n`, то вставка звена с числом `n` в позицию, обозначенную с помощью `pp`, будет выглядеть так:

```
new(tmp);
tmp^.next := pp^;
tmp^.data := n;
pp^ := tmp;
```

Как можно заметить, этот фрагмент дословно повторяет процедуру вставки элемента в начало односвязного списка, которую мы рассматривали на стр. 360, только вместо `first` используется `pp^`; как и в случае с удалением элемента, такой способ вставки элемента представляет собой *обобщение* процедуры вставки в начало. Обобщение основано на том, что любой «хвост» односвязного списка также представляет собой односвязный список, только вместо `first` для него используется поле `next` из элемента, предшествующего такому «хвосту». Этим фактом мы ещё воспользуемся; помимо прочего, он позволяет довольно естественное применение рекурсии для обработки односвязных списков.

Умея вставлять звенья в произвольные места списка, мы можем, например, работать со списком чисел, отсортированных по возрастанию (или по убыванию, или по любому другому критерию). При этом нужно следить за тем, чтобы каждый элемент вставлялся точно в нужную позицию списка — так, чтобы после такого добавления список оставался отсортированным.

Делается это достаточно просто. Если, как и раньше, число, которое нужно вставить, хранится в переменной `n`, а список, на начало которого указывает `first`, при этом отсортирован в порядке возрастания, то вставка нового элемента быть сделана, например, так:

```
pp := @first;
while (pp^ <> nil) and (pp^^.data < n) do
    pp := @(pp^^.next);
new(tmp);
tmp^.next := pp^;
tmp^.data := n;
pp^ := tmp;
```

Интерес здесь представляют только первые три строчки, остальные четыре мы уже видели. Цикл `while` здесь позволяет найти нужное место в списке, или, точнее, *тот указатель*, который указывает на звено, *перед которым* надо вставить новый элемент. Начинаем мы, как обычно, с начала, в данном случае это означает, что в рабочий указатель заносится адрес указателя `first`. Очередного элемента в списке может вообще не оказаться — если в списке нет вообще ни одного элемента, либо если все значения, хранящиеся в списке, оказались меньше того, которое будет вставлено сейчас; при этом в указателе `pp^` (то есть в том, на который указывает наш рабочий указатель) окажется значение `nil` — отсюда первая часть условия в цикле. В такой ситуации идти нам дальше некуда, вставлять нужно прямо сейчас.

Вторая часть условия цикла обеспечивает поиск нужной позиции: пока очередной элемент списка *меньше*, чем вставляемый, нам нужно двигаться по списку дальше.

Здесь можно обратить внимание на одну важную особенность вычисления логических выражений. Если, к примеру, указатель `pp^` оказался равен `nil`, то попытка обратиться к переменной `pp^^.data` приведёт к немедленному аварийному завершению программы, ведь записи `pp^^` попросту не существует. К счастью, такого обращения не произойдёт. Так получается, потому что при вычислении логических выражений Free Pascal использует «ленивую семантику»: *если значение выражения уже известно, дальнейшие его подвыражения не вычисляются*. В данном случае между двумя частями условия стоит связка `and`, так что, если первая часть ложна, то и всё выражение окажется ложным, поэтому вычислять его вторую часть не нужно.

Полезно будет знать, что в оригинальном Паскале, как его описал Никлаус Вирт, этого свойства не было: при вычислении любого выражения там обязательно вычислялись все его подвыражения. Если бы Free Pascal в этом плане действовал так же, нам пришлось бы изрядно покрутиться, поскольку вычисление условия `(pp^ <> nil) and (pp^^.data < n)` гарантированно приводило бы к аварии при `pp^`, равном `nil`. Надо сказать, что оператора `break` в оригинальном Паскале тоже не было. В ситуациях, аналогичных нашей, приходилось вводить дополнительную логическую переменную и обвешивать тело цикла `if`'ами. Впрочем, в оригинальном Паскале не было и операции взятия адреса, так что применить нашу технику «указатель на указатель» мы бы там не смогли.

Free Pascal позволяет задействовать «классическую» семантику вычисления логических выражений, когда все подвыражения обязательно вычисляются. Это включается директивой `{$B+}`, а выключается директивой `{$B-}`. К счастью, по умолчанию используется именно режим `{$B-}`. Скорее всего, вам в вашей

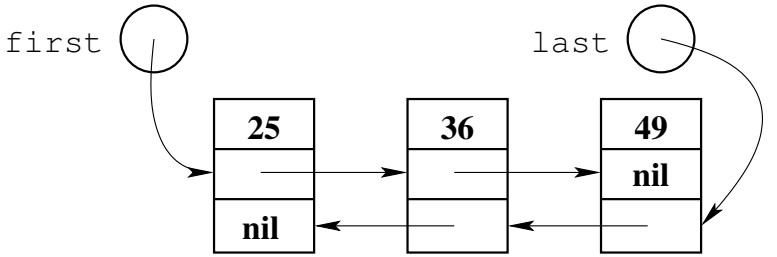

Рис. 2.12. Двусвязный список из трёх элементов

практике никогда не потребуется этот режим менять; если вам кажется, что такая потребность возникла, подумайте ещё.

2.13.7. Двусвязные списки; деки

Списки, которые мы рассматривали до сих пор, называются *односвязными*, что очевидным образом наводит на мысль о существовании каких-то других списков. В этом параграфе мы рассмотрим списки, называемые ***двусвязными***.

Вводя понятие *односвязного списка*, мы отметили, что это название с одинаковым успехом можно объяснить наличием только одного указателя, указывающего на каждый из элементов списка; наличием в каждом элементе только одного указателя для поддержания связности; и наличием в списке только одной связной цепочки, выстроенной из указателей. Подобно этому, в двусвязном списке на каждый элемент списка указывает два указателя (из предыдущего элемента и из следующего), каждый элемент имеет два указателя — на предыдущий элемент и на следующий, и в списке присутствуют две цепочки связности — прямая и обратная. Следует отметить, что обычно (хотя и не всегда) с таким списком работают с помощью двух указателей — на первый его элемент и на последний. Пример двусвязного списка показан на рис. 2.12.

Построить двусвязный список можно из звеньев, описанных, например, так:

```
type
    item2ptr = ^item2;
    item2 = record
        data: integer;
        prev, next: item2ptr;
    end;
```

Как и в случае со словом `next` для указателя на следующий элемент, слово `prev` (от англ. *previous*) традиционно обозначает указатель на

предыдущий; конечно, вам никто не мешает использовать другое название, но если вашу программу будет читать кто-то ещё, то понять её окажется сложнее.

Надо сказать, что двусвязные списки применяются несколько реже, чем односвязные, поскольку требуют вдвое больше действий с указателями при любых операциях вставки и удаления звеньев, да и сами звенья за счёт наличия двух указателей вместо одного становятся больше, т. е. занимают больше памяти. С другой стороны, у двусвязных списков имеется целый ряд несомненных достоинств в сравнении с односвязными. Прежде всего это очевидная симметрия, которая позволяет просматривать список как в прямом направлении, так и в обратном; в некоторых задачах это оказывается важно.

Второе несомненное достоинство двусвязного списка не столь очевидно, но иногда оказывается даже важнее: *зная адрес любого из звеньев двусвязного списка, мы можем найти в памяти все его звенья.* Между прочим, при работе с двусвязным списком никогда не требуется рассмотренная в предыдущем параграфе техника с использованием указателя на указатель. В самом деле, пусть на текущее звено указывает некий указатель `current`; если в текущем звене поле `prev` нулевое, то, значит, это первое звено и при вставке нового элемента слева от него требуется изменить `first`; в противном случае звено не первое, то есть имеется предыдущее, в котором нужно изменить поле `next`, но ведь мы знаем, где оно находится: выражение `current^.prev^.next` как раз даёт тот указатель, который надо будет изменить. Кроме того, нужно будет изменить поле `current^.prev`. Аналогично производится и вставка справа от текущего звена: сначала нужно поменять `current^.next^.prev`, а если его нет (то есть текущее звено — последнее в списке), то `last`; затем меняется `current^.next`.

Если на первое, последнее и текущее звенья двусвязного списка указывают соответственно указатели `first`, `last` и `current`, новое число находится в переменной `n`, а временный указатель называется `tmp`, вставка нового звена слева от текущего выглядит так:

```
new(tmp);
tmp^.prev := current^.prev;
tmp^.next := current;
tmp^.data := n;
if current^.prev = nil then
    first := tmp
else
    current^.prev^.next := tmp;
current^.prev := tmp;
```

Вставка справа выглядит аналогично:

```
new(tmp);
```

```
tmp^.prev := current;
tmp^.next := current^.next;
tmp^.data := n;
if current^.next = nil then
    last := tmp
else
    current^.next^.prev := tmp;
current^.next := tmp;
```

Удаление текущего элемента делается и того проще:

```
if current^.prev = nil then
    first := current^.next
else
    current^.prev^.next := current^.next;
if current^.next = nil then
    last := current^.prev
else
    current^.next^.prev := current^.prev;
dispose(current);
```

Этот фрагмент обладает некоторым недостатком: указатель `current` после его выполнения указывает на несуществующий (уничтоженный) элемент. Этого можно избежать, если воспользоваться временным указателем для удаления элемента, а в `current` предварительно (до уничтожения текущего элемента) занести адрес предыдущего или следующего, в зависимости от того, проходим мы список в прямом направлении или в обратном.

Добавление элемента в начало двусвязного списка с учётом возможного особого случая может выглядеть, например, так:

```
new(tmp);
tmp^.data := n;
tmp^.prev := nil;
tmp^.next := first;
if first = nil then
    last := tmp
else
    first^.prev := tmp;
first := tmp;
```

Добавление в конец производится аналогично (с точностью до замены направления):

```
new(tmp);
tmp^.data := n;
tmp^.prev := last;
tmp^.next := nil;
```

```
if last = nil then
    first := tmp
else
    last^.next := tmp;
last := tmp;
```

Можно обобщить вышеприведённую процедуру вставки слева от текущего звена, предположив, что если текущего нет (то есть вставка происходит «слева от несуществующего звена»), то это вставка в конец списка. Выглядеть это будет так:

```
new(tmp);
if current = nil then
    tmp^.prev := last
else
    tmp^.prev := current^.prev;
tmp^.next := current;
tmp^.data := n;
if tmp^.prev = nil then
    first := tmp
else
    tmp^.prev^.next := tmp;
if tmp^.next = nil then
    last := tmp
else
    tmp^.next^.prev := tmp;
```

Аналогичное обобщение вставки справа от текущего звена на ситуацию «вставка справа от несуществующего есть вставка в начало» происходит практически так же, меняются только строки, отвечающие за заполнение полей `tmp^.prev` и `tmp^.next` (в нашем фрагменте это строки со второй по шестую):

```
tmp^.prev := current;
if current = nil then
    tmp^.next := first
else
    tmp^.next := current^.next;
```

Схематические диаграммы изменения указателей для всех этих случаев мы не приводим, оставляя визуализацию происходящего читателю в качестве очень полезного упражнения.

Двусвязные списки позволяют создать объект, обычно называемый ***декой*** (англ. *deque*[43]). Дека — это абстрактный объект, поддерживающий четыре операции: добавление в начало, добавление в конец,

[43]На самом деле происхождение термина «дека» не столь очевидно. Исходно рассматриваемый объект назывался «очередью с двумя концами», по-английски *double-ended queue*; эти три слова англоговорящие программисты сократили сначала до *dequeue*, а потом и вовсе до *deque*.

извлечение из начала и извлечение из конца. Значение, добавленное в начало деки, можно сразу же извлечь обратно (применив извлечение из начала), но если извлекать значения из конца, то значение, только что добавленное в начало, будет извлечено после всех остальных значений, хранившихся в деке; с добавлением в конец и извлечением из начала ситуация симметричная.

При использовании только операций «добавление в начало» и «извлечение из начала» дека превращается в стек (как, впрочем, и при использовании добавления и извлечения из конца), а при использовании «добавления в начало» и «извлечения из конца» — в очередь; но использовать деку в качестве стека или обычной очереди не стоит, ведь для стека и очереди возможны гораздо более простые реализации: их можно реализовать через односвязный список, тогда как деку односвязным списком реализовать нельзя (точнее, можно использовать два таких списка, но это очень неудобно).

Отметим, что по-английски соответствующие операции обычно называются *pushfront*, *pushback*, *popfront* и *popback*.

Начать реализацию деки можно, как обычно, с заглушки. К примеру, для деки, хранящей числа типа `longint`, заглушка будет такая:

```
type
    LongItem2Ptr = ^LongItem2;
    LongItem2 = record
        data: longint;
        prev, next: LongItem2Ptr;
    end;

    LongDeque = record
        first, last: LongItem2Ptr;
    end;

procedure LongDequeInit(var deque: LongDeque);
begin
end;
procedure LongDequePushFront(var deque: LongDeque, n: longint);
begin
end;
procedure LongDequePushBack(var deque: LongDeque, n: longint);
begin
end;
function LongDequePopFront(var deque: LongDeque, var n: longint)
                                                        : boolean;
begin
end;
function LongDequePopBack(var deque: LongDeque, var n: longint)
                                                        : boolean;
begin
```

```
end;
function LongDequeIsEmpty(var deque: LongDeque) : boolean;
begin
end;
```

Реализовать все эти процедуры и функции мы предложим читателю самостоятельно.

2.13.8. Обзор других динамических структур данных

Прежде всего отметим, что во многих задачах звено списка содержит не одно поле данных, как в наших примерах, а несколько таких полей. Часто возникает потребность связать одно значение с другими, например, инвентарный номер с описанием предмета, или имя человека с его подробной анкетой, или сетевой адрес компьютера с описанием работающих на нём сетевых служб и т. п. В этом случае в одном звене размещают как ту информацию, по которой происходит поиск (номер, имя, адрес), так и те данные, которые служат результатом поиска. Отметим, что информацию, по которой проводится поиск, называют ***ключом поиска***.

Поиск нужной информации в списке — неважно, односвязный он или двусвязный — происходит достаточно медленно, ведь список приходится просматривать звено за звеном, пока не будет найден нужный ключ. Даже если список поддерживается в отсортированном по ключу порядке, в среднем приходится на каждый поиск просмотреть половину звеньев списка. Иначе говоря, в среднем при поиске записи с нужным ключом в списке на каждый поиск тратится kn операций, где n — длина списка, а k — некий постоянный коэффициент; иначе говоря, длительность каждой операции поиска *линейно зависит* от количества хранящихся записей. Если, скажем, длина списка возрастёт втрое, то во столько же раз возрастёт среднее время, затрачиваемое на поиск нужной записи. Такой поиск так и называют ***линейным***.

Линейный поиск может удовлетворить нас только в задачах, где количество записей не превышает нескольких десятков; если записей будет несколько сотен, с линейным поиском начнутся проблемы; при нескольких тысячах записей пользователь может возмутиться тем. насколько медленно работает наша программа, и будет прав; ну а когда счёт записей переходит в область десятков тысяч, про линейный поиск следует забыть — с таким же успехом мы могли бы вовсе не писать программу, всё равно она окажется бесполезной.

К счастью, многообразие структур данных, построенных из звеньев, плавающих в «куче», никоим образом не исчерпывается списками. Резко сократить время поиска позволяют более «хитрые» структуры данных;

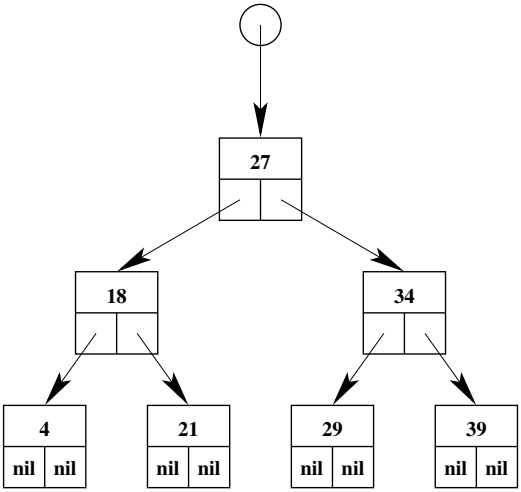

Рис. 2.13. Двоичное дерево поиска, хранящее целые числа

один из самых простых вариантов здесь — так называемое ***двоичное дерево поиска***. Как и список, дерево строится из переменных типа «запись», содержащих поля-указатели на запись того же типа; записи, составляющие дерево, обычно называют ***узлами***. В простейшем случае каждый узел имеет два поля-указателя — на правое поддерево и на левое поддерево; обычно их называют `left` и `right`. Указатель, содержащий значение `nil`, рассматривается как пустое поддерево.

Двоичное дерево поиска поддерживается в *отсортированном* виде; в данном случае это означает, что как для самого дерева, так и для любого его поддерева все значения, которые меньше значения в текущем узле, располагаются в левом поддереве, а все значения, бóльшие текущего — в правом. Пример двоичного дерева поиска, содержащего целые числа, приведён на рис. 2.13. В таком дереве найти любой элемент, зная нужное значение, либо убедиться, что такого элемента в дереве нет, можно за число действий, пропорциональных ***высоте дерева***, то есть длине наибольшей возможной связной цепочки от корня дерева до «листового» узла (в дереве на рисунке такая высота составляет три). Отметим, что двоичное дерево высотой h может содержать до $2^h - 1$ узлов; например, в дерево высотой 20 можно поместить свыше *миллиона* значений, при этом поиск будет происходить за 20 сравнений; если бы те же значения находились в списке, то сравнений на каждый поиск в среднем требовалось бы полмиллиона.

Конечно, при ближайшем рассмотрении всё оказывается не столь просто и радужно. Дерево далеко не всегда так плотно заполняется узлами, как на нашем рисунке. Пример такой ситуации приведён на рис. 2.14; при высоте 5 дерево здесь содержит всего девять узлов из 31

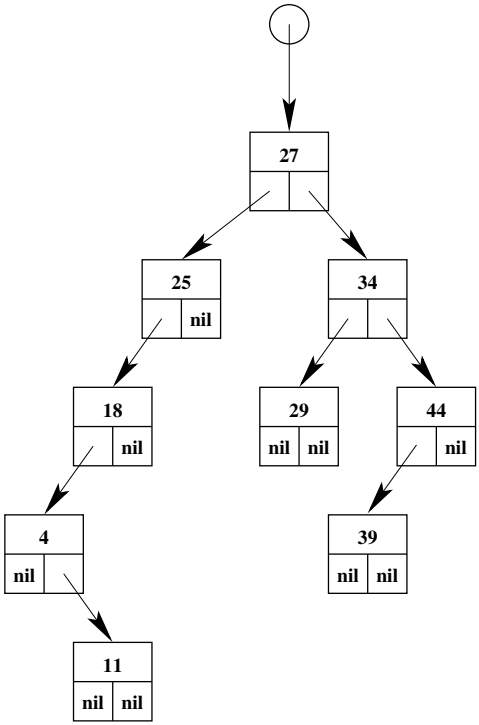

Рис. 2.14. Разбалансированное двоичное дерево поиска

возможных. Такое дерево называется ***разбалансированным***. В наихудшем случае дерево вообще может принять вырожденную форму, когда его высота будет равна количеству элементов; это происходит, например, если значения, вносимые в дерево, исходно расположены по возрастанию или по убыванию. Впрочем, то, что для дерева представляет собой *наихудший* вариант, для списка происходит *всегда*. Кроме того, существуют алгоритмы балансировки деревьев, позволяющие перестроить дерево поиска, уменьшив его высоту до минимально возможной; эти алгоритмы достаточно сложны, но реализовать их можно.

К рассмотрению двоичных деревьев поиска мы вернёмся в главе, посвящённой «продвинутым» случаям использования рекурсии. Дело в том, что все основные операции над двоичным деревом, кроме разве что балансировки, многократно (!) проще записать с помощью рекурсии, чем без неё. Пока же отметим, что деревья, разумеется, бывают не только двоичными; в общем случае количество потомков у узла дерева можно сделать сколь угодно большим или вообще динамически изменяемым. Например, широко известен способ хранения текстовых строк, при котором каждый узел дерева может иметь столько потомков,

сколько есть букв в алфавите; при поиске выбор потомка очередного узла производится в соответствии с очередной буквой в строке. Высота такого дерева строго равна длине самой длинной из хранящихся в нём строк, а в балансировке оно не нуждается.

Время поиска в списке пропорционально длине списка, время поиска в сбалансированном двоичном дереве, как можно догадаться, пропорционально двоичному логарифму от количества узлов; для случаев, когда время поиска критично, а объём хранимых данных велик, известен подход, при котором *время поиска вообще не зависит от количества хранимых элементов* — оно сохраняется постоянным хоть для десятка значений, хоть для десяти миллионов. Метод носит название ***хеш-таблицы***.

Для создания хеш-таблицы используют так называемую хеш-функцию. В принципе, это просто некая арифметическая функция, которая по заданному значению ключа поиска выдаёт некое целое число, причём это число должно быть трудно предугадать; говорят, что хеш-функция должна представлять собой *хорошо распределённую случайную величину*. При этом важно, что хеш-функция должна зависеть только от значения ключа, то есть на одних и тех же ключах хеш-функция должна всегда принимать одни и те же значения. Сама хеш-таблица представляет собой массив размера, заведомо превосходящего нужное количество записей; изначально все элементы массива помечаются как свободные. Вычислив хеш-функцию, по её значению определяют, в какой позиции массива должна находиться запись с заданным ключом; это делают, попросту вычислив остаток от деления значения хеш-функции на длину массива. Чтобы снизить вероятность попаданий двух ключей в одну позицию, размер массива, используемого для таблицы, обычно выбирают равным простому[44] числу.

Если при занесении нового элемента в хеш-таблицу вычисленная позиция оказалась пуста, новую запись заносят в эту позицию. Если при поиске ключа в хеш-таблице вычисленная позиция оказалась пуста, считается, что записи с таким ключом в таблице нет. Несколько интереснее вопрос о том, что делать, если позиция занята, но ключ записи в этой позиции отличается от ключа, который нам нужен; такая ситуация означает, что у двух разных ключей, несмотря на все наши старания, оказались одинаковые остатки от деления хеш-функции на длину массива. Это называется ***коллизией***. Основных методов разрешения коллизии известно два. Первый из них очень простой: в массиве хранятся не сами записи, а указатели, позволяющие сформировать *список* (обычный односвязный) из записей, ключи которых дают одно и то же значение хеша (точнее, не хеша, а его остатка от деления на длину массива). Способ, при его внешней простоте, имеет очень серьёзный

[44]Напомним на всякий случай, что простое число — это натуральное число, которое делится только на единицу и само на себя.

недостаток: списки сами по себе достаточно громоздки, работа с ними отнимает много времени.

Второй способ разрешения коллизий выглядит хитрее: если позиция в таблице занята записью, ключ которой не совпадает с искомым, используют *следующую* позицию, а если и она занята, то следующую за ней, и так далее. При занесении в таблицу новой записи её просто помещают в первую же найденную (после определённой с помощью хеш-функции) свободную позицию; при поиске просматривают записи одну за другой в поисках нужного ключа, и если при этом натыкаются на свободную запись, считается, что искомый ключ в таблице отсутствует. К недостаткам этого метода относится довольно неочевидный алгоритм *удаления* записи из таблицы. Проще всего поступить так: если позиции, непосредственно следующие за удаляемой записью, чем-то заняты, то найти первую свободную позицию таблицы, а затем последовательно каждую запись, расположенную в таблице между только что удалённой и найденной первой свободной, временно удалить из таблицы, а затем включить обратно по обычному правилу (при этом запись может оказаться включённой раньше либо в точности в том же месте). Ясно, что времени на это может уйти довольно много. В литературе можно найти более эффективные процедуры удаления записей, которые при этом существенно сложнее объяснить и, как следствие, можно сделать ошибку при реализации, обусловленную непониманием описания алгоритма.

Надо сказать, что оба способа (первый в меньшей степени, второй в бóльшей) чувствительны к заполнению таблицы. Считается, что хеш-таблица должна быть заполнена не более чем на две трети, в противном случае постоянно производимый линейный поиск (по спискам либо по самой таблице) сведёт на нет все преимущества хеширования. Если записей в таблице стало слишком много, её необходимо перестроить с использованием нового размера; например, можно увеличить текущий размер вдвое, после чего найти ближайшее сверху простое число и объявить новым размером таблицы. Перестроение таблицы — операция крайне неэффективная, ведь *каждую* запись из старой таблицы придётся внести в новую по обычному правилу с вычислением хеш-функции и взятием остатка от деления, и никаких «сокращённых» алгоритмов для этого нет.

Так или иначе, можно заметить, что для построения хеш-таблиц нужны массивы с заранее неизвестной — *динамически определяемой* — длиной. В оригинальный язык Паскаль такие средства не входили; современные диалекты, включая Free Pascal, поддерживают динамические массивы, но рассматривать их мы не будем; при желании вы можете освоить этот инструмент самостоятельно.

2.14. Ещё о рекурсии

Мы уже сталкивались с рекурсией, посвятив этому явлению § 2.4.5; к сожалению, на тот момент мы знали слишком мало и не могли разобрать примеры, в которых выразительная мощь рекурсии раскрылась бы в полной мере. Сейчас, умея работать со строками и списками, получив представление о деревьях и имея в целом больше опыта, мы вернёмся к обсуждению рекурсивного программирования и попытаемся дать более адекватное представление о его возможностях.

2.14.1. Взаимная рекурсия

Ранее мы упоминали о том, что рекурсия может быть *взаимной*; в простейшем случае одна подпрограмма вызывает другую, а вторая — снова первую. При этом возникает небольшая техническая проблема: одна из двух подпрограмм, участвующих во взаимной рекурсии, должна быть в программе описана раньше второй, но в ней должен (по условию) присутствовать вызов второй подпрограммы — то есть вызов подпрограммы, которая будет описана позже. Проблема тут в том, что компилятор пока что не знает имени второй подпрограммы и, как следствие, не позволяет это имя использовать, то есть вызов второй подпрограммы из первой приведёт к ошибке компиляции.

Для решения этой проблемы в Паскале введено понятие ***предварительного объявления*** (декларации) подпрограмм. В отличие от /подпрограмма!описание]описания подпрограммы, декларация только сообщает компилятору, что *позже* в тексте программы появится подпрограмма (процедура или функция) с таким-то именем, причём она будет иметь такие-то параметры, а для функций — такой-то тип возвращаемого значения. Больше никакой информации декларация компилятору не даёт, то есть в ней не пишется ни тело подпрограммы, ни секция локальных описаний. Вместо всего этого сразу после заголовка, сформированного обычным путём, ставится ключевое слово `forward` и точка с запятой, например:

```
procedure TraverseTree(var p: NodePtr); forward;
function CountValues(p: NodePtr; lim: longint): integer; forward;
```

Такие декларации позволяют не писать тело подпрограммы, к чему мы можем быть пока что не готовы, как правило, из-за того, что пока в программе описаны не все подпрограммы, вызываемые из этого тела. Декларация сообщает компилятору, во-первых, имя подпрограммы, и, во-вторых, всю информацию, необходимую для проверки корректности вызова таких подпрограмм и для генерации машинного кода, осуществляющего вызовы (это может потребовать преобразований типов выражений, так что типы параметров подпрограммы в точке её

вызова должны быть известны компилятору не только для проверки корректности).

При построении взаимной рекурсии мы сначала в программе приводим *декларацию* (со словом `forward`) первой из подпрограмм, участвующих в ней, затем описываем (полностью, с телом) вторую из этих подпрограмм, и уже затем приводим полное описание первой. При этом в обоих телах компилятору видны имена обеих подпрограмм и вся информация, нужная для их вызова, что нам, собственно, и требовалось.

Ясно, что задекларированная подпрограмма должна быть позже описана; если этого не сделать, при компиляции произойдёт ошибка.

2.14.2. Ханойские башни

Задачу о ханойских башнях мы рассматривали во вводной части книги при обсуждении понятия алгоритма (см. § 1.5.5). Тогда же мы пообещали продемонстрировать текст этой программы сначала на Паскале, а затем и на Си; сейчас самое время исполнить первую часть обещания.

Напомним, что в этой задаче даны три стержня, на один из которых надето N плоских дисков, различающихся по размеру, внизу самый большой диск, сверху самый маленький. За один ход можно переместить один диск с одного стержня на другой, при этом можно класть только меньший диск сверху на больший; на пустой стержень можно поместить любой диск. Брать сразу несколько дисков нельзя. Цель в том, чтобы за наименьшее возможное число ходов переместить все диски с одного стержня на другой, пользуясь третьим в качестве промежуточного.

Как мы уже обсуждали в § 1.5.5, самое простое и известное решение строится в виде рекурсивной процедуры переноса заданного количества n дисков с заданного исходного стержня на заданный целевой с использованием заданного промежуточного (его можно было бы вычислить, но проще получить его номер параметром процедуры); в качестве более простого случая выступает перенос $n-1$ дисков, а в качестве базиса рекурсии — вырожденный случай $n=0$, когда ничего никуда переносить уже не нужно.

В роли исходных данных для нашей программы выступает общее число дисков, которое мы для удобства получим в качестве параметра командной строки[45]. Рекурсивную реализацию алгоритма оформим в виде процедуры, которая для наглядности получает четыре параметра: номер исходного стержня (`source`), номер целевого стержня (`target`), номер промежуточного стержня (`transit`) и количество дисков (`n`). Саму процедуру назовём «`solve`». В ходе работы она будет печатать строки вроде «`1: 3 -> 2`» или «`7: 1 -> 3`», которые будут означать,

[45] Не будем уподобляться школьным учителям и читать это число с клавиатуры; это неудобно и попросту глупо.

соответственно, перенос диска № 1 с третьего стержня на второй и диска № 7 с первого на третий. Главная программа, преобразовав параметр командной строки в число[46], вызовет подпрограмму `solve` с параметрами `1, 3, 2, N`, в результате чего задача будет решена. Текст программы будет таким:

```
{ hanoi.pas }
program hanoi;

procedure solve(source, target, transit, n: integer);
begin
    if n = 0 then
        exit;
    solve(source, transit, target, n-1);
    writeln(n, ': ',  source, ' -> ', target);
    solve(transit, target, source, n-1);
end;

var
    n, code: integer;
begin
    if ParamCount < 1 then
    begin
        writeln(ErrOutput, 'No parameters given');
        halt(1)
    end;
    val(ParamStr(1), n, code);
    if (code <> 0) or (n < 1) then
    begin
        writeln(ErrOutput, 'Invalid token count');
        halt(1)
    end;
    solve(1, 3, 2, n);
end.
```

Отметим, что процедура, решающая задачу, состоит всего из восьми строк, три из которых служебные.

Попробуем теперь обойтись без рекурсии. Для начала припомним алгоритм, который мы приводили на стр. 149: на нечётных ходах самый маленький диск переносится «по кругу», причём при чётном общем количестве дисков — в «прямом» порядке, то есть $1 \to 2 \to 3 \to 1 \to \ldots$, а при нечётном — в «обратном» порядке, $1 \to 3 \to 2 \to 1 \to \ldots$; что касается чётных ходов, то на них мы не трогаем самый маленький диск, в результате чего ход оказывается определён однозначно.

Попытка реализовать этот алгоритм в виде программы наталкивается на неожиданное препятствие. Для человека действие вроде «посмот-

[46] Здесь мы для краткости воспользуемся встроенной процедурой `val`.

реть на стержни, найти самый маленький диск и, не трогая его, сделать единственный возможный ход» настолько просто, что мы выполняем такую инструкцию, ни на секунду не задумавшись; в программе же придётся помнить, на каком стержне находятся в настоящий момент какие диски, и выполнять множество сравнений, учитывая ещё и то, что стержни могут оказаться пустыми.

В архиве примеров к этой книге вы найдёте соответствующую программу в файле с именем `hanoi2.pas`. Здесь её текст мы не приводим с целью экономии места. Отметим только, что для хранения информации о расположении дисков на стержнях мы в этой программе воспользовались тремя односвязными списками, по одному на каждый стержень, причём в первый список в начале работы внесли в обратном порядке числа от n до 1, где n — количество дисков; для хранения указателей на первые элементы этих списков мы используем массив из трёх соответствующих указателей. Для решения задачи мы запускаем цикл, который выполняется до тех пор, пока хотя бы один «диск» (то есть элемент списка) присутствует в списках №1 и №2. Движение самого маленького диска мы на нечётных ходах просто вычисляем по формулам, для чего используем номер хода. В частности, номер стержня, *с которого* нужно забрать диск, при чётном общем количестве дисков вычисляется так:

```
src := (turn div 2) mod 3 + 1;
```

а при нечётном так (`turn` означает номер хода):

```
src := 3 - ((turn div 2 + 2) mod 3);
```

Гораздо сложнее обстоят дела с чётными ходами. Для вычисления такого хода нужно, во-первых, узнать, на каком из стержней располагается самый маленький диск, и исключить этот стержень из рассмотрения. Затем для оставшихся двух стержней нужно определить, в каком направлении между ними переносится диск, с учётом того, что один из них может оказаться пустым (и тогда диск переносится на него с другого стержня), либо они могут оба содержать диски, тогда меньший из дисков переносится на другой стержень, где находится больший. Тело подпрограммы, в которую вынесены эти действия, несмотря на их кажущуюся простоту, заняло 15 строк.

Общая длина программы составила 111 строк (против 27 для рекурсивного решения), если же мы отбросим пустые и служебные строки, а также и текст главной части программы, который в обоих случаях практически один и тот же (отличаются только параметры вызова процедуры `solve`), и посчитаем только значимые строки, реализующие решение, то в рекурсивном варианте таких строк было восемь (текст подпрограммы `solve`), тогда как в новом решении их получилось 87. Иначе говоря, решение стало сложнее более чем на порядок!

Теперь попробуем сделать решение без рекурсии, которое не использует вышеизложенный «хитрый» алгоритм. Вместо этого вспомним, что для перемещения всех дисков с одного стержня на другой нужно для начала переместить все диски, кроме самого большого, на промежуточный стержень, затем переместить самый большой диск на целевой стержень и, наконец, переместить все остальные стержни с промежуточного на целевой. Несмотря на то, что описание алгоритма очевидным образом рекурсивно, реализовать его можно без рекурсии; в принципе, это справедливо для любого алгоритма, то есть рекурсию всегда можно заменить циклом, вопрос лишь в том, насколько это сложно.

Трудность в том, что мы в каждый момент должны помнить, что́ мы сейчас куда переносим и с какой целью. Например, в процессе решения задачи для четырёх дисков мы в какой-то момент переносим первый (самый маленький) диск со второго стержня на третий, чтобы перенести два диска со второго стержня на первый, чтобы иметь возможность перенести третий диск со второго стержня на третий, чтобы перенести три диска со второго стержня на третий (поскольку четвёртый диск мы туда уже перенесли), чтобы перенести все четыре диска с первого на третий стержень. Это можно несколько удобнее расписать «с конца»:

- мы решаем задачу переноса четырёх дисков с первого стержня на третий, причём уже убрали все диски на второй стержень, перенесли четвёртый диск на третий стержень и теперь находимся в процессе переноса туда всех остальных дисков, т. е.
- решаем задачу переноса трёх дисков со второго стержня на третий, причём сейчас как раз пытаемся освободить третий диск, чтобы потом его перенести на третий стержень, для чего
- решаем задачу переноса двух дисков со второго стержня на первый, причём сейчас как раз пытаемся освободить второй диск, чтобы потом перенести его на первый стержень, для чего
- переносим первый диск со второго стержня на третий.

Очевидно, что здесь мы имеем дело с *задачами*, которые характеризуются *информацией* о том, сколько дисков мы хотели перенести, откуда, куда, а также *в каком состоянии у нас с этим дела.* Задач у нас переменное количество, так что придётся использовать какую-то динамическую структуру данных, проще всего, видимо, обычный список. Для каждой задачи придётся хранить количество дисков и два номера стержней, а также *состояние дел*, причём таких состояний можно выделить три:

1. мы ещё вообще ничего не делали; следующее действие в этом случае — расчистить самый большой из наших дисков, убрав все, которые меньше его, на промежуточный стержень;
2. мы уже расчистили самый большой диск, теперь его надо перенести, а затем перенести на него все диски, которые меньше его;

3. мы уже решили задачу, так что её можно убрать из списка.

В принципе, одно из этих состояний (последнее) можно было бы «сэкономить», но это пошло бы в ущерб ясности программы. Для обозначения состояний мы воспользуемся перечислимым типом на три возможных значения, обозначающих, что нужно делать следующим действием, когда мы снова видим данную задачу:

```
type
    TaskState = (StClearing, StLargest, StFinal);
```

Опишем тип для звеньев списка задач:

```
type
    ptask = ^task;
    task = record
        amount, src, dst: integer;
        state: TaskState;
        next: ptask;
    end;
```

и два указателя, рабочий и временный:

```
var
    first, tmp: ptask;
```

Для решения поставленной проблемы нам нужно перенести все диски (`n` штук) с первого стержня на третий. Оформим это в виде задачи и занесём получившуюся задачу в список (единственным элементом), учитывая, что мы ещё ничего не сделали, так что в качестве состояния задачи укажем `StClearing`:

```
    new(first);
    first^.amount := n;
    first^.src := 1;
    first^.dst := 3;
    first^.state := StClearing;
    first^.next := nil;
```

Далее организуется цикл, работающий до тех пор, пока список задач не опустеет. На каждом шаге цикла выполняемые действия зависят от того, в каком состоянии находится задача, стоящая в начале списка. Если она в состоянии `StClearing`, то в случае, когда эта задача требует переноса более чем одного диска, нужно добавить ещё одну задачу — для переноса $n-1$ дисков на промежуточный стержень; если текущая задача создана для переноса только одного диска, этого делать не надо. В обоих случаях сама текущая задача переводится в следующее

состояние, StLargest, то есть когда мы её увидим в следующий раз (а это произойдёт либо после переноса всех меньших дисков, либо, если диск всего один, то прямо на следующем шаге), нужно будет перенести самый большой из дисков, подразумеваемых данной задачей, и перейти к финальной стадии.

Если на вершине списка находится задача в состоянии StLargest, то первое, что мы делаем — это переносим самый большой из дисков, на которые данная задача распространяется; номер этого диска совпадает с их количеством в задаче. Под словами «переносим диск» мы в данном случае подразумеваем, что мы просто печатаем соответствующее сообщение. После этого, если дисков в задаче больше одного, нужно добавить новую задачу по переносу всех меньших дисков с промежуточного стержня на целевой; если диск всего один, этого делать не надо. В любом случае текущая задача переводится в состояние StFinal, чтобы, когда мы её увидим в начале списка в следующий раз, её можно было ликвидировать.

Очередная возникающая перед нами проблема — это вычисление номера промежуточного стержня; для этого мы предусмотрели в программе отдельную функцию:

```
function IntermRod(src, dst: integer) : integer;
begin
    if (src <> 1) and (dst <> 1) then
        IntermRod := 1
    else
    if (src <> 2) and (dst <> 2) then
        IntermRod := 2
    else
        IntermRod := 3;
end;
```

С использованием этой функции основной цикл «решения задач» выглядит так:

```
while first <> nil do
begin
    case first^.state of
        StClearing:
            begin
                first^.state := StLargest;
                if first^.amount > 1 then
                begin
                    new(tmp);
                    tmp^.src := first^.src;
                    tmp^.dst := IntermRod(first^.src, first^.dst);
                    tmp^.amount := first^.amount - 1;
                    tmp^.state := StClearing;
                    tmp^.next := first;
                    first := tmp
                end
            end;
```

```
        StLargest:
            begin
                first^.state := StFinal;
                writeln(first^.amount, ': ', first^.src,
                        ' -> ', first^.dst);
                if first^.amount > 1 then
                begin
                    new(tmp);
                    tmp^.src := IntermRod(first^.src, first^.dst);
                    tmp^.dst := first^.dst;
                    tmp^.amount := first^.amount - 1;
                    tmp^.state := StClearing;
                    tmp^.next := first;
                    first := tmp
                end
            end;
        StFinal:
            begin
                tmp := first;
                first := first^.next;
                dispose(tmp)
            end;
    end
end;
```

Полностью текст программы вы найдёте в файле `hanoi3.pas`; отметим, что его размер — 90 строк, из которых 70 — значащие строки реализации процедуры `solve` (включая вспомогательные подпрограммы). Получилось несколько лучше, чем в предыдущем случае, но всё равно в сравнении с рекурсивным решением разница почти на порядок.

2.14.3. Сопоставление с образцом

Рассмотрим такую задачу. Даны две строки символов, длина которых заранее неизвестна. Первую строку мы рассматриваем как *сопоставляемую*, вторую — как *образец*. В образце символ `'?'` может сопоставляться с произвольным символом, символ `'*'` — с произвольной *подцепочкой символов* (возможно, пустой), остальные символы обозначают сами себя и сопоставляются только сами с собой. Так, образцу `'abc'` соответствует только строка `'abc'`; образцу `'a?c'` соответствует любая строка из трёх символов, начинающаяся на `'a'` и заканчивающаяся на `'c'` (символ в середине может быть любым). Наконец, образцу `'a*'` соответствует любая строка, начинающаяся на `'a'`, ну а образцу `'*a*'` соответствует любая строка, содержащая букву `'a'` в любом месте. Необходимо определить, соответствует ли (целиком) заданная строка образцу.

Алгоритм такого сопоставления, если при этом можно использовать рекурсию, окажется достаточно простым. На каждом шаге мы рассматриваем *оставшуюся часть* строки и образца; сначала эти оставшиеся части совпадают со строкой и образцом, затем, по мере продвижения алгоритма, от них отбрасываются символы, стоящие в начале, и мы предполагаем, что для уже отброшенных символов сопоставление прошло успешно. Первое, что нужно сделать в начале каждого шага —

это проверить, не кончился ли у нас образец. Если он кончился, то результат зависит от того, кончилась ли при этом и строка тоже. Если кончилась, то мы фиксируем положительный результат сопоставления, если не кончилась — констатируем неудачу; действительно, с пустым образцом можно сопоставить только пустую строку.

Если образец ещё не кончился, проверяем, не находится ли в его начале (то есть в первом символе остатка образца) символ `'*'`. Если нет, то всё просто: мы производим сопоставление первых символов строки и образца; если первый символ образца не является символом `'?'` и не равен первому символу строки, то алгоритм на этом завершается, констатируя неудачу сопоставления, в противном случае считаем, что очередные символы образца и строки успешно сопоставлены, отбрасываем их (то есть укорачиваем остатки обеих строк спереди) и возвращаемся к началу алгоритма.

Самое интересное происходит, если на очередном шаге первый символ образца оказался символом `'*'`. В этом случае нам нужно последовательно перебрать возможности сопоставления этой «звёздочки» с пустой подцепочкой строки, с одним символом строки, с двумя символами и т. д., пока не кончится сама строка. Делаем мы это следующим образом. Заводим целочисленную переменную, которая будет у нас обозначать текущий рассматриваемый вариант. Присваиваем этой переменной ноль (начинаем рассмотрение с пустой цепочки). Теперь для каждой рассматриваемой альтернативы отбрасываем от образца один символ (звёздочку), а от строки — столько символов, какое сейчас число в нашей переменной. Полученные остатки *пытаемся сопоставить, используя для этого вызов той самой подпрограммы, которую мы сейчас пишем*, то есть производим рекурсивный вызов «самих себя». Если результат вызова — «истина», то мы на этом завершаемся, тоже вернув «истину». Если же результат — «ложь», то мы проверяем, можно ли ещё увеличивать переменную (не выйдем ли мы при этом за пределы сопоставляемой строки). Если увеличиваться уже некуда, завершаем работу, вернув «ложь». В противном случае возвращаемся к началу цикла и рассматриваем следующее возможное значение длины цепочки, сопоставляемой со «звёздочкой».

Весь этот алгоритм мы реализуем в виде функции, которая будет получать в качестве параметров две строки, сопоставляемую и образец, и два целых числа, показывающих, с какой позиции следует начинать рассмотрение той и другой строки. Возвращать наша функция будет значение типа `boolean`: `true`, если сопоставление удалось, и `false`, если не удалось. Принимая во внимание, что значения типа `string` занимают в памяти достаточно много места (256 байт) и что наша функция постоянно выполняет вызовы самой себя, строки мы ей будем передавать как var-параметры, избегая таким образом их копирования (см. рассуждение на эту тему на стр. 304). Функцию мы назовём `MatchIdx`, поскольку

она решает задачу сопоставления, но не просто над строками, а над «остатками» заданных строк, начиная с заданных *индексов*.

Исходную задачу будет решать другая функция, которую мы назовём просто `Match`; в неё будут передаваться только две строки, и всё. Тело этой функции будет состоять из вызова функции `MatchIdx` с теми же двумя строками и со значениями индексов, равными единице, что означает рассмотрение обеих строк с их начала. В этот раз мы опишем строковые параметры как обычные параметры-значения. Дело в том, что функция `Match` вызывается один раз, никаких рекурсивных вызовов для неё не предусмотрено, так что здесь копирование строк при вызове не столь страшно; с другой стороны, если её параметры объявить как параметры-переменные, то мы потеряем возможность вызывать эту функцию с параметрами, которые не являются переменными — например, не сможем задать образец в виде строкового литерала, а это часто бывает нужно. Используя параметры-значения для `Match`, мы можем для `MatchIdx` совершенно спокойно использовать параметры-переменные, ведь она будет вызываться только из `Match` — и, стало быть, работать с её локальными переменными. Иначе говоря, если нам понадобится сопоставить строку с образцом, мы обратимся к нашим функциям, они для себя сделают копии обеих строк, а дальше будут работать только с этими копиями.

Для тестирования добавим к нашей программе заголовок и главную часть, которая берёт строку и образец из параметров командной строки. Результат получится такой:

```
{ match_pt.pas }
program MatchPattern;

function MatchIdx(var str, pat: string; idxs, idxp: integer)
                                                    : boolean;
var
    i: integer;
begin
    while true do
    begin
        if idxp > length(pat) then
        begin
            MatchIdx := idxs > length(str);
            exit
        end;
        if pat[idxp] = '*' then
        begin
            for i:=0 to length(str)-idxs+1 do
                if MatchIdx(idxs+i, idxp+1) then
                begin
                    MatchIdx := true;
```

```
                    exit
                end;
            MatchIdx := false;
            exit
        end;
        if (idxs > length(str)) or
           ((str[idxs] <> pat[idxp]) and (pat[idxp] <> '?')) then
        begin
            MatchIdx := false;
            exit
        end;
        idxs := idxs + 1;
        idxp := idxp + 1;
    end
end;

function Match(var str, pat: string): boolean;
begin
    Match := MatchIdx(str, pat, 1, 1)
end;

begin
    if ParamCount < 3 then
    begin
        writeln(ErrOutput, 'Two parameters expected');
        halt(1)
    end;
    if Match(ParamStr(1), ParamStr(2)) then
        writeln('yes')
    else
        writeln('no')
end.
```

При работе с этой программой учтите, что символы «*» и «?» воспринимаются командным интерпретатором специфически: он тоже считает своим долгом произвести некое сопоставление с образцом (подробности см. в § 1.4.5). Самый простой и надёжный способ избежать неприятностей со спецсимволами — взять ваши параметры в апострофы, внутри них командный интерпретатор никакие подстановки не выполняет; сами апострофы при передаче вашей программе исчезнут, то есть вы не увидите их в составе строк, возвращённых функцией `ParamStr`. Сеанс работы с программой `match_pt` может выглядеть так:

```
avst@host:~/work$ ./match_pt 'abc' 'a?c'
yes
avst@host:~/work$ ./match_pt 'abc' 'a??c'
no
avst@host:~/work$ ./match_pt 'abc' '***a***c***'
```

```
yes
avst@host:~/work$
```

и так далее.

2.14.4. Рекурсия при работе со списками

Как уже упоминалось, односвязный список рекурсивен по своей сути: можно считать, что *список либо пустой, либо состоит из первого элемента и списка.* Это свойство можно использовать, обрабатывая односвязные списки с помощью рекурсивных подпрограмм, при этом почти всегда базисом рекурсии оказывается пустой список. Пусть, например, у нас имеется простейший список целых чисел, состоящий, как и в предыдущих примерах, из элементов типа `item`:

```
type
    itemptr = ^item;
    item = record
        data: integer;
        next: itemptr;
    end;
```

Начнём с простого подсчёта суммы чисел в таком списке. Конечно, можно пройти по списку циклом, как мы это уже неоднократно делали:

```
function ItemListSum(p: itemptr) : integer;
var
    sum: integer;
    tmp: itemptr;
begin
    tmp := p;
    sum := 0;
    while tmp <> nil do
    begin
        sum := sum + tmp^.data;
        tmp := tmp^.next
    end;
    ItemListSum := sum
end;
```

Теперь попробуем воспользоваться тем фактом, что сумма пустого списка равна нулю, а сумма непустого равна сумме его остатка, к которой прибавлен первый элемент. Новая (рекурсивная) реализация функции `ItemListSum` получится такой:

```
function ItemListSum(p: itemptr) : integer;
begin
```

```
    if p = nil then
        ItemListSum := 0
    else
        ItemListSum := p^.data + ItemListSum(p^.next)
end;
```

По правде говоря, если в вашем списке хранится несколько миллионов записей, то так лучше не делать, потому что может не хватить стека, но, с другой стороны, для хранения миллионов записей списки обычно не используются. Если же переполнение стека вам не угрожает, то удачные рекурсивные решения, как это ни странно, могут работать даже быстрее, чем «традиционные» циклы.

Рассмотрение примеров мы продолжим процедурой удаления всех элементов списка. Как это делается циклически, мы подробно рассмотрели ранее; теперь заметим, что для удаления пустого списка ничего делать не нужно, тогда как для удаления непустого списка следует освободить память от его первого элемента и удалить его остаток. Конечно, удаление остатка следует произвести первым, чтобы при удалении первого элемента не потерять указатель на этот самый остаток. Пишем:

```
procedure DisposeItemList(p: itemptr);
begin
    if p = nil then
        exit;
    DisposeItemList(p^.next);
    dispose(p)
end;
```

Нельзя сказать, чтобы мы здесь сильно выиграли в объёме кода, но то, что рекурсивное удаление списка проще прочитать, несомненно: для понимания здесь даже не требуется рисовать диаграммы.

Рассмотрим теперь пример посложнее. На стр. 378 мы рассматривали пример кода, вставляющего новый элемент в список целых чисел, отсортированный по возрастанию, с сохранением сортировки. Для циклического решения этой задачи нам потребовался указатель на указатель. Заметим теперь, что, коль скоро нам потребовалось вставить новый элемент в отсортированный список, у нас возможны три случая:

- список пустой — нужно вставить в него элемент первым;
- список непустой, причём его первый элемент больше, чем вставляемый — нужно вставить новый элемент в начало;
- список непустой, а первый элемент меньше или равен вставляемому — нужно произвести его вставку в остаток списка.

Как мы уже обсуждали, если указатель на первый элемент списка передать в подпрограмму как параметр-переменную, то такая подпрограмма сможет вставлять и удалять элементы в любом месте списка, в том числе и в начале. Кроме того, полезно будет припомнить, что вставка

элемента в пустой односвязный список и вставка элемента в начало непустого списка производятся абсолютно одинаково, что позволяет объединить два первых случая в один, который как раз и послужит базисом рекурсии. Заметим, что роль «указателя на первый элемент» для того списка, который является остатком исходного, играет поле `next` в первом элементе исходного списка. С учётом всего этого процедура, вставляющая новый элемент в отсортированный список с сохранением сортировки, будет выглядеть так:

```
procedure AddNumIntoSortedList(var p: itemptr; n: integer);
var
    tmp: itemptr;
begin
    if (p = nil) or (p^.data > n) then
    begin
        new(tmp);
        tmp^.data := n;
        tmp^.next := p;
        p := tmp
    end
    else
        AddNumIntoSortedList(p^.next, n)
end;
```

Сравните это с кодом, приведённым на стр. 378, и пояснениями к нему. Комментарии, как говорится, излишни.

2.14.5. Работа с двоичным деревом поиска

Обсуждая двоичные деревья в § 2.13.8, мы намеренно не приводили примеров кода для работы с ними, оставив этот вопрос до обсуждения рекурсивных методов работы. Если в случае с односвязными списками применение рекурсии способно *несколько облегчить* работу, то для деревьев речь должна идти скорее не о том, чтобы работу облегчить, а о том, чтобы сделать её *возможной*.

Для примера рассмотрим двоичное дерево поиска, содержащее целые числа типа `longint`, которое может состоять, например, из таких узлов:

```
type
    TreeNodePtr = ^TreeNode;
    TreeNode = record
        data: longint;
        left, right: TreeNodePtr;
    end;
```

Для работы с ним нам потребуется указатель, который часто называют *корнем дерева*; впрочем, корнем не менее часто называют начальный

узел сам по себе, а не указатель на него. Из контекста обычно легко понять, о чём идёт речь. Корневой указатель опишем так:

```
var
    root: TreeNodePtr = nil;
```

Теперь мы можем приступать к описанию основных действий с деревом. Поскольку нам придётся активно использовать рекурсию, мы каждое действие будем оформлять в виде подпрограммы. Чтобы приблизительно понять, как всё это будет выглядеть, для начала предположим, что дерево уже построено, и напишем функцию, которая вычисляет сумму значений во всех его узлах. Для этого заметим, что для пустого дерева такая сумма, очевидно, равна нулю; если же дерево непустое, то для подсчёта искомой суммы нужно будет вычислить для начала сумму левого поддерева, потом сумму правого поддерева, сложить их между собой и прибавить число из текущего узла. Поскольку левое и правое поддеревья представляют собой такие же деревья, как и дерево целиком, только с меньшим количеством узлов, мы можем для вычисления сумм поддеревьев воспользоваться той же функцией, которую пишем:

```
function SumTree(p: TreeNodePtr): longint;
begin
    if p = nil then
        SumTree := 0
    else
        SumTree :=
            SumTree(p^.left) + p^.data + SumTree(p^.right)
end;
```

Собственно говоря, практически всё, что мы делаем с деревом, строится по этой же схеме: в качестве вырожденного случая для базиса рекурсии используется пустое дерево, а дальше делаются рекурсивные вызовы для левого и/или правого поддерева. Продолжим наш набор примеров подпрограммой, добавляющей в дерево новый элемент; когда дерево пустое, необходимо создать элемент «прямо здесь», то есть изменить тот указатель, который служит для данного дерева корневым; с учётом этого указатель в процедуру будем передавать как параметр-переменную. Если же дерево не пусто, то есть, по крайней мере, существует его корневой элемент, то тогда возможны три случая. Во-первых, добавляемый элемент может быть строго меньше того, который находится в корневом элементе; тогда добавление следует произвести в левое поддерево. Во-вторых, он может быть строго больше, и тогда следует использовать правое поддерево. Третий случай портит нам всю картину, добавляя особый случай: числа могут оказаться равными. В зависимости от задачи в такой ситуации возможны различные варианты дальнейших действий: иногда при совпадении ключей выдают ошибку, иногда увеличивают

какой-нибудь счётчик, чтобы показать, что данный ключ заносился на один раз больше, иногда вообще ничего не делают. Мы остановимся на том, что сообщим вызывающему о невозможности выполнения добавления; для этого нашу подпрограмму оформим как функцию типа `boolean` и сделаем так, чтобы она возвращала «истину», когда значение успешно добавлено, и «ложь», когда возник конфликт ключей. Получится следующее:

```
function AddToTree(var p: TreeNodePtr; val: longint): boolean;
begin
    if p = nil then
    begin
        new(p);
        p^.data := val;
        p^.left := nil;
        p^.right := nil;
        AddToTree := true
    end
    else
    if val < p^.data then
        AddToTree := AddToTree(p^.left, val)
    else
    if val > p^.data then
        AddToTree := AddToTree(p^.right, val)
    else
        AddToTree := false
end;
```

Если попробовать написать функцию, определяющую, есть ли в данном дереве заданное число, получится нечто очень похожее:

```
function IsInTree(p: TreeNodePtr; val: longint): boolean;
begin
    if p = nil then
        IsInTree := false
    else
    if val < p^.data then
        IsInTree := IsInTree(p^.left, val)
    else
    if val > p^.data then
        IsInTree := IsInTree(p^.right, val)
    else
        IsInTree := true
end;
```

Схожесть двух подпрограмм не случайна, ведь в обоих случаях производится *поиск*. В принципе, для поиска нужной позиции можно написать одну обобщённую функцию, через которую затем описать и добавление,

и обычный поиск: такая функция будет по заданному дереву и значению искать *позицию* в дереве, где должен находиться (но не обязательно находится) узел с таким значением. Как уже можно догадаться, эта «позиция» есть не что иное, как *адрес* того *указателя*, который указывает или должен указывать на соответствующий узел. Поскольку этот адрес указателя будет служить возвращаемым значением функции, соответствующий тип значения нам придётся описать и снабдить именем:

```
type
    TreeNodePos = ^TreeNodePtr;
```

Отметим, во-первых, что такая функция должна в некоторых случаях возвращать адрес того указателя, который ей дали — если дерево оказалось пустым, а равно и в том случае, если корневой элемент дерева содержит искомое число. Для того, чтобы вернуть такой адрес, его надлежит по меньшей мере *знать*, и для этого мы указатель будем передавать как параметр-переменную; имя такого параметра, как мы помним, на время выполнения подпрограммы становится синонимом переменной, использованной в качестве параметра в точке вызова, так операция взятия адреса, применённая к имени параметра, как раз и даст нам адрес этой переменной. Во-вторых, случаи пустого дерева и равенства искомого значения значению в корневом узле теперь можно объединить: в случае равенства мы нашли позицию, где соответствующее число *находится*, а в случае пустого поддерерва — позицию, где оно *должно было бы находиться*. Различить эти два случая сможет вызывающий, проверив, чему равен указатель, расположенный по полученному из функции адресу: `nil` или нет. Текст функции получается такой:

```
function SearchTree(var p: TreeNodePtr; val: longint)
                                          : TreeNodePos;
begin
    if (p = nil) or (p^.data = val) then
        SearchTree := @p
    else
    if val < p^.data then
        SearchTree := SearchTree(p^.left, val)
    else
        SearchTree := SearchTree(p^.right, val)
end;
```

Используя эту функцию, две ранее написанные подпрограммы мы можем переписать по-новому, при этом они станут заметно короче:

```
function AddToTree(var p: TreeNodePtr; val: longint): boolean;
var
    pos: TreeNodePos;
begin
    pos := SearchTree(p, val);
    if pos^ = nil then
    begin
        new(pos^);
        pos^^.data := val;
        pos^^.left := nil;
        pos^^.right := nil;
        AddToTree := true
    end
    else
        AddToTree := false
end;

function IsInTree(p: TreeNodePtr; val: longint): boolean;
var
    pos: TreeNodePos;
begin
    pos := SearchTree(p, val);
    IsInTree := pos^ <> nil
end;
```

Небольшую демонстрационную программу, использующую эти функции, вы найдёте в файле `treedemo.pas`. К сожалению, мы вынуждены напомнить о серьёзном недостатке двоичных деревьев поиска: при неудачном порядке занесения в них хранимых значений дерево может получиться несбалансированным. Существует несколько подходов к такому построению двоичного дерева поиска, при котором разбалансировка либо вообще не возникает, либо оперативно устраняется, но рассказ о них выходит далеко за рамки нашей книги.

На случай возникновения у читателя непреодолимого желания освоить алгоритмы балансировки деревьев возьмём на себя смелость обратить его внимание на несколько книг, в которых соответствующие вопросы подробно разобраны. Начать стоит с учебника Н. Вирта «Алгоритмы и структуры данных» [8]; изложение в этой книге отличается лаконичностью и понятностью, поскольку рассчитано на начинающих. Если этого не хватило, попробуйте воспользоваться огромной по объёму книгой Кормена, Лейзерсона и Ривеста [9]; наконец, для сильных духом существует ещё и четырёхтомник Дональда Кнута, третий том которого [10] содержит подробный анализ всевозможных структур данных, ориентированных на сортировку и поиск.

В качестве упражнения рискнём предложить читателю попробовать написать подпрограммы для работы с деревьями без использования рекурсии. Отметим, что поиск в дереве при этом реализуется сравнительно просто и немного похож на вставку элемента в нужное место

односвязного списка, как это было показано в § 2.13.6. Гораздо сложнее оказывается алгоритм обхода дерева, который нужен в том числе для подсчёта суммы элементов списка; но и здесь мы уже успели рассмотреть нечто похожее, когда решали задачу о ханойских башнях (§ 2.14.2). В программе, обсуждение которой мы начали на стр. 393, нам приходилось запоминать, каким путём мы пришли к необходимости сделать то или иное действие, а затем возвращаться обратно и продолжать прерванную деятельность. При обходе дерева нужно, по сути, то же самое: для каждого узла, в который мы попали, мы вынуждены помнить то, *откуда* мы в него попали, а также то, что уже успели в этом узле сделать, или, точнее, что нам предстоит делать, когда мы снова вернёмся к этому узлу.

Если не получится — ничего страшного, эта задача довольно сложная; мы вернёмся к ней, когда будем изучать язык Си.

2.15. Ещё об оформлении программ

> Если бы строители строили дома так, как программисты пишут программы, то первый залетевший дятел разрушил бы цивилизацию.
>
> *Второй закон Вайнберга*

Мы уже неоднократно обращали внимание читателя на то, что текст программы нельзя писать как попало: он предназначается в первую очередь для прочтения человеком, и лишь во вторую — для компилятора, и если об этом забыть, то в вашей программе не только *другие* люди ничего не поймут, но и, что особенно обидно, вы рискуете *сами* заблудиться в собственной программе, не успев её дописать.

Для повышения понятности и читаемости программы существует ряд простых, но очень важных правил. В частности, мы уже знаем, что вложенные фрагменты следует сдвигать вправо относительно того, во что они вложены, а строки, начинающие и заканчивающие любую конструкцию, должны начинаться в одной и той же горизонтальной позиции. При этом размер сдвига вправо, называемого также структурным отступом, может составлять два пробела, три пробела, четыре пробела или один символ табуляции; вы можете выбрать любой из этих четырёх вариантов, но затем придерживаться избранного варианта во всей вашей программе. Заметим, что все примеры программ в этой книге набраны с использованием отступа в четыре пробела.

Мы также уже знаем, что существует три допустимых варианта расположения операторных скобок (для Паскаля это слова `begin` и `end`). Во всех программах на Паскале, приведённых в этой части книги,

```
while p <> nil do
begin
    s := s + p^.data;
    p := p^.next
end;
```

```
while p <> nil do begin
    s := s + p^.data;
    p := p^.next
end;
```

```
while p <> nil do
    begin
        s := s + p^.data;
        p := p^.next
    end;
```

Рис. 2.15. Три стиля расположения операторных скобок

мы всегда сносили слово `begin` на следующую строку после заголовка оператора (`while` или `if`), при этом писали его, начиная с той же позиции, где начинается сам заголовок, то есть мы не делали отдельного сдвига для операторных скобок (рис. 2.15, слева). Наряду с этим, допустимы ещё два стиля. Слово `begin` можно оставить на одной строке с заголовком оператора, при этом `end` располагается точно под началом оператора (**а не под словом `begin`!**), как это показано на том же рисунке в середине; именно так будут оформлены программы на Си в соответствующих частях нашей книги. Наконец, можно `begin` снести на следующую строку, но при этом предусмотреть для операторных скобок отдельный сдвиг (на рисунке этот вариант показан справа). При использовании последнего варианта обычно выбирают размер отступа в два пробела, потому что иначе горизонтальное пространство на экране очень быстро исчерпывается; как уже говорилось, этот вариант допустим, но рекомендовать его мы не станем.

2.15.1. Оператор `if` с веткой `else`

В Паскале оператор ветвления можно использовать как в полном варианте с веткой `else`, так и в сокращённом — без неё. Если ветка `else` отсутствует, то с оформлением текста конструкции всё понятно; никакой свободы нам не оставляет также и случай, когда обе ветки присутствуют, но состоят из одного простого оператора, то есть не требуют использования операторных скобок:

```
if p <> nil then
    res := p^.data
else
    res := 0;
```

Нужно только не забывать, что тело всегда следует размещать на отдельной строчке, и в случае оператора `if` это касается обеих ветвей. Так, следующий вариант недопустим:

```
if p <> nil then res := p^.data
else res := 0;
```

Рассмотрим теперь случай, когда *обе ветви оператора `if` используют составной оператор*. В этой ситуации нужно прежде всего вспомнить, какой из трёх допустимых вариантов размещения открывающей

операторной скобки мы в итоге выбрали. Если наш выбор пал на первый или третий вариант (в обоих случаях открывающая операторная скобка сносится на следующую строку), то всё достаточно просто. Если мы решили не сдвигать скобки составного оператора, а сдвигать только его тело, `if` придётся оформить так:

```
if p <> nil then
begin
    flag := true;
    x := p^.data
end
else
begin
    new(p);
    p^.data := x
end
```

Собственно говоря, именно так мы и оформляли ветвление в наших примерах. Если же вы, невзирая на все наши старания, избрали вариант со сдвигом операторных скобок, то `if` будет выглядеть так:

```
if p <> nil then
    begin
        flag := true;
        x := p^.data
    end
else
    begin
        new(p);
        p^.data := x
    end
```

В обоих случаях `else` вместе со скобками занимает три строки, что, по мнению многих программистов, многовато для разделителя. Если же открывающая операторная скобка оставляется на одной строке с заголовком, то в этом случае для ветки `else` также остаётся два варианта, а именно поместить слово `else` на одной строке с закрывающей операторной скобкой или же писать `else` с новой строки:

```
if p <> nil then begin
    flag := true;
    x = p^.data
end else begin
    new(p);
    p^.data := x
end
```

```
if p <> nil then begin
    flag := true;
    x := p^.data
end
else begin
    new(p);
    p^.data := x
end
```

Рассмотрим теперь случай, когда *только одна* ветка конструкции `if-else` требует использования составного оператора, тогда как вторая состоит из одного

простого оператора. Если принято решение сносить открывающую операторную скобку на следующую строку (не важно, со сдвигом или без), то этот случай не является особым и не требует отдельного рассмотрения. Можно было бы вообще не рассматривать этот случай в качестве специального, но результаты могут получиться несколько неэстетичные (в программах на Паскале это не столь заметно, но очень заметно, если мы пишем на Си). В связи с этим можно дать следующую рекомендацию: **если одна ветвь конструкции `if-else` представляет собой составной оператор, то для второй следует также использовать составной оператор, даже если она состоит из одного простого оператора**. Отметим, что следование этой рекомендации, вообще говоря, факультативно; больше того, как можно заметить, мы в наших примерах так не делали — но обязательно будем так делать, когда доберёмся до изучения Си.

2.15.2. Особенности оформления оператора выбора

Язык Паскаль не предполагает в составе оператора выбора (`case`) использование слова `begin`, хотя предполагает слово `end`, так что оформление заголовка и тела оператора выбора не зависит от того, оставляем ли мы открывающую операторную скобку на одной строке с заголовком сложного оператора, сносим ли мы её на следующую строку и снабжаем ли мы её своим собственным отступом; с другой стороны, в большинстве случаев в каждой ветке приходится использовать составные операторы, а их оформление уже зависит от избранного стиля.

В любом случае потребуется ответить ещё на один вопрос: *будете ли вы сдвигать метки относительно самого оператора*; **метки, обозначающие начало очередной альтернативы в операторе выбора, можно либо оставлять в той колонке, где начинается заголовок оператора выбора, либо сдвинуть относительно заголовка на размер отступа**. Как обычно в таких случаях, вы можете выбрать любой из двух вариантов, но только один раз — на всю программу.

Если мы оставляем операторную скобку на одной строке с заголовком, то в зависимости от выбранного стиля размещения меток мы можем написать так:

```
case Operation of
    '+': begin
        writeln('Addition');
        c := a + b;
    end;
    '-': begin
        writeln('Subtraction');
        c := a - b;
    end;
    else begin
        writeln('Error');
        c := 0;
    end
end
```

```
case Operation of
'+': begin
    writeln('Addition');
    c := a + b;
end;
'-': begin
    writeln('Subtraction');
    c := a - b;
end;
else begin
    writeln('Error');
    c := 0;
end
end
```

Отметим, что, хотя оба варианта являются допустимыми, первый выглядит существенно привлекательнее и понятнее, так что *если мы остав-*

ляем операторную скобку на одной строке с заголовком, будет лучше принять решение в пользу сдвигания меток.

Если мы условились сносить операторную скобку на следующую строку, но не сдвигать её, вышеприведённый фрагмент кода должен выглядеть так:

```
case Operation of
    '+':
    begin
        writeln('Addition');
        c := a + b;
    end;
    '-':
    begin
        writeln('Subtraction');
        c := a - b;
    end;
    else
    begin
        writeln('Error');
        c := 0;
    end
end
```

```
case Operation of
'+':
begin
    writeln('Addition');
    c := a + b;
end;
'-':
begin
    writeln('Subtraction');
    c := a - b;
end;
else
begin
    writeln('Error');
    c := 0;
end
end
```

Для этого случая мы также порекомендуем сдвигать метки (мы именно так и делали в наших примерах), но окончательное решение оставим за читателем.

Наконец, если мы не только сносим операторную скобку, но и сдвигаем её, то выглядеть это будет приблизительно так:

```
case Operation of
    '+':
        begin
            writeln('Addition');
            c := a + b;
        end;
    '-':
        begin
            writeln('Subtraction');
            c := a - b;
        end;
    else
        begin
            writeln('Error');
            c := 0;
        end
end
```

```
case Operation of
'+':
    begin
        writeln('Addition');
        c := a + b;
    end;
'-':
    begin
        writeln('Subtraction');
        c := a - b;
    end;
else
    begin
        writeln('Error');
        c := 0;
    end
end
```

Для этого случая наша рекомендация будет противоположной: если операторные скобки имеют свой собственный сдвиг, то метки лучше не сдвигать, результат будет эстетичнее.

2.15.3. Метки и оператор `goto`

Мы уже знаем из § 2.6.3, что `goto` использовать в некоторых случаях не только можно, но и нужно; более того, мы точно знаем, какие это

случаи; повторим ещё раз, этих случаев ровно два, и третьего нет. Остаётся ответить на вопрос, как всё это оформлять. Точнее, сам оператор `goto` никаких проблем с оформлением не вызывает, это обыкновенный оператор, оформляемый по обычным правилам; но вот то, как следует ставить *метку*, может стать предметом жаркой дискуссии.

Напомним, что меткой всегда помечается *оператор*, то есть, даже если мы пометим меткой «пустое место», компилятор будет считать, что там располагается пустой оператор. Вопросов, на которые нужно дать (зафиксировать) ответ, оказывается два: сдвигать ли метку относительно объемлющей конструкции (отметим, что с аналогичной дилеммой мы уже встречались при обсуждении операторов выбора в § 2.15.2) и размещать ли помеченный оператор на той же строке, где метка, или на отдельной.

Наиболее популярен вариант, при котором метка не сдвигается, то есть пишется в той же горизонтальной позиции, в которой размещены начало и конец объемлющей управляющей структуры, а помеченный оператор сносится на следующую строку, например:

```
procedure GoodProc;
label quit;
var p, q: ^SomethingBig;
begin
    new(p);
    new(q);
    { ... }
quit:
    dispose(q);
    dispose(p);
end;
```

Метка здесь оказалась в крайней левой позиции исключительно потому, что именно там размещается объемлющая структура (в данном случае это подпрограмма). Метка может встретиться и не на верхнем уровне, например:

```
if cond = 1 then begin
    while f <> 0 do begin
        while g <> 0 do begin
            while h <> 0 do begin
                if t = 0 then
                    goto eject
            end
        end;
    eject:
        writeln("go on...")
    end
end
```

Приведённый вариант наиболее популярен, но это не единственный *допустимый* вариант. Довольно часто оператор, помеченный меткой, пишут в той же строке, что и метку, примерно так:

```
    { ... }
quit: dispose(q);
    dispose(p);
end;
```

Этот вариант неплохо смотрится, если метка — вместе с двоеточием и пробелом после него — занимает по горизонтали меньше места, чем выбранный размер отступа, что позволяет выдержать горизонтальное выравнивание для операторов:

```
    { ... }
q:  dispose(q);
    dispose(p);
end;
```

Поскольку имя метки из одной буквы само по себе выглядит не слишком приятно, обычно такой стиль применяют в сочетании с отступами в табуляцию (максимальная длина имени метки при этом составляет 6 символов).

Некоторые программисты предпочитают рассматривать метку просто как часть помеченного оператора, не выделяя её как особую сущность; оператор сдвигают как обычно, но на этот раз уже вместе с меткой. Концовка вышеприведённых подпрограмм в таком стиле будет выглядеть так:

```
    { ... }
    quit: dispose(q);
    dispose(p);
end;
```

Иногда метку сдвигают, но помеченный оператор сносят на следующую строку, примерно так:

```
    { ... }
    quit:
    dispose(q);
    dispose(p);
end;
```

Основной недостаток таких решений — метка сливается с окружающим «пейзажем», перестаёт быть заметна в качестве особой точки в структуре кода; возьмём на себя смелость рекомендовать воздержаться от такого стиля, но, тем не менее, сохраним решение за читателем.

2.15.4. Максимальная ширина текста программы

Thou shalt not cross 80 columns in thy file[47].

(The sacred 80 column rule)

Традиционная в программировании ширина текста составляет 80 символов. Происхождение числа 80 восходит ко временам перфокарт; перфокарты наиболее популярного формата, предложенного фирмой IBM, содержали 80 колонок для пробивания отверстий, причём при использовании этих карт для представления текстовой информации каждая колонка задавала один символ. Одна перфокарта, таким образом, содержала строку текста до 80 символов длиной, и именно из таких строк состояли тексты компьютерных программ тех времён. Длина строки текста, равная 80 символам, ещё в начале 1990-х годов оставалась одним из стандартов для матричных принтеров. При появлении в начале 1970-х алфавитно-цифровых терминалов их ширина составила 80 знакомест, чтобы обеспечить совместимость двух принципиально различных способов ввода компьютерных программ. До сих пор многие компьютеры, оснащённые дисплеями, после включения питания начинают работу в текстовом режиме, и лишь после загрузки операционной системы переключаются в графический режим; ширина экрана в текстовом режиме в большинстве случаев составляет всё те же 80 знакомест.

Одним из традиционных способов управления компьютером остаётся хорошо знакомая нам командная строка; чаще всего для этого используются графические программы, *эмулирующие* алфавитно-цифровой терминал. Ширина строки в таких программах по умолчанию составляет, как несложно убедиться, 80 символов, хотя это обычно легко исправить, просто изменив размеры окна.

Было бы неверно полагать, что число 80 здесь абсолютно случайно. Если ограничение на длину строк сделать существенно меньшим, писать программы станет неудобно, в особенности когда речь идёт о структурированных языках, в которых необходимо использование структурных отступов; так, в ширину 40 символов не уложатся даже самые простенькие программы. С другой стороны, программы с существенно более длинными строками тяжело читать, даже если соответствующие строки помещаются на экране или листе бумаги. Причина здесь сугубо эргономическая и связана с необходимостью постоянно переводить взгляд влево-вправо; отметим, что в большинстве книг ширина полосы набора также не превышает 80 символов (обычно книжная строка вмещает 65–75 знаков). Традиция ограничивать длину строк текста

[47] Да не пересчёшь ты 80 столбцов в файле твоём // Священное правило восьмидесятого столбца (англ.)

80 символами насчитывает столь долгую историю[48] именно потому, что число 80 представляет собой удачный компромисс между малой вместительностью коротких строк и неудобочитаемостью длинных, и этот компромисс, что в данном случае важно, был известен издателям книг задолго до компьютерной эры.

При современном размере дисплеев, их графическом разрешении и возможности сидеть к ним близко без вреда для здоровья многие программисты не видят ничего плохого в редактировании текста при ширине окна, существенно превышающей 80 знакомест. С точки зрения эргономики такое решение не вполне удачно; целесообразно либо сделать шрифт крупнее, чтобы глаза меньше уставали, либо использовать ширину экрана для размещения нескольких окон с возможностью одновременного редактирования разных файлов — это сделает более удобной навигацию в вашем коде, ведь код сложных программ обычно состоит из множества файлов, и вносить изменения часто приходится одновременно в несколько из них. Отметим, что многие оконные текстовые редакторы, ориентированные на программирование, такие как `geany`, `gedit`, `kate` и т. п., штатно показывают на экране линию правой границы — как раз на уровне 80-го знакоместа.

Немалое число программистов предпочитает не распахивать окно текстового редактора шире, чем на 80 знакомест; более того, многие программисты пользуются редакторами текстов, работающими в эмуляторе терминала, такими, как vim или emacs; оба редактора имеют графические версии, но не всем программистам эти версии нравятся. Довольно часто в процессе эксплуатации программы возникает потребность просматривать и даже редактировать исходные тексты на *удалённой* машине, при этом качество связи (либо политика безопасности) может не позволять использование графики, и тогда окно алфавитно-цифрового терминала становится единственным доступным инструментом. Известны программные средства, предназначенные для работы с исходными текстами программ (например, выявляющие различия между двумя версиями одного и того же исходного текста), которые реализованы в предположении, что строки исходного текста не превышают 80 символов в длину.

Часто листинг программы бывает нужно напечатать на бумаге. Наличие длинных строк в такой ситуации поставит вас перед неприятным выбором. Можно заставить длинные строки умещаться на бумаге в одну строчку — либо уменьшив размер шрифта, либо используя более широкий лист бумаги или «пейзажную» ориентацию — но при этом

[48] 80-колоночные перфокарты были предложены IBM ещё в тридцатые годы XX в. — задолго до появления первых ЭВМ; они предназначались для сортирующих автоматов — табуляторов, которые использовались, в частности, для обработки статистической информации. Впрочем, ещё в XIX в. аналоги перфокарт применялись для управления ткацкими станками.

бóльшая часть площади листа бумаги останется пустой, а читать такой листинг будет труднее; если строки обрезать, попросту отбросив несколько правых позиций, есть риск упустить что-то важное; наконец, если заставить строки автоматически переноситься, читаемость полученного бумажного листинга будет хуже, нежели читаемость исходного текста с экрана, что уже совсем никуда не годится.

Вывод из всего вышесказанного напрашивается довольно очевидный: каким бы текстовым редактором вы ни пользовались, не следует допускать появления в программе строк, длина которых превосходит 80 символов. В действительности желательно всегда укладываться в 75 символов, что позволяет уместить в 80 символов не только строку исходного кода, но и её номер (четыре символа на номер, одно знакоместо на пробел между номером и собственно строкой, оставшиеся позиции на текст). Это позволит комфортно работать с вашим текстом, например, программисту, использующему редактор vim с включённой нумерацией строк; из такого исходного кода можно будет сформировать красивый и легко читаемый листинг с пронумерованными строками. Как уже говорилось, редакторы текстов, ориентированные на программирование, обычно поддерживают изображение правой границы и по умолчанию отображают её именно после 80-го знакоместа в строке; не пренебрегайте этим.

Некоторые руководства по стилю оформления кода допускают «в исключительных случаях» превышать предел длины строки. Например, стиль оформления, установленный для ядра ОС Linux, категорически запрещает разносить на несколько строк текстовые сообщения, и для этого случая говорится, что лучше будет, если строка исходного текста «вылезет» за установленную границу. Причина такого запрета вполне очевидна. Ядро Linux — программа крайне обширная, и ориентироваться в её исходных текстах довольно трудно. Часто в процессе эксплуатации возникает потребность узнать, какой именно фрагмент исходного текста стал причиной появления того или иного сообщения в системном журнале, и проще всего найти соответствующее место простым текстовым поиском, который, разумеется, не сработает, если сообщение, которое мы пытаемся найти, разнесено на несколько текстовых констант, находящихся в разных строках исходника.

Тем не менее, превышение допустимой длины строк остаётся нежелательным. В том же руководстве по оформлению кода для ядра Linux на этот счёт имеются дополнительные ограничения — так, за правой границей экрана не должно быть «ничего существенного», чтобы человек, бегло просматривающий программу и не видящий текста справа от границы, не пропустил какое-то важное её свойство. Чтобы определить, насколько допустим ваш случай, может потребоваться серьёзный опыт. Поэтому наилучшим вариантом будет всё же считать требование соблюдения 80-символьной границы жёстким, то есть не допускающим исключений; как показывает практика, с этим всегда можно справиться, удачно разбив выражение, сократив текстовое сообщение, уменьшив уровень вложенности путём вынесения частей алгоритма во вспомогательные подпрограммы.

Кроме стандартной ширины экрана, следует обратить внимание также и на его высоту. Как уже говорилось выше, подпрограммы следует по возможности делать небольшими, чтобы они помещались на экран по высоте; остаётся вопрос, какой следует предполагать эту «высоту экрана». Традиционный ответ на этот вопрос — 25 строк, хотя имеются и вариации (например, 24 строки). Предполагать, что экран будет больше, не следует; впрочем, как уже говорилось, длина подпрограммы в некоторых случаях имеет право слегка превышать высоту экрана, но не намного.

2.15.5. Как разбить длинную строку

Коль скоро мы решили не выходить за 80 колонок, логично возникает вопрос: что же делать, если очередная строка программы никак не хочет в этот лимит укладываться. Начнём с обсуждения случая, когда слишком длинным оказался *заголовок оператора* (`if`, `while` или `case`).

Прежде всего следует подумать, нельзя ли подсократить условное выражение. Во многих случаях длинные выражения в заголовках структурных операторов возникают в силу недостатка опыта программиста; так, автор часто видел в студенческих программах проверку принадлежности символа к тому или иному множеству (например, множеству знаков препинания) через последовательность явно прописанных сравнений с каждым из элементов множества, что-то вроде:

```
if (a = ',') or (a = '.') or (a = ';') or (a == ':') || (a ...
```

Разумеется, здесь никакой проблемы с недостатком места в строке на самом деле нет, есть лишь проблема недостаточности воображения. Программист, миновавший стадию новичка, в такой ситуации опишет *функцию*, проверяющую принадлежность заданного символа предопределённому множеству, а заголовок `if`'а будет содержать вызов этой функции:

```
if IsPunctuation(a) then
```

Более опытный программист воспользуется готовой функцией из стандартной библиотеки, а ещё более опытный, возможно, заявит, что стандартная функция чрезмерно сложна, поскольку зависит от настроек локали в окружении, и вернётся к варианту со своей собственной функцией. Так или иначе, никакой проблемы с длиной заголовка тут нет.

К сожалению, не всегда проблемы решаются так просто. Многострочные заголовки, как бы мы ни пытались их побороть, всё равно иногда в программе возникают. Однозначного ответа, как с ними поступить, к сожалению, нет; мы рассмотрим один из возможных вариантов, который нам представляется наиболее практичным и отвечающим поставленной задаче читаемости программы.

Итак, **если заголовок сложного оператора приходится разнести на несколько строк, то:**

- **разбейте выражение в заголовке на несколько строк; предпочтительно разрывать строку по «операции верхнего уровня», это обычно логическая связка «и» либо «или»;**
- **каждую последующую строку заголовка сдвиньте относительно первой строки заголовка на обычный размер отступа;**
- **вне зависимости от количества простых операторов в теле, обязательно возьмите тело вашего оператора в операторные скобки, то есть сделайте его составным оператором;**
- **вне зависимости от используемого стиля, снесите открывающую операторную скобку на следующую строку, чтобы она послужила зрительным разделителем между строками заголовка и строками тела вашего оператора.**

Всё вместе будет выглядеть примерно так:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ') do
begin
    SkipSpace(TheCollection)
end;
```

Такой вариант нормально сработает, если вы не сдвигаете составной оператор относительно заголовка; если же вы предпочли именно этот («третий») стиль оформления, можно посоветовать снести на следующую строку слово `then`, `do` или `of`, примерно так:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ')
do
    begin
        SkipSpace(TheCollection)
    end;
```

Роль зрительного разделителя здесь играет завершающее слово из заголовка.

Если вы используете стиль, при котором открывающая операторная скобка оставляется на одной строке с заголовком, вы можете воспользоваться ещё одним вариантом форматирования: как в предыдущем примере, снесите последнюю лексему заголовка на отдельную строку, и на этой же строке оставьте открывающую операторную скобку:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ')
do begin
    SkipSpace(TheCollection)
end;
```

Такой стиль нравится не всем, но вы и не обязаны ему следовать; даже если везде вы оставляете операторную скобку на строке заголовка, для многострочного заголовка вы вполне можете сделать исключение и оформлять его так, как показано в первом примере этого параграфа.

Рассмотрим другие ситуации, когда строка может не поместиться в отведённое горизонтальное пространство. Хотелось бы сразу заметить, что лучший способ справиться с такими ситуациями — это *не допускать* их. Часто руководства по стилю оформления кода пишутся в предположении, что программист всегда может избежать нежелательной ситуации, и там попросту не говорится о том, что́ же нужно делать, если ситуация всё-таки сложилась; такое умолчание приводит к тому, что программисты начинают выходить из положения как попало, причём часто даже разными способами в рамках одной программы. Во избежание этого мы приведём несколько примеров того, как *можно* действовать, если длинная строка никак не хочет становиться короче.

Допустим, слишком длинное выражение встретилось в правой части присваивания. Первое, что мы посоветуем — это **попытаться разбить строку по знаку присваивания**. Если слева от присваивания стоит что-то достаточно длинное, такой вариант вполне может помочь, например:

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) + AnotherStrangeFunction(z);
```

Обратите внимание, что выражение, стоящее справа от присваивания, мы не просто снесли на следующую строку, но и сдвинули вправо на размер отступа. Если после этого выражение по-прежнему не помещается на экран, можно начать разбивать и его, и делать это лучше всего по знакам операций наименьшего приоритета, например:

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / SecondSillyCoeff +
    JustAVariable;
```

Может получиться так, что даже после этого экран останется слишком узким для вашего выражения. Тогда можно попытаться начать разбивать на несколько строк подвыражения, входящие в ваше выражение; их части нужно, в свою очередь, сдвинуть ещё на один отступ, чтобы

выражение более-менее легко читалось, насколько вообще возможно для такого монструозного выражения говорить о лёгкости прочтения:

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) +
    AnotherStrangeFunction(z) *
        FunctionWhichReturnsCoeff(z) *
        AnotherSillyFunction(z) +
    JustAVariable;
```

Если выражение состоит из большого количества подвыражений верхнего уровня (например, слагаемых), которые сами по себе не очень длинны, вполне допустимо оставлять несколько таких подвыражений в одной строке:

```
MyArray[f(x)].ThePtr^.MyField :=
    a + b + c + d + e + f + g + h + i + j + k + l + m +
    n + o + p + q + r + s + t + u + v + w + x + y + z;
```

Конечно, если в реальности вам пришлось вот так сложить 26 переменных, то это повод задуматься, почему вы не используете массив; здесь мы приводим сумму простых переменных исключительно для иллюстрации, в реальной жизни вместо переменных у вас будет что-то более сложное.

Отдельного обсуждения заслуживает ситуация, когда слева от присваивания стоит простое имя переменной или даже выражение, но короткое, так что разбивка строки по знаку присваивания не даёт никакого (или почти никакого) выигрыша. Естественно, выражение справа от присваивания по-прежнему лучше всего разбивать по операциям низшего приоритета; вопрос лишь в том, с какой позиции начинать каждую следующую строку. Ответов на этот вопрос ровно два: каждую следующую строку можно либо сдвигать на один отступ, как в примерах выше, либо размещать её начало точно под началом выражения в первой строке нашего присваивания (сразу после знака присваивания). Сравните, вот пример первого варианта:

```
MyArray[n] := StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / AnotherCoeff +
    JustAVariable;
```

А вот так тот же код будет выглядеть, если выбрать второй вариант:

```
MyArray[n] := StrangeFunction(p, q, r) * SillyCoeff +
              AnotherStrangeFunction(z) / AnotherCoeff +
              JustAVariable;
```

Оба варианта допустимы, но имеют существенные недостатки. Первый вариант заметно проигрывает второму в ясности, но второй вариант требует для второй и последующих строк нестандартного размера отступа, который оказывается зависящим от длины выражения слева от присваивания. Отметим, что этот (второй) вариант совершенно не годится, если вы используете в качестве отступа табуляцию, потому что добиться такого выравнивания можно только пробелами, а смешивать пробелы и табуляции не следует ни в коем случае.

Если недостатки обоих вариантов кажутся вам существенными, вы можете взять себе за правило *всегда* переводить строку после знака присваивания, если весь оператор целиком не поместился на одну строку. Такой вариант (рассмотренный в начале параграфа) свободен от обоих недостатков, но требует использования лишней строки; впрочем, запас строк во Вселенной неограничен. Выглядеть это будет так:

```
MyArray[n] :=
    StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / AnotherCoeff +
    JustAVariable;
```

Следующий случай, требующий рассмотрения — это слишком длинный вызов подпрограммы. Если при обращении к процедуре или функции у вас не помещаются в одну строку параметры, то строку, естественно, придётся разорвать, и обычно это делают после очередной запятой, отделяющей параметры друг от друга. Как и в случае с разбросанным по нескольким строкам выражением, возникает вопрос, с какой позиции начать вторую и последующие строки, и вариантов тоже два: сдвинуть их либо на размер отступа, либо так, чтобы все параметры оказались записаны «в столбик». Первый вариант выглядит примерно так:

```
VeryGoodProcedure("This is the first parameter",
    "Another parameter", YetAnotherParameter,
    More + Parameters * ToCome);
```

Второй вариант для приведённого примера будет выглядеть так:

```
VeryGoodProcedure("This is the first parameter",
                  "Another parameter",
                  YetAnotherParameter,
                  More + Parameters * ToCome);
```

Отметим, что это вариант, как и аналогичный вариант форматирования выражений, не годится при использовании табуляции в качестве размера отступа: добиться такого выравнивания можно только пробелами, а смешивать пробелы и табуляции не следует.

Если оба варианта вам по тем или иным причинам не понравились, мы можем предложить ещё один вариант, который используется

очень редко, хотя и выглядит вполне логичным: рассматривать имя подпрограммы и круглые скобки в качестве объемлющей конструкции, а параметры — в качестве вложенных элементов. В этом случае наш пример будет выглядеть так:

```
VeryGoodProcedure(
    "This is the first parameter",
    "Another parameter", YetAnotherParameter,
    More + Parameters * ToCome
);
```

Часто бывает так, что слишком длинным оказывается заголовок подпрограммы. В этой ситуации следует прежде всего внимательно рассмотреть возможности его сокращения, при этом допуская, в числе прочего, вариант с изменением разбивки кода на подпрограммы. Как уже говорилось, подпрограммами с шестью и более параметрами очень тяжело пользоваться, так что, если причиной «распухания» заголовка оказалось большое количество параметров, следует подумать, нельзя ли так изменить вашу архитектуру, чтобы это количество сократить (возможно, ценой появления бóльшего количества подпрограмм).

Следующее, на что нужно обратить внимание — это имена (идентификаторы) параметров. Поскольку эти имена *локальны для вашей подпрограммы*, их вполне можно сделать короткими, вплоть до двух-трёх букв. Конечно, при этом мы лишаем эти имена самопоясняющей силы, но заголовок подпрограммы в любом случае обычно снабжают комментарием, хотя бы коротким, и соответствующие пояснения о смысле каждого параметра можно вынести в этот комментарий.

Бывает, что даже после всех этих ухищрений заголовок по-прежнему не помещается в 79 знакомест. Скорее всего, придётся разнести на разные строки список параметров, но прежде чем это делать, стоит попытаться убрать на отдельные строки начало и конец заголовка. Так, можно написать на отдельной строке слово `procedure` или `function` (следующая строка при этом *не сдвигается*!). Кроме того, тип возвращаемого значения функции, указанный в конце заголовка, также можно снести на отдельную строку вместе с двоеточием, но эту строку следует сдвинуть так, чтобы тип возвращаемого значения оказался где-то под концом списка параметров (даже если вы используете табуляции). Дело в том, что читатель вашей программы именно там (где-то справа) ожидает увидеть тип возвращаемого значения, и на то, чтобы отыскать его на следующей строчке слева, а не справа, читателю придётся потратить лишние усилия. Всё вместе может выглядеть примерно так:

```
procedure
VeryGoodProcedure(fpar: integer; spar: MyBestPtr; str: string);
begin
```

```
    {...}
end;

function
VeryGoodFunction(fpar: integer; spar: MyBestPtr; str: string)
                                                : ExcellentRecordPtr;
begin
    {...}
end;
```

Если это не помогло и заголовок по-прежнему слишком длинный, остаётся только один вариант — разбить на части список параметров. Естественно, переводы строк вставляются между описаниями отдельных параметров. Если при этом несколько параметров имеют один тип и перечислены через запятую, желательно оставить их на одной строке, а разрывы строк помещать после точки с запятой, стоящей после имени типа. В любом случае, остаётся вопрос относительно горизонтального размещения (сдвига) второй и последующих строк. Как и для рассмотренных выше случаев длинного выражения и длинного вызова подпрограммы, здесь есть три варианта. Во-первых, можно начать список параметров на одной строке с именем подпрограммы, а последующие строки сдвинуть на размер отступа. Во-вторых, можно начать список на одной строке с именем подпрограммы, а последующие строки сдвинуть так, чтобы все описания параметров начинались в одной и той же позиции (этот случай не годится, когда для форматирования используется табуляция). Наконец, можно, рассматривая имя подпрограммы и открывающую скобку как заголовок сложной структуры, снести описание первого параметра на следующую строку, сдвинув его на размер отступа, остальные параметры разместить под ним, а круглую скобку, закрывающую список параметров, разместить на отдельной строке в первой позиции (под началом заголовка).

Несколько особняком стоит в нашем списке случай длинной строковой константы. Конечно, самое *худшее*, что можно сделать — это «заэкранировать» символ перевода строки, продолжив строковый литерал в начале следующей строчки кода. **Не делайте так никогда**:

```
        writeln('This is a string which unfortunately is \
too long to fit on a single code line');
```

В Паскале благодаря операции сложения строк можно сделать так:

```
        writeln('This is a string which unfortunately is ' +
                'too long to fit on a single code line');
```

Но и это не вполне правильно. Единое текстовое сообщение, выдаваемое как одна строка (т. е. не содержащее символов перевода строки среди

выдаваемого текста), вообще лучше не разносить на разные строки кода (см. замечание на стр. 416). Остаётся попробовать ещё два способа борьбы с длиной строкового литерала.

Во-первых, как это ни банально, стоит подумать, нельзя ли сократить содержащуюся в строке фразу без потери смысла. Как известно, краткость — сестра таланта. Например, для рассматриваемого примера возможен такой вариант:

```
writeln('String too long to fit on a line');
```

Смысл английской фразы мы оставили прежним, но теперь она, вопреки собственному содержанию, вполне нормально помещается в строке кода.

Во-вторых (если сокращать ничего не хочется), можно заметить, что некоторые строковые константы уместились бы в строке кода, если бы содержащий их оператор начинался в крайней левой позиции, т. е. если бы не структурный отступ. В такой ситуации справиться с упрямой константой совсем легко: достаточно дать ей имя — например, описать в виде константы:

```
const TheLongString =
    "This string could be too long if it was placed in the code";
{ ... }

        writeln(TheLongString);
```

К сожалению, бывает и так, что ни один из перечисленных способов не помогает. Тогда остаётся лишь последовать правилам из Linux Kernel Coding Style Guide и оставить в коде строку, длина которой превышает 80 символов. Следите только, чтобы это превышение не выходило за грань разумного. Так, если получившаяся строка кода «вылезла» за 100 символов, и при этом вам кажется, что ни одним из вышеперечисленных способов побороть зловредную константу нельзя, то это вам, скорее всего, только кажется; автор этих строк ни разу за всю свою практику не видел ситуации, в которой строковую константу нельзя было бы уместить в обычные 80 символов, не говоря уже о ста.

2.15.6. Пробелы и разделители

Знаки, которые в тексте программы выделяются в отдельные лексемы независимо от наличия или отсутствия вокруг них пробельных символов, называются *разделителями*. Обычно это знаки арифметических операций, скобки и знаки препинания, такие как запятая, точка с запятой и двоеточие. Например, операции `+` и `-` являются разделителями, поскольку можно написать `a + b`, а можно и `a+b`, смысл от этого не изменится. В то же время операция `and` разделителем не является: `a and b` — это не то же самое, что `aandb`.

Несмотря на то, что пробельные символы вокруг разделителей не обязательны, в ряде случаев их добавление может сделать текст программы более эстетичным и читаемым — *но не всегда*. При этом, как водится, невозможно предложить единый универсальный свод правил по поводу расстановки таких пробелов; существуют различные подходы, имеющие свои достоинства и недостатки. С уверенностью можно сказать, что **для знаков препинания — запятых, точек с запятой и двоеточий** — лучше всего подходит одно простое правило: **перед ними пробелы не ставятся, а после них — наоборот, ставятся** (это может быть как собственно пробел, так и перевод строки).

Некоторые программисты ставят пробелы с внутренней стороны скобок (круглых, квадратных, фигурных и угловых), примерно так:

```
MyProcedure( a, b[ idx + 5 ], c );
```

Мы не рекомендуем так делать, хотя это и допустимо; лучше написать так:

```
MyProcedure(a, b[idx + 5], c);
```

При обращении к процедурам и функциям **пробел между именем вызываемой подпрограммы и открывающей скобкой обычно не ставят**, так же как и пробел между именем массива и открывающей квадратной скобкой операции индексирования.

Несколько особняком стоит вопрос о том, какие из *арифметических операций* следует выделять пробелами, и как (с одной стороны или с обеих). Одна из самых популярных и при этом внятных рекомендаций звучит так: **символы бинарных операций выделяются пробелами с обеих сторон, символы унарных операций пробелами не выделяются**. При этом следует учитывать, что операции выборки поля из сложной переменной (точка в Паскале и Си, «стрелка» в Си) бинарными не являются, поскольку справа у них не операнд, а название поля, которое не может быть значением выражения. Подчеркнём, что такой стиль является наиболее популярным, но никоим образом не единственным; возможно следование совсем другим правилам, например — в любом выражении выделять пробелами бинарные операции наименьшего приоритета (то есть операции самого «верхнего» уровня), а остальные пробелами не выделять, и т. д.

2.15.7. Выбор имён (идентификаторов)

Общее правило при выборе имён достаточно очевидно: **идентификаторы следует выбирать в соответствии с тем, для чего они используются**. Некоторые авторы утверждают, что идентификаторы всегда обязаны быть осмысленными и состоять из нескольких слов.

На самом деле, это не всегда так: если переменная исполняет сугубо локальную задачу и её применение ограничено несколькими строчками программы, имя такой переменной вполне может состоять из одной буквы. В частности, целочисленную переменную, играющую роль переменной цикла, чаще всего называют просто «`i`», и в этом нет ничего плохого. Но однобуквенные переменные уместны только тогда, когда из контекста однозначно (и без дополнительных усилий на анализ кода) понятно, что это такое и зачем оно нужно, ну и ещё, пожалуй, разве что в тех редких случаях, когда переменная содержит некую физическую величину, традиционно обозначаемую именно такой буквой — например, температуру вполне можно хранить в переменной `t`, а пространственные координаты — в переменных `x`, `y` и `z`. Указатель можно назвать `p` или `ptr`, строку — `str`, переменную для временного хранения какого-то значения — `tmp`; переменную, значение которой будет результатом вычисления функции, часто называют `result` или просто `res`, для сумматора вполне подойдёт лаконичное `sum` и так далее.

Важно понимать, что подобные лаконичности подходят лишь для локальных идентификаторов, то есть таких, область видимости которых ограничена — например, одной подпрограммой. Если же идентификатор виден во всей программе, он просто обязан быть длинным и понятным — хотя бы для того, чтобы не возникало конфликтов с идентификаторами из других подсистем. Чтобы понять, о чём идёт речь, представьте себе программу, над которой работают два программиста, и один из них имеет дело с температурным датчиком, а другой — с часами; как температура, так и время традиционно обозначаются буквой `t`, но если наши программисты воспользуются этим обстоятельством для именования глобально видимых объектов, то проблем не миновать: программа, в которой есть две разные глобальные переменные с одним и тем же именем, не имеет шансов пройти этап компоновки.

Более того, когда речь идёт о глобально видимых идентификаторах, сама по себе длина и многословность ещё не гарантирует отсутствия проблем. Допустим, нам потребовалось написать функцию, которая опрашивает датчик температуры и возвращает полученное значение; если мы назовём её `GetTemperature`, то формально вроде бы всё в порядке, на самом же деле с очень хорошей вероятностью нам в другой подсистеме потребуется узнать температуру, ранее записанную в файл или просто хранящуюся где-то в памяти программы, и для такого действия тоже вполне подойдёт идентификатор `GetTemperature`. К сожалению, не существует универсального рецепта, как избежать таких конфликтов, но кое-что посоветовать всё же можно: **выбирая имя для глобально видимого объекта, подумайте, не могло бы такое имя обозначать что-то другое**. В рассматриваемом примере для идентификатора `GetTemperature` можно сразу предложить две-три альтернативные роли, так что его следует признать неудачным. Более удачным мог бы быть, на-

пример, идентификатор `ScanTemperatureSensor`, но лишь в том случае, если он используется для работы со *всеми* температурными датчиками, с которыми имеет дело ваша программа — например, если такой датчик заведомо единственный, либо если функция `ScanTemperatureSensor` получает на вход номер или другой идентификатор датчика. Если же ваша функция предназначена для измерения, к примеру, температуры в салоне автомобиля, причём существует ещё и датчик, скажем, температуры охлаждающей жидкости в двигателе, то в имя функции следует добавить ещё одно слово, чтобы полученное имя идентифицировало происходящее однозначно, например: `ScanCabinTemperatureSensor`.

2.15.8. Регистр букв, ключевые слова и идентификаторы

Одна из особенностей Паскаля — его принципиальная нечувствительность к регистру букв: один и тот же идентификатор можно написать как `myname`, `MYNAME`, `MyName`, `mYnAmE` и так далее. То же самое касается и ключевых слов: `begin`, `Begin`, `BEGIN`, `bEgIn`... компилятор всё стерпит.

Тем не менее, наиболее распространено мнение, что **ключевые слова следует писать в нижнем регистре (т. е. маленькими буквами)** и не придумывать на эту тему ничего лишнего. Из этого правила можно сделать одно вполне логичное исключение: написать большими буквами те `BEGIN` и `END`, которые обрамляют главную часть программы.

Иногда можно встретить программы на Паскале, в которых ключевые слова написаны с большой буквы: `Begin`, `End`, `If` и т. д.; попадаются также программы, в которых с большой буквы написаны только названия управляющих операторов (`If`, `While`, `For`, `Repeat`), а все прочие, включая `begin` и `end`, пишутся в нижнем регистре. Всё это допустимо, хотя и экзотично; необходимо только чётко сформулировать для себя правила, в каких случаях как мы пишем ключевые слова, и неукоснительно следовать им во всей программе.

Совсем редко можно встретить текст, где все ключевые слова набраны заглавными буквами. Как показывает практика, такой текст читается тяжелее; следовательно, так писать не надо. Ну и, естественно, не надо прибегать к изыскам вроде `BeGiN` или, скажем, `FunctioN` — такое тоже встречается, но относится к области бессмысленного позёрства.

Что касается выбора имён для идентификаторов, то в Паскале сложились определённые традиции на эту тему. Если имя переменной состоит из одной буквы или представляет собой короткую аббревиатуру (что вполне допустимо для локальных переменных, см. § 2.15.7), имя такой переменной обычно пишут маленькими буквами: `i`, `j`, `t`, `tmp`, `res`, `cnt` и т. п. Если же имя переменной (а равно имя типа, процедуры или функции, константы, метки и т. д.) состоит из нескольких *слов*, то эти слова записывают слитно, начиная каждое с большой буквы:

`StrangeGlobalVariable`, `ListPtr`, `UserListItem`, `ExcellentFunction`, `KillThemAll`, `ProduceSomeCompleteMess` и т. п. Остаётся вопрос, как быть с именами, представляющими собой *одно слово*; мы в нашей книге писали их в нижнем регистре (`counter`, `flag`, `item`), но многие программисты предпочитают писать их с большой буквы (`Counter`, `Flag`, `Item`); иногда короткие имена переменных и типов пишут с маленькой буквы, а имена функций и процедур — с большой. Как обычно в таких случаях, выбор за вами.

2.15.9. Как справиться с секциями описаний

Стандарт Паскаля требует жесткого порядка следования секций описаний, но, к счастью, существующие реализации этому требованию не следуют, позволяя располагать секции описаний в произвольном порядке и создавать больше одной секции любого типа — в частности, не обязательно ограничиваться только одной секцией описания переменных.

Всё это даёт возможность отличать настоящие глобальные переменные от переменных, которые нужны в главной части программы, но к которым при этом не нужен доступ из подпрограмм. Например, если в главной части присутствует цикл и для него нужна целочисленная *переменная цикла*, то, за неимением лучшего, описывать такую переменную придётся в секции описаний переменных, относящейся ко всей программе, однако совершенно ясно, что никакого отношения к глобальным переменным она не имеет. В связи с этим будет лучше, если **всё, что нужно в главной части программы и только в ней — переменные, метки, иногда типы, которые не используются нигде, кроме главной части — следует описать непосредственно перед словом `begin`, обозначающим начало главной части, т. е. после всех описаний подпрограмм**. Если вам нужны настоящие глобальные переменные — то есть такие, доступ к которым осуществляется из более чем одной подпрограммы, либо из главной программы и подпрограммы — то для их описания следует создать ещё одну секцию `var`, на этот раз *перед* описаниями подпрограмм.

Заметим, что переход между частями программы по меткам невозможен, так что **метки, используемые в подпрограммах, должны в них же и описываться**, тогда как метки, описываемые в глобальной секции описания меток, должны предназначаться для главной программы. Следовательно, **если в главной программе используются метки, секция их описаний должна находиться непосредственно перед началом главной программы**.

2.16. Тестирование и отладка

2.16.1. Отладка в жизни программиста

К настоящему моменту ваши знания достаточны, чтобы написать программу заметного объёма; если вы хотя бы раз пробовали это сделать, то уже знаете, что свеженаписанная программа почти никогда не работает так, как от неё ожидается; требуется долгий кропотливый процесс поиска ошибок и их исправления, который называется ***отладкой***.

Для начала мы попытаемся сформулировать ряд базовых принципов, связанных с отладкой, и сделаем это в такой форме, что они, возможно, покажутся вам шуткой; вы вскоре сами убедитесь, что в этой шутке доля шутки совсем незначительна, а всё остальное — самая настоящая лютая правда. Итак:

- ошибка всегда есть;
- ошибка всегда не там;
- если вы точно знаете, где ошибка, то у ошибки может оказаться другое мнение;
- если вы считаете, что программа должна работать, то самое время вспомнить, что «должен» — это когда взял взаймы и не отдал;
- если отладка — это процесс исправления ошибок, то написание программы — это процесс их внесения;
- сразу после обнаружения ошибки дело всегда выглядит безнадёжным;
- найденная ошибка всегда кажется глупой;
- чем безнадёжнее всё выглядело, тем глупее кажется найденная ошибка;
- компьютер делает не то, чего вы хотите, а то, о чём вы попросили;
- корректная программа работает правильно в любых условиях, некорректная — тоже иногда работает;
- и лучше бы она не работала;
- если программа работает, то это ещё ничего не значит;
- если программа «свалилась», надо радоваться: ошибка себя проявила, значит, её теперь можно найти;
- чем громче грохот и ярче спецэффекты при «падении» программы, тем лучше — заметную ошибку искать гораздо проще;
- если ошибка в программе точно есть, а программа всё-таки работает, вам не повезло — это самый противный случай;
- ни компилятор, ни библиотека, ни операционная система ни в чём не виноваты;
- никто не хочет вашей смерти, но если что — никто не расстроится;
- на самом деле всё совсем не так плохо — всё гораздо хуже;

- первая написанная строчка текста будущей программы делает этап отладки неизбежным;
- если вы не готовы к отладке — не начинайте программировать;
- компьютер не взорвётся; но большего вам никто не обещал.

В компьютерных классах часто (особенно на контрольных работах и зачётах) можно наблюдать студентов, которые, написав некий текст на языке программирования, добившись с горем пополам успешной компиляции и убедившись, что полученный результат никак не отвечает поставленной задаче, на этом какую-либо конструктивную деятельность прекращают, переключаясь, например, на другую задачу — как правило, с аналогичным результатом на выходе. Столь странный выбор стратегии они обычно объясняют сакраментальной фразой «ну, я её написал, а она не работает», причём произносится с интонацией, подразумевающей, что во всём виновата сама программа, а ещё преподаватель, компьютер, погода в Африке, посол Аргентины в Швеции или буфетчица из студенческой столовой, но уж точно не говорящий — ведь он же *написал* программу.

В такой ситуации следует сразу же вспомнить один простой принцип: компьютер делает ровно то, что написано в программе. Этот факт кажется банальным, но из него немедленно следует второй, а именно: если программа работает неправильно, то она неправильно написана. С учётом этого утверждение «я написал программу» требует уточнения: правильнее будет сказать «я написал неправильную программу».

Ясно, что написание *неправильной* программы, пусть даже успешно проходящей компиляцию, уж точно не является сколько-нибудь достойным внимания делом; в конце концов, простейший текст на Паскале, успешно проходящий компиляцию, состоит всего из двух слов и одной точки: «`begin end.`» Конечно, эта программа не решает поставленную задачу — но ведь и та, которую «написали, а она не работает», тоже ничего не решает, и чем же она в таком случае лучше?

Не менее типична и другая ситуация, тоже возникающая преимущественно на контрольных работах и выражающаяся фразой «я всё написал, а отладить времени не хватило». Проблема здесь в наполнении слова «всё»: авторы таких программ часто даже не подозревают, насколько на самом деле они были далеки от решения задачи.

Ощущения новичка, с трудом осилившего написание текста программы и обнаружившего, что программа совершенно не желает соответствовать его ожиданиям, даже можно понять. Сам процесс написания программы, обычно называемый «кодированием», новичку всё ещё кажется делом очень трудным, так что подсознательно автор такой программы ожидает хоть какого-нибудь вознаграждения за «успешно» преодолённые трудности, а то, что делает в итоге компьютер, скорее напоминает не вознаграждение, а издевательство.

Опытные программисты воспринимают всё это совершенно иначе. Во-первых, они точно знают, что кодирование, то есть сам процесс написания текста программы — это лишь небольшая часть разнообразной деятельности, называемой программированием, и не просто небольшая, но вдобавок ещё и самая лёгкая. Во-вторых, имея опыт создания программ, программист хорошо понимает, что в момент, когда текст программы наконец успешно прошёл компиляцию, ничего не заканчивается, а наоборот — начинается самая трудоёмкая фаза создания программы, именуемая, как мы уже догадались, отладкой. Отладка отнимает больше сил, чем кодирование, требует существенно более изощрённых навыков, а главное — по времени может тянуться дольше в несколько раз, и это ничуть не преувеличение.

Будучи психологически готовым к отладке, программист рационально рассчитывает свои силы и время, так что ошибки, обнаруженные при первом запуске программы, его не обескураживают: *так и должно быть!* Если мало-мальски сложная программа при первых запусках не проявляет ошибок — это довольно странно. Проблема новичка может оказаться в том, что, забыв о предстоящей отладке, он всё своё время и силы потратил на написание первой версии текста; когда дело доходит до самого интересного, ни сил, ни времени уже нет.

Альпинисты, штурмующие серьёзные горы, твёрдо следуют одному важнейшему принципу: цель восхождения не в том, чтобы достичь вершины, а в том, чтобы вернуться обратно. Те, кто про этот принцип забывают, зачастую, достигнув вожделенной вершины, потом гибнут на спуске. Конечно, с программированием всё не столь жестоко — во всяком случае, гибель вам здесь не грозит; но если ваша цель — написать программу, которая делает, что от неё требуется, то вам нужна готовность потратить две трети времени и сил именно на отладку, а не на что-то другое.

Избежать отладки невозможно, но соблюдение некоторых достаточно простых правил может её изрядно облегчить. Итак, самое главное: **старайтесь проверять работу отдельных частей программы по мере их написания**. Здесь на вас работают сразу два закона: во-первых, только что написанный код отлаживать гораздо проще, чем тот, который вы уже успели подзабыть; во-вторых, с ростом объёма текста, который нужно отлаживать, сложность отладки растёт нелинейно. Кстати, из этого правила вытекает довольно очевидное следствие: **старайтесь разбивать программу на подпрограммы так, чтобы они как можно меньше зависели друг от друга**; помимо других выгод, этот подход позволит упростить раздельное тестирование частей вашей программы. Можно предложить ещё более общий принцип: **когда вы пишете код, думайте о том, как вы будете его отлаживать**. Отладка не прощает безалаберности, проявленной на стадии кодирования; даже такое «безобидное» нарушение стиля оформления,

как короткое тело оператора ветвления или цикла, оставленное на одной строчке с заголовком оператора, может вам стоить изрядного количества впустую испорченных нервов.

Второе правило мы уже видели в начале параграфа в виде простой и ёмкой фразы: **ошибка всегда не там**. Если у вас возникла «интуитивная уверенность», что к тому эффекту, который вы наблюдаете, может привести, конечно же, только ошибка вот в этой процедуре, вот в этом цикле, вот в этом фрагменте — не верьте своей интуиции. В целом интуиция — прекрасная штука, но во время отладки программ она не работает. Объясняется это очень просто: если бы ваша интуиция чего-то стоила против данной конкретной ошибки, то вы бы эту ошибку не допустили. Итак, прежде чем пытаться исправить тот или иной фрагмент, вам необходимо **объективно** (а не «интуитивно») убедиться в том, что ошибка находится именно здесь. Помните: в таком месте программы, где вы знаете, что можно ошибиться, вы скорее всего не ошибётесь; напротив, самые заковыристые ошибки оказываются как раз там, где вы никак не могли их ожидать; между прочим, именно поэтому они там и оказываются.

К объективным методам локализации ошибок относятся отладочная печать и пошаговое выполнение под управлением программы-отладчика; если вы надеетесь без них обойтись, лучше вообще не начинайте писать программы; и здесь мы подходим к третьему правилу: **метод пристального взгляда в ходе отладки практически не работает**. Сколько бы вы ни пялились в свой текст, результат будет выражаться фразой «вроде всё правильно, почему же она не работает?!» Опять же, причина здесь достаточно очевидна: если бы вы могли увидеть собственную ошибку, вы бы не ошиблись. Можно назвать ещё два соображения в пользу неэффективности «пристального взгляда»: наиболее внимательно вы будете просматривать те фрагменты своего кода, где *ожидаете* обнаружить ошибку, а она, как мы уже знаем, наверняка не там; плюс к тому здесь включается такое хорошо известное явление, как «замыленность взгляда» — даже глядя прямо на ту строчку программы, в которой допущена ошибка, вы вряд ли ошибку заметите. Причину этого эффекта тоже несложно понять: вы сами только что написали этот фрагмент кода, задействовав некие соображения, которые кажутся вам правильными, и, как следствие, сам фрагмент кода продолжает вам казаться правильным, даже если на самом деле содержит ошибку. Итак, **не тяните резину и не тратьте драгоценное время на «внимательное изучение» собственного кода: отлаживать программу всё равно придётся**.

Отметим здесь ещё один момент: переписывание программы заново — в целом дело хорошее, но избежать отладки это вам не поможет; скорее всего, вы просто снова наделаете ошибок в тех же местах. Что касается переписывания заново отдельных фрагментов программы, то с этим всё

ещё хуже: ошибка наверняка окажется не в том фрагменте, который вы решите переписать.

Следующее правило звучит так: **не надейтесь, что ошибка где-то вне вашей программы**. Конечно, компилятор и операционная система — это тоже программы, и в них тоже есть ошибки, но все *простые* ошибки там уже повылавливали сотни тысяч других пользователей. Вероятность нарваться на неизвестную ошибку в системном программном обеспечении гораздо ниже, чем шанс выиграть джекпот в какой-нибудь лотерее. Пока ваши программы не превышают нескольких сотен строк, вы можете считать, что у них просто-напросто *не та весовая категория*: чтобы проявить ошибку в том же компиляторе, нужно что-то существенно более хитрое. Ну а к тому времени, когда ваши программы станут достаточно сложными, вы сами поймёте, что попытки свалить вину на компилятор и операционку выглядят довольно нелепо.

Ещё один момент, который стоит учитывать: **если вы сами не можете найти ошибку в своей программе, то никто другой её тем более не найдёт**. Студенты часто задают один и тот же вопрос: «а почему моя программа не работает?» Ваш покорный слуга, слыша этот вопрос, обычно, в свою очередь, спрашивает, за кого его изволят держать: за экстрасенса, за телепата или за ясновидящего. Разобраться в вашей программе стороннему человеку в большинстве случаев сложнее, чем написать аналогичную программу с нуля. Кроме того, это же *ваша* программа; сами наворотили, сами разбирайтесь. Интересно, что в подавляющем большинстве случаев студент, задающий такой вопрос, даже не пытался ничего сделать для отладки своей программы.

Закончить этот параграф мы попробуем в позитивном ключе. Отладка — дело неизбежное и очень тяжёлое, но, как это часто бывает, процесс отладки оказывается занятием весьма увлекательным, даже азартным. Некоторые программисты заявляют, что программа-отладчик — это их любимая компьютерная игра, поскольку никакая стратегия, никакой пасьянс, никакие аркады не дают такого разнообразия головоломок и пищи для мозгов, никакие леталки и стрелялки не приводят к выделению такого количества адреналина, как процесс отладки, и никакие успешные прохождения квестов не приносят такого удовлетворения, как успешно найденная и изничтоженная ошибка в программе. Как несложно догадаться, вопрос тут исключительно в вашем личном отношении к происходящему: постарайтесь воспринимать это как игру с ошибкой в «кто кого», и вы увидите, что даже этот аспект программирования способен доставлять удовольствие.

2.16.2. Тесты

Если вы обнаружили, что ваша программа содержит ошибку — значит, вы по меньшей мере один раз *запустили* её и, скорее всего, подали

ей на вход какие-то данные; впрочем, последнее не обязательно, в некоторых случаях программы «падают» сразу после старта, не успев ничего прочитать. Так или иначе, вы уже приступили к ***тестированию*** вашей программы; об организации этого дела тоже стоит сказать пару слов.

Начинающие, как правило, «тестируют» свои программы очень просто: запускают, набирают какие-то входные данные и смотрят, что получится. Такой подход на самом деле никуда не годится, но это, увы, становится понятно не сразу и не всем; автору приходилось встречать *профессиональные команды программистов*, в которых даже есть специально нанятые *тестировщики*, и они с утра до вечера делают именно это: гоняют тестируемую программу и так и этак, вколачивая разнообразные данные в те или иные формы ввода, нажимая на кнопочки и совершая другие телодвижения в надежде рано или поздно наткнуться на какую-нибудь несообразность. Когда программисты вносят в программу изменения, работа таких «тестировщиков» начинается с самого начала, ведь, как известно, при любых изменениях сломаться может что угодно.

Аналогичную картину можно наблюдать в компьютерных классах: студенты при каждом запуске своих программ набирают входные данные на клавиатуре, так что один и тот же текст набирается по десять, по двадцать, по сорок раз. При виде такой картины сам собой возникает вопрос, ну когда же этому студенту станет хоть чуть-чуть лень заниматься подобной ерундой.

Чтобы понять, в чём такой студент неправ и как действовать правильно, вспомним, что *поток стандартного ввода* не обязательно связан с клавиатурой; при запуске программы мы можем сами решить, откуда она будет читать информацию. Мы уже использовали это, причём речь шла о тестировании очень простых программ (см. стр. 216); даже для программы, которой на вход требуется подать всего одно целое число, мы предпочли не вводить это число каждый раз, а воспользовались командой `echo`. Конечно, число всё-таки приходится набирать, когда мы формируем команду, но сама команда остаётся в истории, которую для нас запоминает командный интерпретатор, так что *второй* раз нам то же число набирать не нужно: вместо этого мы воспользуемся «стрелкой вверх» или поиском через `Ctrl-R` (см. § 1.4.6), чтобы повторить команду, которую уже вводили.

Конечно, использовать возможности командного интерпретатора для хранения и повторного прогона тестовых примеров можно разве что в самых простых случаях; если подходить к делу *правильно*, то для проверки работы программы следует создать ***набор тестов***, представленный в каком-то объективном виде — как правило, в виде файла или нескольких файлов.

Под ***тестом*** понимается, и это очень важно, *вся информация*, которая нужна, чтобы запустить программу или какую-то её часть, подать на вход такие данные, которые проявляют тот или иной аспект её функционирования, проверить результат и выдать вердикт о том, правильно всё отработало или нет. Тест может состоять из одних только данных — например, в одном файле мы можем сформировать данные, которые программе следует подать на вход, в другом файле — то, что мы ожидаем получить на выходе. Более сложный тест может включать в себя специальный тестовый *программный код* — именно так приходится действовать при тестировании отдельных частей программы, например, отдельных её подпрограмм. Наконец, сложные тесты оформляют в виде целых *программ* — таких, которые сами запускают тестируемую программу, подают ей на вход те или иные данные и проверяют результаты.

Допустим, мы решили написать программу для сокращения простых дробей: на вход ей подаются два целых числа, означающие числитель и знаменатель, в качестве результата она тоже печатает два числа — числитель и знаменатель той же дроби, приведённой к простейшему возможному виду.

Когда программа будет написана и пройдёт компиляцию, дальнейшие действия начинающего программиста, никогда не задумывавшегося о правильной организации тестирования, могут выглядеть так:

```
newbie@host:~/work$ ./frcancel
25 15
5 3
newbie@host:~/work$ ./frcancel
7 12
7 12
newbie@host:~/work$ ./frcancel
100 2000
1 20
newbie@host:~/work$
```

В большинстве случаев начинающие на этом успокаиваются, решив, что программа «правильная», но задача сокращения дроби не столь проста, как кажется на первый взгляд. Если программу написать «в лоб», она, скорее всего, не будет работать для отрицательных чисел. Об этом нашему начинающему может сказать его более опытный товарищ или, если дело происходит на занятиях в классе — преподаватель; попробовав запустить свою программу и подать ей на вход что-нибудь «с минусом», её автор может убедиться, что старшие товарищи правы и программа, к примеру, «зациклилась» (именно это произойдёт с простейшей реализацией алгоритма Евклида, которая не учитывает особенностей работы операции `mod` для отрицательных операндов). Конечно, исправить программу особых проблем не составляет, важнее другое: любые

исправления могут «сломать» то, что до этого работало, так что *новую версию программы придётся тестировать с самого начала*, то есть тестовые запуски, которые уже были проделаны, придётся повторить, каждый раз вводя числа с клавиатуры.

Как уже, скорее всего, догадался читатель, тестирование в исполнении более опытного программиста могло бы выглядеть так:

```
advanced@host:~/work$ echo 25 15 | ./frcancel
5 3
advanced@host:~/work$ echo 7 12 | ./frcancel
7 12
advanced@host:~/work$ echo 100 2000 | ./frcancel
1 20
advanced@host:~/work$
```

Этот подход, несомненно, лучше предыдущего, но и он пока ещё далёк от полноценного тестирования, ведь для повторения каждого из тестов его нужно будет найти в истории вручную, а после запуска тратить время, проверяя, правильные числа напечатаны или нет.

Чтобы понять, как правильно организовать тестирование программы, давайте для начала заметим, что каждый тест у нас состоит из четырёх чисел: два из них программе подаются на вход, а два других нужны, чтобы сравнить с ними напечатанный программой результат. Такой тест можно записать в одну строчку; так, три теста из нашего примера выражаются строками

```
25 15 5 3
7 12 7 12
100 2000 1 20
```

Осталось придумать какой-нибудь механизм, который, имея набор тестов в таком виде, сам, без нашего участия запустит тестируемую программу нужное число раз, подаст ей на вход тестовые данные и проверит правильность результатов. В нашей ситуации самый простой способ добиться этого — написать *скрипт* на языке командного интерпретатора; если вы не помните, как это делается, перечитайте § 1.4.13.

Чтобы понять, как наш скрипт будет выглядеть, представим себе, что четыре числа, составляющих тест, располагаются в переменных `$a`, `$b`, `$c` и `$d`. «Прогнать» тест можно командой «`echo $a $b | ./frcancel`»; но нам нужно не просто запустить программу, а сравнить результат с ожидаемым, для чего результат нужно тоже поместить в переменную. Для этого можно воспользоваться присваиванием и «обратными апострофами», которые, как мы помним, подставляют вместо себя результат выполнения команды:

```
res=‘echo $a $b | ./frcancel‘
```

Результат, попавший в переменную `$res`, можно сравнить с ожидаемым, и если обнаружено несовпадение, сообщить об этом пользователю:

```
    if [ x"$c $d" != x"$res" ]; then
        echo TEST $a $b FAILED: expected "$c $d", got "$res"
    fi
```

Последовательно «загнать» в переменные `$a`, `$b`, `$c` и `$d` числа из тестов нам поможет встроенная в интерпретатор команда `read`; в качестве параметров эта команда принимает имена переменных (без знака «`$`»), читает из своего потока ввода строку, разбивает её на слова и «раскладывает» эти слова по заданным переменным, причём если слов оказалось больше, то в последнюю переменную попадёт весь остаток строки, состоящий из всех «лишних» слов. Команда `read` обладает полезным свойством: если очередную строку прочитать удалось, она завершается успешно, а если поток кончился — неуспешно. Это позволяет с её помощью организовать цикл `while` примерно так:

```
while read a b c d; do
       # здесь будет тело цикла
done
```

В теле такого цикла переменные `$a`, `$b`, `$c` и `$d` последовательно принимают своими значениями первое, второе, третье и четвёртое слово из очередной строки. Отметим, что каждый наш тест как раз представляет собой строку из четырёх слов (в роли слов выступают числа, но они ведь тоже состоят из символов; в командно-скриптовых языках нет ничего, кроме строк). В теле цикла мы поместим приведённый выше `if`, прогоняющий отдельный тест, и останется только придумать, как подать получившейся конструкции на стандартный ввод последовательность наших тестов. Для этого можно воспользоваться перенаправлением вида «документ здесь», которое делается с помощью знака «`<<`». После этого знака ставится некоторое слово («стоп-слово»), а затем пишется текст, который следует подать на вход команде, и завершается этот текст строкой, состоящей целиком из стоп-слова.

Всё вместе будет выглядеть примерно так:

```
#!/bin/bash
#                frcancel_test.sh

while read a b c d ; do
    res=‘echo $a $b | ./frcancel‘
    if [ x"$c $d" != x"$res" ]; then
        echo TEST $a $b FAILED: expected "$c $d", got "$res"
    fi
done <<END
```

```
    25 15 5 3
    7 12 7 12
    100 2000 1 20
END
```

Несмотря на свою примитивность, это уже самый настоящий полноценный комплект тестов с возможностью автоматического прогона. Как можно заметить, добавление нового теста сводится к вписыванию ещё одной строчки перед `END`, но это не главное; важнее то, что прогон всех тестов не требует от нас никаких усилий, мы просто запускаем скрипт и смотрим, выдаст ли он что-нибудь. Если он ничего не выдал — прогон прошёл успешно. Общее правило, которому мы фактически уже последовали, звучит так: **тест может быть трудно написать, но должно быть легко прогнать**.

Дело тут вот в чём. Тесты нужны не только для того, чтобы попытаться выявить ошибки сразу после написания программы, но и для того, чтобы после каждого исправления можно было до какой-то разумной степени уверить себя, что мы, внося изменения, ничего не сломали. Поэтому отладку никогда не следует считать оконченной, а тесты ни в коем случае не надо выбрасывать (например, стирать) — они ещё не раз пригодятся; и, конечно же, необходимо расчитывать на то, что после любых мало-мальски заметных изменений в программе нам нужно будет подвергнуть её проверке на всех имеющихся у нас тестах, а делать это вручную, понятное дело, несколько накладно.

Если очередной тест выявил ошибку в программе, не торопитесь бросаться что-то исправлять. Для начала стоит подумать, нельзя ли *упростить* тест, на котором проявилась ошибка, то есть нельзя ли написать тест, проявляющий ту же ошибку, но при этом более простой. Конечно, это не значит, что более сложный тест при этом нужно выбросить. Тесты вообще не надо выбрасывать. Но чем тест проще, тем меньше факторов, которые могут повлиять на работу программы, и тем легче будет найти ошибку. Последовательно упрощая один тест, вы можете создать целое семейство тестов, причём, возможно, самые простые из них уже не будут проявлять ошибку. Это не повод останавливаться: попробуйте взять самый простой из тестов, который всё ещё «ошибается», и упростить его в каком-нибудь другом направлении. В любом случае все созданные тесты, вне зависимости от того, проявляют ли они прямо сейчас какую-то ошибку или нет, ценны для дальнейшей отладки: одни могут указать путь поиска ошибки, другие — показать, где ошибку искать точно не надо.

Существует особый подход к написанию программ, который называется *test first* — на русский это можно приблизительно перевести как «тест сначала». При этом подходе сначала пишут тесты, запускают их, убеждаются, что они не работают, а затем пишут текст программы так, чтобы заставить тесты заработать. Если программисту начинает казаться, что программа всё равно написана не

так, как должна быть, ему нужно сначала написать новый тест, который, не сработав, тем самым даст некое объективное подтверждение «неправильности» программы, и лишь затем изменить программу так, чтобы проходил и новый тест, и все старые. Написание программного текста, предназначенного для чего-то иного, кроме удовлетворения имеющихся тестов, полностью исключается.

При этом подходе тесты в основном пишут не для программы целиком, а для каждой её процедуры и функции, для подсистем, включающих несколько связанных между собой процедур и т. д. Следование принципу «test first» позволяет править программу смелее, не боясь её испортить: если мы что-то испортим, об этом нам скажут переставшие работать тесты, если же что-то испортится, но ни один тест при этом не перестанет работать — значит, тестовое покрытие недостаточно и нужно написать больше тестов.

Конечно, вы не обязаны следовать этому подходу, но знать о его существовании и иметь в виду возможность его применения будет как минимум полезно.

2.16.3. Отладочная печать

Когда из тестов выжато всё что только можно, фантазия на тему «что бы ещё проверить» окончательно иссякла, а программа продолжает работать неправильно, наступает момент, когда необходимо понять, *почему* тесты дают столь неожиданные результаты; иначе говоря, нужно выяснить, *что в действительности происходит в программе*.

Пожалуй, самый «дешёвый и сердитый» способ для этого — вставить в программу дополнительные операторы, которые будут что-то печатать. Информация, которую такие операторы печатают, не имеет отношения к решаемой задаче, она нужна только в процессе отладки; собственно говоря, всё это так и называется — ***отладочная печать***. Чаще всего отладочная печать позволяет выяснить ответы на два вопроса: «доходит ли программа до этого места» и «как изменяется (какое значение получает) эта переменная».

Первое, что нужно запомнить относительно отладочной печати: **отладочные сообщения должны быть легко узнаваемы и не должны сливаться с остальной информацией, которую выдаёт ваша программа**. Можно начать каждое отладочное сообщение, например, с пяти звёздочек, или со слова «`DEBUG`» (обязательно заглавными буквами, если только ваша программа штатно не выдаёт сообщений заглавными — в таком случае маркер отладочной печати стоит набрать, напротив, строчными), или с лаконичного «`XXX`»; главное — чтобы отладочную печать было хорошо видно. Второе простое правило для отладочной печати — **не забывайте про переводы строк**; для Паскаля это означает, что нужно применять `writeln`. Дело тут в так называемой *буферизации вывода*; сообщение, которое не закончилось переводом строки, вы можете увидеть не сразу, а если программа завер-

шится аварийно — то и вообще не увидеть, что совершенно не годится для отладочной печати.

Если говорить точнее, операции вывода помещают информацию в *буфер*, из которого она отдаётся операционной системе для выдачи в поток в определённых случаях: при заполнении буфера, при завершении программы, при вызове процедуры, принудительно очищающей буфер (для Free Pascal такая процедура называется `flush`; в частности, `flush(output)` принудительно вытесняет буфер стандартного потока вывода). Кроме того, *при выводе на терминал* (в отличие от вывода в файл) вытеснение производится также при выдаче перевода строки и при затребовании программой операции ввода — на случай, если перед этим было выдано приглашение к вводу.

Ещё один момент, связанный с отладочной печатью, довольно очевиден, но некоторым ученикам это почему-то приходится повторять по несколько раз: **для операторов, связанных с отладочной печатью, структурные отступы никто не отменял**. Даже если вы намерены через пять минут вычистить из текста программы вставляемые операторы, это не повод превращать текст программы в нелепые каракули, пусть даже всего лишь на пять минут.

Впрочем, не торопитесь убирать из текста отладочную печать. Как показывает практика и гласит вселенский закон подлости, как только из программы будет убран последний оператор отладочной печати, в ней тут же обнаружится очередная ошибка, для отлова которой бóльшую часть только что убранных операторов придётся вставить обратно. Будет лучше взять отладочные операторы в фигурные скобки, превратив их в комментарии (но не нарушая при этом структурных отступов!) Однако даже вставка и удаление знаков комментария может оказаться неоправданно трудоёмким делом. Правильнее будет воспользоваться ***директивами условной компиляции***, которые пришли в Паскаль из языка Си и на первый взгляд выглядят довольно странно. Для условной компиляции используются «символы», которые нужно «определять»; ни для чего другого они не годятся, в отличие от Си, где основная роль аналогичных «символов» совершенно иная. Так или иначе, придумайте себе какой-то идентификатор, который вы будете использовать в качестве «символа» для включения и отключения отладочной печати и больше ни для чего; можно порекомендовать на эту роль слово «`DEBUG`». Поместите в начале программы директиву «`{$DEFINE DEBUG}`», которая «определит» этот символ. Теперь каждый фрагмент программы, предназначенный для отладочной печати, поместите между директивами условной компиляции, примерно так:

```
{$IFDEF DEBUG}
writeln('DEBUG: x = ', x, ' y = ', y);
{$ENDIF}
```

Пока в начале программы есть директива, определяющая символ `DEBUG`, такой оператор `writeln` будет учитываться компилятором, как обычно,

но если убрать директиву DEFINE, «символ» DEBUG станет неопределён, так что всё, что заключено между {$IFDEF DEBUG} и {$ENDIF}, компилятор просто проигнорирует. Заметим, директиву DEFINE даже не обязательно убирать полностью, достаточно убрать из неё символ «$», и она превратится в обычный комментарий, а вся отладочная печать отключится; если отладочная печать потребуется снова, достаточно будет вставить символ на место. Но можно сделать ещё лучше: директиву DEFINE в программу вообще не вставлять, а символ при необходимости определять из командной строки компилятора. Для этого достаточно добавить в командной строке флажок «-dDEBUG», примерно так:

```
fpc -dDEBUG myprog.pas
```

Действуя так, мы можем компилировать нашу программу с использованием отладочной печати или без, не изменяя при этом исходный текст. Почему это может оказаться важным, вы поймёте, когда начнёте использование системы контроля версий.

В некоторых случаях при отладке требуется узнать, каково текущее наполнение сложной структуры данных — списка, дерева, хеш-таблицы или чего-то ещё более сложного. Бояться таких ситуаций не следует, ничего сложного они собой не представляют; просто стоит, по-видимому, описать какую-нибудь процедуру, специально предназначенную для распечатки текущего состояния нужной нам структуры данных. Такую процедуру можно целиком заключить в условно-компилируемый фрагмент, чтобы в версию исполняемого файла без отладочной печати процедура не входила и не увеличивала объём машинного кода.

2.16.4. Отладчик gdb

Отладочная печать — средство, бесспорно, мощное, но в ряде случаев мы можем разобраться в происходящем гораздо быстрее, если нам позволят выполнить нашу программу по шагам с просмотром текущих значений переменных. Для этого используется специальная программа, которая называется ***отладчиком***; в современных версиях ОС Unix, в том числе Linux, наиболее популярен отладчик gdb[49], его мы сейчас и попробуем задействовать.

Первое, что нужно уяснить, начиная работать с отладчиком — нам потребуется определённая помощь со стороны компилятора. Отладчик работает с исполняемым файлом, в котором содержится результат перевода нашей программы в машинный код; как следствие, там обычно нет никакой информации об именах наших переменных и подпрограмм, о строках исходного текста и т. п.; отладчик мог бы, конечно, показать

[49] Сокращение от Gnu Debugger.

нам машинный код, получившийся из нашей программы, в виде мнемонических (ассемблерных) обозначений, и предложить нам пройтись по нему пошаговым выполнением, но толку от этого было бы не слишком много: скорее всего, глядя на мнемоники машинных команд, мы просто не узнаем конструкции программы, из которых этот код получился. С этой проблемой позволяет справиться так называемая ***отладочная информация***, которую компилятор может по нашей просьбе вставить в исполняемый файл. Эта информация включает сведения обо всех именах, которые мы использовали в программе, а также об именах файлов, содержавших исходный текст нашей программы, и о номерах строк исходного текста, из которых получился тот или иной фрагмент машинного кода.

Отладочная информация занимает сравнительно много места в исполняемом файле, никак не отражаясь на исполнении программы — она нужна только при использовании отладчика. Поэтому компилятор не снабжает исполняемый файл отладочной информацией, если его об этом не попросить, указав в командной строке ключ `-g`:

```
fpc -g myprog.pas
```

В этом случае исполняемый файл станет гораздо больше по размеру, но зато мы сможем, выполняя нашу программу под управлением отладчика, видеть фрагменты нашего исходного текста и использовать имена переменных.

Отладчик `gdb` представляет собой программу, имеющую свой собственный встроенный интерпретатор командной строки; все действия с нашей программой мы выполняем, давая отладчику команды. Отладчик умеет работать в разных режимах — в частности, его можно подключить к уже запущенной программе (процессу), а также с его помощью можно разобраться, в каком месте программы и по каким причинам произошло аварийное завершение[50], но нам пока будет достаточно разобраться только с одним, наиболее популярным режимом, при котором отладчик сам запускает нашу программу и контролирует ход её выполнения, повинуясь нашим командам. Командная строка, встроенная в `gdb`, оснащена функциями редактирования, автодополнения (только дополняются, естественно, не имена файлов, а имена переменных и подпрограмм), хранения истории введённых команд и поиска по ней, так что работать с `gdb` оказывается довольно удобно — при условии, что мы *умеем* это делать.

Запустить отладчик для работы в этом режиме можно, указав параметром имя *исполняемого* файла:

```
gdb ./myprog
```

[50] Если операционная система при этом сгенерировала так называемый core-файл; впрочем, с программами, откомпилированными с помощью `fpc`, этого почти никогда не происходит.

Если нужно передать нашей программе те или иные аргументы командной строки, это делается с помощью ключа `--args`, например:

```
gdb --args ./myprog abra schwabra kadabra
```

(в этом примере программа `myprog` будет запущена с тремя аргументами командной строки — `abra`, `schwabra` и `kadabra`).

После запуска отладчик сообщит свою версию и некоторую другую информацию и выдаст приглашение своей командной строки, обычно выглядящее так:

```
(gdb)
```

Начать выполнение программы мы можем командой `start`, в этом случае отладчик запустит нашу программу, но остановит её на первом же операторе главной части, не позволив ей ничего сделать; дальнейшее выполнение будет происходить под нашим контролем. Можно поступить иначе: дать команду `run`, тогда программа запустится и будет работать как обычно (то есть так, как она работает без отладчика), и если всё будет в порядке, то программа благополучно завершится, а отладчик так ничего и не сделает; но если выполнение программы прервать нажатием `Ctrl-C`, то при выполнении под отладчиком программа не будет уничтожена; вместо этого отладчик остановит её и выдаст своё приглашение командной строки, спрашивая нас, что дальше делать. Кроме того, если в ходе выполнения программы под отладчиком произойдёт её аварийное завершение, отладчик покажет нам то место в исходном тексте, где это произошло, и позволит просмотреть текущие значения переменных, что в большинстве случаев позволяет понять, почему произошла авария.

Когда отлаживаемая программа остановлена, мы можем приказать отладчику выполнить в ней *один шаг*; вопрос в том, что будет считаться «шагом». Ответов на этот вопрос предусмотрено два. Команда `next` рассматривает в качестве «шага» одну строку текста исходной программы, причём если в этой строке встречаются вызовы процедур и функций, их выполнение рассматривается как часть «шага», то есть внутрь вызываемых подпрограмм команда `next` не заходит. Вторая команда для «одного шага» называется `step` и отличается тем, что *заходит* внутрь вызываемых процедур и функций, то есть если в текущей строке присутствовал вызов подпрограммы и мы дали команду `step`, то после этого текущей станет первая строка текста этой подпрограммы. Команды `step` и `next` можно повторять, просто нажимая Enter, что изрядно ускоряет пошаговое выполнение.

Остановив выполнение программы (в том числе после команд `step` или `next`), отладчик обычно показывает строку исходного текста, которой соответствует текущая точка выполнения, что в большинстве случаев позволяет сориентироваться и понять, где мы. Если выданной

строки недостаточно, можно воспользоваться командой `list`, которая выдаст на экран окрестности текущей строки — пять строчек перед ней, её саму и пять строчек после. Если и этого не хватает, можно указать команде `list` номер строки, с которого начать; например, «`list 120`» выдаст десять строк, начиная со 120-й. При желании можно увидеть следующие десять строк, нажав Enter вместо ввода следующей команды, и так до конца файла.

Если «шагов», выполняемых программой, оказывается слишком много для пошагового выполнения, мы можем запустить программу на обыкновенное исполнение, в ходе которого она не будет останавливаться после каждого шага; при начальном запуске, как уже было сказано, для этого используется команда `run`, если же программа была приостановлена, продолжить её выполнение можно командой `cont` (от слова *continue*). Обычно перед тем, как воспользоваться одной из этих команд, в программе расставляют так называемые ***точки останова***; дойдя до такой точки, программа остановится, а отладчик выдаст нам приглашение, испрашивая дальнейших указаний. Точки останова устанавливаются с помощью команды `break`, которой необходим параметр; это может быть либо номер строки исходного текста (а при отладке программы, состоящей из нескольких исходных текстов — имя файла и номер строки, разделённые двоеточием), либо *имя подпрограммы* (процедуры или функции). В первом случае программа будет остановлена, дойдя до указанной строки (но до того, как эта строка будет выполнена), во втором случае программа остановится, как только будет вызвана указанная подпрограмма.

Отметим, что, поскольку Паскаль не различает в идентификаторах заглавные и строчные буквы, на уровне объектного кода (и в том числе отладочной информации) компилятор приводит все имена к верхнему регистру. Это означает, что **при использовании отладчика имена процедур и функций придётся писать заглавными буквами**, иначе отладчик нас не поймёт. Формально говоря, то же самое касается и имён переменных, но, например, версии компилятора и отладчика, которые использовал автор этих строк при подготовке книги, прекрасно понимали имена локальных переменных, написанные в любом регистре.

При создании новой точки останова отладчик показывает её номер, которым можно воспользоваться для более гибкой работы с остановками. Например, команда «`disable 3`» временно выключит точку останова №3, а команда «`enable 3`» включит её обратно. Команда «`ignore 3 550`» укажет отладчику, что точку останова №3 следует «проследовать без остановки» (проигнорировать) 550 раз, а остановиться лишь после этого — то есть если когда-нибудь до неё дойдёт дело в 551-й раз. Наконец, команда `cond` (от слова *conditional*) позволяет задать *условие* останова в виде логического выражения. Например,

```
cond 5 I < 100
```

указывает, что останавливаться на точке № 5 следует лишь в том случае, если значение переменной `i` окажется меньше ста. Команда «`info breakpoints`» позволяет узнать, какие у вас имеются точки останова, каковы установленные для них условия, счётчики игнорирования и т. п.

Просмотреть значения переменных, когда программа остановлена, можно с помощью команды `inspect`. При необходимости команда «`set var`» позволяет изменить значение переменной, хотя это используется сравнительно редко; например, «`set var x=50`» принудительно занесёт в переменную `x` значение `50`.

В программе, активно использующей подпрограммы, очень полезна может оказаться команда `bt` (или, если полностью, `backtrace`). Эта команда показывает, какие подпрограммы были вызваны (но ещё не завершились), с какими параметрами они были вызваны и из каких мест программы. Например, в ходе отладки программы `hanoi2` (см. § 2.14.2) команда `bt` могла бы выдать:

```
(gdb) bt
   #0  MOVELARGER (RODS=...) at hanoi2.pas:52
   #1  0x080483d9 in SOLVE (N=20) at hanoi2.pas:91
   #2  0x08048521 in main () at hanoi2.pas:110
```

Это означает, что сейчас активна процедура `MOVELARGER` (в тексте программы она называется `MoveLarger`), текущая строка — 52-я в файле `hanoi2.pas`; процедура `MoveLarger` была вызвана из процедуры `SOLVE` (`Solve`), вызов расположен в строке 91. Наконец, `Solve` вызвана из главной части программы (обозначается словом `main`; дело тут в том, что `gdb` ориентирован в основном на язык Си, а в нём вместо главной программы используется функция с именем `main`), вызов находится в строке 110.

Первое число в каждой строке выдачи команды `bt` — это *номер фрейма*. Используя этот номер, мы можем переключаться между контекстами перечисленных подпрограмм; например, чтобы просмотреть значения переменных в точках, где были вызваны нижестоящие подпрограммы. Например, в нашем примере команда «`frame 1`» позволит нам заглянуть в то место процедуры `Solve`, где она вызывает `MoveLarger`. После команды `frame` можно воспользоваться командами `list` и `inspect`, они будут выдавать информацию, относящуюся к текущей позиции выбранного фрейма.

Ещё одна полезная команда — `call`; она позволяет в любой момент вызвать любую из ваших подпрограмм с заданными параметрами. К сожалению, здесь имеются определённые ограничения; `gdb` ничего не знает, например, о паскалевских строках, так что, если ваша подпрограмма требует строки в качестве одного из параметров, вы можете

вызвать её, указав в качестве параметра какую-нибудь подходящую переменную, но конкретное строковое значение вы задать не можете.

Выход из отладчика производится командой `quit`, или вы можете устроить ситуацию «конец файла», нажав `Ctrl-D`. Кроме того, полезно знать, что в отладчике есть команда `help`, хотя работать с ней не так просто.

Как было сказано в начале параграфа, `gdb` может использоваться в разных режимах. Так, если вы уже запустили вашу программу, она ведёт себя неправильно, но вам не хочется повторять действия, которые вызвали такое поведение, либо вы не уверены, что вообще сможете воссоздать имеющуюся ситуацию, можно подключить отладчик к существующему процессу. Для этого нужно, естественно, узнать номер процесса; как это делается, мы рассказывали в § 1.4.7. Далее `gdb` запускается с двумя параметрами: именем исполняемого файла и номером процесса, например:

```
gdb ./myprogram 2765
```

Имя исполняемого файла необходимо отладчику, поскольку только из него можно взять отладочную информацию, то есть сведения об именах переменных и номерах строк. После успешного подключения отладчик приостанавливает процесс и ждёт ваших указаний; вы можете с помощью команды `bt` узнать, где находитесь и как туда попали, воспользоваться командой `inspect` для просмотра текущих значений переменных, использовать команды `break`, `cont`, `step`, `next` и т. д. После выхода из отладчика процесс продолжит выполнение, если, конечно, в ходе отладки вы его не убили.

2.17. Модули и раздельная компиляция

Пока исходный текст программы состоит из нескольких десятков строк, его проще всего хранить в одном файле. С увеличением объёма программы, однако, работать с одним файлом становится всё труднее, и тому можно назвать несколько причин. Во-первых, длинный файл элементарно тяжело перелистывать. Во-вторых, как правило, программист в каждый момент работает только с небольшим фрагментом исходного кода, старательно выкидывая из головы остальные части программы, чтобы не отвлекаться, и в этом плане было бы лучше, чтобы фрагменты, не находящиеся сейчас в работе, располагались бы где-нибудь подальше, то есть так, чтобы не попадаться на глаза даже случайно. В-третьих, если программа разбита на отдельные файлы, в ней оказывается гораздо проще найти нужное место, подобно тому, как проще найти нужную бумагу в шкафу с офисными папками, нежели в большом ящике, набитом сваленными в беспорядке бумажками. Наконец, часто бывает так,

что один и тот же фрагмент кода используется в разных программах — а ведь его, скорее всего, приходится время от времени редактировать (например, исправлять ошибки), и тут уже совершенно очевидно, что гораздо проще исправить файл в одном месте и скопировать файл целиком во все остальные проекты, чем исправлять один и тот же фрагмент, который вставлен в разные файлы.

Почти все языки программирования поддерживают включение содержимого одного файла в другой файл во время трансляции; в большинстве реализаций Паскаля, включая и наш Free Pascal, это делается директивой `{$I` *имя_файла*`}`, например:

```
{$I myfile.pas}
```

Это будет работать так же, как если бы вместо этой строчки вы прямо в этом месте вставили всё содержимое `myfile.pas`.

Разбиение текста программы на отдельные файлы, соединяемые транслятором, снимает часть проблем, но, к сожалению, не все, поскольку такой набор файлов остаётся, как говорят программисты, *одной единицей трансляции* — иначе говоря, мы можем их откомпилировать только все вместе, за один приём. Между тем, хотя современные компиляторы работают довольно быстро, но объёмы наиболее серьёзных программ таковы, что их полная перекомпиляция может занять несколько часов, а иногда и суток. Если после внесения любого, даже самого незначительного изменения в программу нам, чтобы посмотреть, что получилось, придётся ждать сутки (да и пару часов — этого уже будет достаточно) — работать станет совершенно невозможно. Кроме того, программисты практически всегда используют так называемые ***библиотеки*** — комплекты готовых подпрограмм, которые изменяются очень редко и, соответственно, постоянно тратить время на их перекомпиляцию было бы глупо. Наконец, проблемы создают постоянно возникающие конфликты имён: чем больше объём кода, тем больше в нём требуется различных глобальных идентификаторов (как минимум имён подпрограмм), растёт вероятность случайных совпадений, а сделать с этим при трансляции в один приём почти ничего нельзя.

Все эти проблемы позволяет решить техника ***раздельной трансляции***. Суть её в том, что программа создаётся в виде множества обособленных частей, каждая из которых компилируется отдельно. Такие части называются ***единицами трансляции***, или ***модулями***. Большинство языков программирования, включая Паскаль, предполагает, что в роли модулей выступают отдельные файлы. Обычно в виде обособленной единицы трансляции оформляют набор логически связанных между собой подпрограмм; в модуль также помещают и всё необходимое для их работы — например, глобальные переменные, если такие есть, а также всевозможные константы и прочее. Каждый модуль компилируется отдельно; в результате трансляции каждого из них

получается некий промежуточный файл, содержащий так называемый ***объектный код***[51], и такие файлы с помощью ***редактора связей*** (компоновщика) объединяются в готовый исполняемый файл; редактор связей обычно работает настолько быстро, что необходимость каждый раз перестраивать исполняемый файл из промежуточных не создаёт существенных проблем.

Очень важным свойством модуля является наличие у него собственного ***пространства видимости имён***: при создании модуля мы можем решить, какие из вводимых имён будут видны из других модулей, а какие нет; говорят, что модуль ***экспортирует*** часть вводимых в нём имён. Часто бывает так, что модуль вводит несколько десятков, а иногда и сотен идентификаторов, но все они оказываются нужны только в нём самом, а из всей остальной программы требуются обращения лишь к одной-двум подпрограммам, и именно их имена модуль экспортирует. Это снимает проблему конфликтов имён: в разных модулях могут появляться метки с одинаковыми именами, и это никак нам не мешает, если только они не экспортируются. Технически это означает, что при трансляции исходного текста модуля в объектный код все идентификаторы, кроме экспортируемых, исчезают.

2.17.1. Модули в Паскале

В Паскале файл, содержащий главную часть программы, синтаксически отличается от файлов, реализующих остальные (если угодно, «подчинённые») единицы трансляции. Главный модуль, как мы видели, начинается с необязательного *заголовка программы* с ключевым словом `program`; «неглавные» модули (с которыми мы пока не сталкивались) начинаются с **обязательного** заголовка с ключевым словом `unit`:

```
unit myunit;
```

В отличие от идентификатора, фигурирующего в заголовке программы, который абсолютно ни на что не влияет, идентификатор (имя) модуля — вещь очень важная. Во-первых, по этому имени модуль идентифицируется в других единицах трансляции, в том числе в главной программе; чтобы получить в своё распоряжение возможности модуля, необходимо поместить в программу (а при необходимости — в другой модуль, но об этом позже) директиву `uses`, уже знакомую нам по главе о полноэкранных программах. Использовавшийся нами ранее модуль `crt` входит в комплект поставки компилятора, но его подключение ничем

[51] Объектный код представляет собой своего рода заготовку для машинного кода: фрагменты программы представлены в нём последовательностями кодов команд, но в этих кодах могут не быть расставлены некоторые адреса, поскольку на момент компиляции они не были известны; окончательное превращение кода в машинный — задача редактора связей.

принципиально не отличается от подключения модулей, которые мы написали сами:

```
program MyProgram1;
uses myunit;
```

Директив `uses` можно использовать несколько, а можно перечислить модули через запятую в одной такой директиве, например:

```
uses crt, mymodule, mydata;
```

Проще всего сделать так, чтобы имя модуля, указанное в его заголовке, совпадало с основной частью имени файла, или, точнее, чтобы исходный файл модуля имел имя, образованное из названия модуля добавлением суффикса «`.pp`»; например, файл модуля `mymodule` проще всего будет назвать `mymodule.pp`. Это соглашение можно обойти, но мы такую возможность обсуждать не будем.

Дальнейший текст модуля должен состоять из двух частей: ***интерфейса***, помеченного ключевым словом `interface`, и ***реализации***, помеченной ключевым словом `implementation`. В интерфейсной части мы описываем всё то, что будет *видно из других единиц трансляции, использующих данный модуль*, причём для подпрограмм в интерфейсную часть мы помещаем только их заголовки; кроме подпрограмм, в интерфейсной части можно описать также константы, типы и глобальные переменные (только не надо при этом забывать, что глобальные переменные лучше вообще не использовать).

В реализации мы должны, во-первых, написать все подпрограммы, заголовки которых вынесены в интерфейсную часть; во-вторых, мы здесь можем описать любые объекты, которые не хотим показывать «внешнему миру» (то есть другим модулям); это могут быть константы, переменные, типы и даже подпрограммы, заголовков которых в интерфейсной части не было.

Основная идея разделения модуля на интерфейс и реализацию состоит в том, что обо всех особенностях интерфейса мы вынуждены подробно рассказывать тем программистам, которые будут использовать наш модуль, иначе они просто не смогут его использовать. Создавая документацию по нашему модулю, мы должны описать в ней все имена, которые вводит интерфейсная секция. Более того, когда кто-то начнёт использовать наш модуль (заметим, это касается и случая, когда мы используем его сами), нам придётся делать всё возможное, чтобы правила использования имён, вводимых в интерфейсе, не изменялись, то есть добавить новые типы или подпрограммы мы можем, а вот если нам придёт в голову *изменить* что-то из того, что там уже было, придётся сначала хорошенько подумать: ведь при этом все программы, использующие наш модуль, «сломаются».

С реализацией всё гораздо проще. Рассказывать о ней пользователям нашего модуля не нужно, включать в документацию тоже[52]; менять её мы можем в любой момент, не боясь, что сломается что-то кроме самого нашего модуля.

Кроме прочего, необходимо учитывать возможные конфликты имён. Если все имена, используемые в модуле, будут видимы во всей программе, а сама программа окажется достаточно большой, проблема случайных конфликтов имён вида «ой, кажется, кто-то так уже назвал совсем другую процедуру в совсем другом месте» доставляет программистам изрядную головную боль, особенно если некоторые модули используются одновременно в разных программах. Очевидно, что сокрытие в модуле тех имён, которые не предназначены для прямого использования из других единиц трансляции, резко снижает вероятность таких случайных совпадений.

Собственные пространства имён модулей позволяют решить не только проблему конфликта имён, но и проблему простейшей «защиты от дурака», особенно актуальную в крупных программных разработках, в которых принимает участие несколько человек. Если автор модуля не предполагает, что та или иная процедура будет вызываться из других модулей, либо что переменная не должна изменяться никак иначе, чем процедурами того же модуля, то ему достаточно не объявлять соответствующие метки глобальными, и можно ни о чём не беспокоиться — обратиться к ним другие программисты не смогут чисто технически.

В целом сокрытие деталей реализации той или иной подсистемы в программе называется ***инкапсуляцией*** и позволяет программистам более смело исправлять код модулей, не боясь, что другие модули при этом перестанут работать: достаточно сохранять неизменными и работающими те имена, которые вынесены в интерфейс.

Как и файл главной программы, файл модуля заканчивается ключевым словом `end` и точкой. В принципе, перед этим можно предусмотреть ещё и так называемую ***секцию инициализации*** (написать слово `begin`, несколько операторов и только потом `end` с точкой; эти операторы будут выполняться перед стартом главной части программы), но она имеет смысл только при наличии глобальных переменных, так что, если вы будете всё делать правильно, вам секция инициализации ещё очень долго не потребуется — возможно, даже никогда, если только вы не решите сделать Free Pascal своим основным инструментом.

В качестве примера вернёмся к нашему двоичному дереву поиска из § 2.14.5 и попробуем вынести в отдельный модуль всё необходимое для работы с ним. Интерфейс у нас будет состоять из двух *типов* —

[52] На самом деле реализацию тоже часто документируют, но такая документация предназначена не для пользователей модуля, а для тех программистов, которые работают в одной команде с нами и кому может потребоваться наш модуль отлаживать или совершенствовать.

собственно узла дерева и указателя на него, то есть типов `TreeNode` и `TreeNodePtr`, а также двух функций: `AddToTree` и `IsInTree`. При этом мы можем заметить, что «обобщённая» функция `SearchTree`, а также возвращаемый ею тип `TreeNodePos` представляют собой особенности реализации, о которых пользователю модуля знать не обязательно: а вдруг мы захотим эту реализацию изменить. Поэтому тип `TreeNodePos` будет описан в реализационной части модуля, а из заголовков функций в интерфейсной части будут присутствовать только `AddToTree` и `IsInTree`, но не `SearchTree`. Выглядеть это будет так:

```
{ lngtree.pp }
unit lngtree;
interface

type
    TreeNodePtr = ^TreeNode;
    TreeNode = record
        data: longint;
        left, right: TreeNodePtr;
    end;

function AddToTree(var p: TreeNodePtr; val: longint): boolean;
function IsInTree(p: TreeNodePtr; val: longint): boolean;

implementation

type
    TreeNodePos = ^TreeNodePtr;

function SearchTree(var p: TreeNodePtr; val: longint): TreeNodePos;
begin
    if (p = nil) or (p^.data = val) then
        SearchTree := @p
    else
    if val < p^.data then
        SearchTree := SearchTree(p^.left, val)
    else
        SearchTree := SearchTree(p^.right, val)
end;

function AddToTree(var p: TreeNodePtr; val: longint): boolean;
var
    pos: TreeNodePos;
begin
    pos := SearchTree(p, val);
    if pos^ = nil then
    begin
        new(pos^);
        pos^^.data := val;
        pos^^.left := nil;
        pos^^.right := nil;
        AddToTree := true
    end
    else
        AddToTree := false
end;

function IsInTree(p: TreeNodePtr; val: longint): boolean;
var
    pos: TreeNodePos;
begin
    pos := SearchTree(p, val);
```

```
    IsInTree := pos^ <> nil
end;

end.
```

Для демонстрации работы этого модуля напишем небольшую программу, которая будет читать с клавиатуры запросы вида «`+ 25`» и «`? 36`», на запрос первого вида будет добавлять указанное число в дерево, на запрос второго вида — печатать `Yes` или `No` в зависимости от того, есть указанное число в дереве или пока нет. Выглядеть программа будет так:

```
{ unitdemo.pas }
program UnitDemo;
uses lngtree;
var
    root: TreeNodePtr = nil;
    c: char;
    n: longint;
begin
    while not eof do
    begin
        readln(c, n);
        case c of
            '?': begin
                if IsInTree(root, n) then
                    writeln('Yes!')
                else
                    writeln('No.')
            end;
            '+': begin
                if AddToTree(root, n) then
                    writeln('Successfully added')
                else
                    writeln('Couldn''t add!')
            end;
            else
                writeln('Unknown command "', c, '"')
        end
    end
end.
```

Для компиляции всей программы достаточно запустить компилятор один раз:

```
avst@host:~$ fpc unitdemo.pas
```

Модуль `lngtree.pp` будет откомпилирован автоматически, причём только в том случае, если это требуется. Результатом будут два файла: `lngtree.ppu` и `lngtree.o`. Если исходный текст модуля изменить, то при следующей пересборке всей программы компилятор снова перекомпилирует модуль, а если его не трогать, перекомпилироваться будет только главная программа. О необходимости перекомпиляции модуля (или об отсутствии таковой) компилятор узнаёт, сравнив даты последней модификации файлов `lngtree.pp` и `lngtree.ppu`; если первый новее

(либо если второго просто нет), компиляция выполняется, в противном случае компилятор считает её излишней и пропускает. Впрочем, никто не мешает откомпилировать модуль «вручную», дав для этого отдельную команду:

```
avst@host:~$ fpc lngtree.pp
```

2.17.2. Использование модулей друг из друга

Довольно часто модулям приходится использовать возможности других модулей. Самый простой из таких случаев возникает, когда необходимо из подпрограммы одного модуля вызвать подпрограмму другого. Точно так же может возникнуть потребность использовать в теле подпрограммы имя константы, типа или глобальной переменной, которые введены другим модулем. Все эти случаи не создают никаких сложностей; достаточно вставить директиву `uses` **в секцию реализации** (обычно сразу после слова `implementation`), и все возможности, предоставляемые интерфейсом указанного в директиве модуля, станут вам доступны.

Несколько хуже обстоят дела, если вам приходится сделать *интерфейс* вашего модуля зависимым от другого модуля. К счастью, такие случаи встречаются гораздо реже, но они всё же возможны. Например, вам может потребоваться в интерфейсной части вашего модуля описать новый тип на основе типа, введённого в другом модуле (например, в одном модуле был введён какой-нибудь тип-запись, а другой модуль вводит тип-массив из таких записей, и т. п.). С тем же успехом вам может потребоваться при создании нового типа массива сослаться на константу, введённую другим модулем; наконец, вам может понадобиться в ваших интерфейсных подпрограммах параметр типа, описанного в другом модуле, или возврат значения такого типа из функции, или, в конце концов, просто глобальная переменная, имеющая тип, пришедший из другого модуля. Все эти ситуации объединены общим признаком: в *интерфейсной* части вашего модуля вы *используете* имя, введённое другим модулем.

В принципе, особых проблем нет и в этом случае: достаточно поместить директиву `uses` в интерфейсной части (обычно сразу после слова `interface`) или вообще в самом начале модуля сразу после его заголовка; эффект будет совершенно одинаковым. Следует только учитывать, что такая зависимость, в отличие от зависимости на уровне реализации, порождает определённые ограничения: **интерфейсные части двух и более модулей не могут зависеть друг от друга перекрёстно или «по кругу»**.

Вообще-то перекрёстных зависимостей между модулями в любом случае желательно избегать, но иногда такое всё же требуется; позаботьтесь тогда хотя бы о том, чтобы ваши модули использовали возможности

друг друга только в своих реализациях, но не в интерфейсах. В принципе, зависимостей интерфейсов друг от друга программисты стараются избегать, даже если они не перекрёстные, но это не всегда получается.

2.17.3. Модуль как архитектурная единица

При распределении кода программы по модулям следует помнить несколько правил.

Прежде всего, **все возможности одного модуля должны быть логически связаны между собой**. Когда программа состоит из двух-трёх модулей, мы ещё можем помнить, как именно распределены по модулям части программы, даже если такое распределение не подчинено никакой логике. Ситуация резко меняется, когда число модулей достигает хотя бы десятка; между тем, программы, состоящие из *сотен* модулей — явление достаточно обычное; больше того, можно легко найти программы, в состав которых входят тысячи и даже десятки тысяч модулей. Ориентироваться в таком океане кода можно только в том случае, если реализация программы не просто раскидана по модулям, но в соответствии с некой логикой разделена на подсистемы, каждая из которых состоит из одного или нескольких модулей.

Чтобы проверить, правильно ли вы проводите разбивку на модули, задайте себе по поводу каждого модуля (а также по поводу каждой подсистемы, состоящей из нескольких модулей) простой вопрос: *«За что конкретно отвечает этот модуль (эта подсистема)?»* Ответ должен состоять из одной фразы. Если дать такой ответ не получается, то, скорее всего, ваш принцип разбивки на модули нуждается в коррекции. В частности, если модуль отвечает не за *одну* задачу, а за *две*, притом не связанные между собой, логично будет рассмотреть вопрос о разбивке этого модуля на два.

Отметим ещё один момент, связанный с глобальными идентификаторами. В Паскале отсутствуют обособленные пространства для имён глобальных объектов, так что во избежание возможных конфликтов имён все глобально видимые идентификаторы, относящиеся к одной подсистеме (модулю или какому-то логически объединённому набору модулей) часто снабжают общим префиксом, обозначающим эту подсистему. Например, если вы создаёте модуль для работы с комплексными числами, имеет смысл все экспортируемые идентификаторы такого модуля начать со слова `Complex`, что-то вроде `ComplexAddition`, `ComplexMultiplication`. `ComplexRealPart` и т. д. В небольших программах это не столь актуально, но в крупных проектах конфликты имён могут стать серьёзной проблемой.

2.17.4. Ослабление сцепленности модулей

При реализации одних модулей постоянно приходится использовать возможности, реализованные в других; говорят, что реализация одного

модуля *зависит* от существования другого, или что модули ***сцеплены*** между собой. Опыт показывает, что **чем слабее сцепленность модулей, то есть их зависимость друг от друга, тем эти модули полезнее, универсальнее и легче поддаются модификации**.

Сцепленность модулей может быть разной; в частности, если один модуль использует возможности другого, но второй никак не зависит от первого, говорят об ***односторонней зависимости***, тогда как если каждый из двух модулей написан в предположении о существовании второго, приходится говорить о ***взаимной зависимости***. Кроме того, если модуль только вызывает подпрограммы из другого модуля, говорят о ***сцепленности по вызовам***, если же модуль обращается к глобальным переменным другого модуля, говорят о ***сцепленности по переменным***. Отличают также ***сцепленность по данным***, когда одну и ту же структуру данных, находящуюся в памяти, используют два и более модуля; вообще говоря, такая сцепленность может возникнуть и без сцепленности по переменным — например, если одна из подпрограмм, входящих в модуль, возвращает указатель на структуру данных, принадлежащую модулю.

Опыт показывает, что односторонняя зависимость всегда лучше, нежели зависимость взаимная, а сцепленность по вызовам всегда предпочтительнее сцепленности по переменным. Особенной осторожности требует сцепленность по данным, которая часто становится источником неприятных ошибок. В программе, идеальной с точки зрения разбиения на модули, все зависимости между модулями — односторонние, глобальных переменных нет вообще, а для каждой структуры данных, размещённой в памяти, можно указать её владельца (модуль, который отвечает, например, за своевременное уничтожение этой структуры), причём пользуется каждой структурой данных только её владелец.

Практика вносит некоторые коррективы в «идеальные» требования. Часто возникает необходимость во взаимозависимых модулях. Конечно, остаётся возможность слить такие модули в один, и в некоторых случаях именно так и следует поступить, но не всегда. К примеру, при реализации многопользовательской игры мы могли бы выделить в один модуль общение с пользователем, а в другой — поддержку связи с другими экземплярами нашей программы, которые обслуживают других игроков; практически неизбежно такие модули будут вынуждены обращаться друг к другу, но поскольку каждый из них отвечает за свою (чётко сформулированную!) подзадачу, объединять их в один модуль не нужно — ясность программы от этого нисколько не выиграет. Можно сказать, что взаимной зависимости модулей следует по возможности избегать, но всерьёз бояться её возникновения не стоит.

Иначе обстоит дело со сцепленностью по переменным и по данным. Без глобальных переменных можно обойтись *всегда*; при этом глобальные переменные затрудняют понимание работы программы,

ведь функционирование той или иной подпрограммы начинает зависеть не только от поданных ей на вход параметров, но и от текущих значений глобальных переменных, а обнаружить такую зависимость можно только путём внимательного просмотра текста подпрограммы. Во время отладки мы можем обнаружить, что некая подпрограмма работает неправильно из-за «странного» значения глобальной переменной, причём может остаться совершенно непонятно, кто и когда занёс в неё это значение. Говорят, что *глобальные переменные накапливают состояние.* Такое накапливание состояния способно затруднить отладку, даже если глобальные переменные локализованы в своих модулях, но в этом случае мы хотя бы знаем, где искать причины странного поведения программы: один модуль — это ещё не вся программа. Если же глобальная переменная видна во всей программе (экспортируется из своего модуля), то, во-первых, любое её изменение потенциально может нарушить работу любой из подсистем программы, и, во-вторых, изменения в неё может внести кто угодно, то есть приходится быть готовыми искать причину любого сбоя по всей программе.

Сцепленность по переменным имеет и другой негативный эффект: такие модули сложнее модифицировать. Представьте себе, что какой-то из ваших модулей должен «помнить» координаты некоего объекта в пространстве. Допустим, при его создании вы решили хранить обычные ортогональные (декартовы) координаты. Уже в процессе эксплуатации программы может выясниться, что удобнее хранить не декартовы, а полярные координаты; если модули общаются между собой только путём вызова подпрограмм, такая модификация никаких проблем не составит, но если переменные, в которых хранятся координаты, доступны из других модулей и активно ими используются, то о модификации, скорее всего, придётся забыть — переписывание всей программы может оказаться чрезмерно сложным. Кроме того, часто возникает такая ситуация, когда значения нескольких переменных как-то друг с другом связаны, так что при изменении одной переменной должна измениться и другая (другие); в таких случаях говорят, что необходимо обеспечить ***целостность состояния***. Сцепленность по глобальным переменным лишает модуль возможности гарантировать такую целостность.

Можно назвать ещё одну причину, по которой глобальных переменных следует по возможности избегать. Всегда (!) существует вероятность того, что объект, который сейчас в вашей программе один, потребуется «размножить». Например, если вы реализуете игру и в вашей реализации имеется игровое поле, то весьма и весьма вероятно, что в будущем вам понадобятся *два* игровых поля. Если ваша программа работает с базой данных, то можно (и нужно) предположить, что рано или поздно потребуется открыть одновременно две или больше таких баз данных (например, для изменения формата представления данных). Ряд примеров можно продолжать бесконечно. Если теперь

предположить, что информация, критичная для работы с вашей базой данных (или игровым полем, или любым другим объектом), хранится в глобальной переменной и все подпрограммы завязаны на использование этой переменной, то совершить «метапереход» от одного экземпляра объекта к нескольким у вас не получится.

Хуже всего обстоят дела со сцепленностью по динамическим структурам данных. Один из модулей может посчитать структуру данных ненужной и удалить её, тогда как в других модулях сохранятся указатели на эту структуру данных, которые будут по-прежнему использоваться. Возникающие при этом ошибки практически невозможно локализовать. Поэтому следует строго придерживаться «правила одного владельца»: у каждой создаваемой динамической структуры данных должен быть «владелец» (подсистема, модуль, структура данных, а в объектно-ориентированных языках программирования, соответственно, объект), и притом только один. За время своего существования динамическая структура данных может в случае крайней необходимости поменять владельца (например, одна подсистема может создать структуру данных, а использовать её будет другая подсистема), но правило существования и единственности владельца должно соблюдаться неукоснительно. Использовать структуру данных имеет право либо сам владелец, либо кто-то, кого владелец вызвал; в этом последнем случае вызванный не вправе предполагать, что структура данных просуществует дольше, чем до возврата управления владельцу, и не должен запоминать какие-либо указатели на эту структуру данных или на её части.

Отметим, что понятие «владельца» динамической структуры данных не поддерживается средствами языка программирования и, следовательно, существует лишь в голове программиста. Если отношение «владения» не вполне очевидно из текста программы, обязательно напишите соответствующие комментарии.

Общий подход к сцепленности модулей можно сформулировать следующими краткими правилами:

- **избегайте возникновения взаимных (двунаправленных) зависимостей между модулями, если это не сложно, но не считайте их совершенно недопустимыми;**
- **избегайте использования глобальных переменных, покуда это возможно; применяйте их только в случае, если такое применение способно сэкономить по меньшей мере несколько дней работы (экономию нескольких часов работы поводом для введения глобальных переменных лучше не считать);**
- **избегайте сцепленности по данным, а если это невозможно, то неукоснительно соблюдайте правило одного владельца.**

Литература

[1] Баурн С. Операционная система UNIX. М.:Мир, 1986.

[2] Dennis M. Ritchie. The Evolution of the Unix Time-sharing System. In: Lecture Notes in Computer Science 79: Language Design and Programming Methodology, Springer-Verlag, 1980. *Online version:* `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`

[3] Эрик С. Реймонд. Искусство программирования для Unix. М.: изд-во Вильямс, 2005. *В оригинале на английском языке книга доступна в сети Интернет по адресу* `http://www.faqs.org/docs/artu/`

[4] Линус Торвальдс, Девид Даймон. Just For Fun (рассказ нечаянного революционера). М.: изд-во Эксмо-пресс, 2002.

[5] Перельман Я. Живая математика. М.: Наука. Гл. ред. физ.-мат. лит., 1967. 160 с.

[6] Босс В. Лекции по математике. Т. 6: От Диофанта до Тьюринга: учебное пособие. М.: КомКнига, 2006. 208 с.

[7] Фейнман Р. Ф. Вы, конечно, шутите, мистер Фейнман! Пер. с англ. Н. А. Зубченко, О. Л. Тиходеевой, М. Шифмана. М.: НИЦ Регулярная и хаотическая динамика, 2001. 336 с

[8] Вирт Н. Алгоритмы и структуры данных. Пер. с англ. 2-е изд. СПб.: Невский Диалект, 2001. 352 с. ISBN 5-7940-0065-1

[9] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 2000. 960 с. ISBN 5-900916-37-5.

[10] Кнут Д. Искусство программирования, т. 3. Сортировка и поиск, 2-е изд. Пер. с англ. М.: Издательский дом «Вильямс», 2000. 832 с. ISBN 5-8459-0082-4

Предметный указатель

escape-последовательности, 64

Linux, 50
    дистрибуции, 61

Unix, 48, 51

абсолютное имя файла, 334
адрес, 53
    возврата, 241
    порта ввода-вывода, 56
    ячейки памяти, 187
алгебра двоичной логики, 131
алгоритм, 143
    детерминированность, 152
    дискретность, 151
    завершаемость, 154, 156
    конечность, 151
    массовость, 153
    объективность, 151
    применимость к входному слову, 155
алгоритмическая неразрешимость, 155
алфавит, 138, 141
аналоговые ЭВМ, 152
арифметическое выражение, 206
арифмометр
    Лейбница, 36
    Паскаля, 35
    Шиккарда, 34
ассемблер, 192

базис рекурсии, 252
байт, 126
Беббиджа машина
    аналитическая, 37
    разностная, 36
безусловный переход, 55
бесконечность
    континуальная, 138
    счётная, 137
библиотека, 447
бит, 126, 162
блок-схема, 221

ветвление, 221
взаимная зависимость, 455
видеопамять, 330
вызов процедуры, 241

генерализация блок-схемы, 263
главная часть программы, 198

дека, 382
дерево
    высота, 385
    двоичное, поиска, 385
    разбалансированное, 386
детализация блок-схемы, 263
диалоговый режим, 63
дизъюнктивная нормальная форма, 136
дизъюнкция, 131
директива условной компиляции, 440
драйвер, 53, 57

заглушка, 263
заголовок программы, 197
заём из несуществующего разряда, 170
зарезервированное слово, 214

идентификатор, 197, 210
импликация, 132
имя файла, 334
инициализация, 212
инкапсуляция, 450
интерфейс

модуля, 449
пользовательский, 61
исходный текст, 190, 196
итерация, 228

Кантора диагональный метод, 138
кеш, 58
ключ поиска, 384
ключевое слово, 214, 223
код
дополнительный, 170
завершения процесса, 266
исходный, 196
команды, 41
машинный, 189
операции, 55
символа, 166, 271
коллизия (в хеш-таблице), 387
команда
командной строки, 63
перехода, 55
командная строка, 63
компилятор, 190
компоновщик, 201, 448
конвейер, 86
конец файла (ситуация), 80, 280
конечный пользователь, 62
константа
восьмеричная, 261
времени исполнения, 257
времени компиляции, 256
двоичная, 261
именованная, 257
описание, 258
строковая, 198
типизированная, 260
шестнадцатеричная, 261
континуум, 138
контроллер, 52, 56
конъюнкция, 131

линейный поиск, 384
литерал, 256
восьмеричный, 261
двоичный, 261
строковый, 309
шестнадцатеричный, 261
логическое выражение, 223
логическое значение, 223

мантисса, 172, 261
массив, 293
машинное слово, 188
машинный код, 55
машинный порядок, 172
метка, 267
микропроцессор, 46
мнемоника, 192
модуль, 447
мусор, 355

Ньютона бином, 118

набор тестов, 434
недействительный адрес, 353
фон Неймана машина, 42
нулевой указатель, 353

область памяти, 166
объектный код, 448
объектный модуль, 201
односторонняя зависимость, 455
оконный менеджер, 65
окружение, 99
октет, 166
оперативная память, 187
оперативное запоминающее устройство, 52
оператор
if, 222
безусловного перехода, 267
ввода, 214
ветвления, 222
выбора, 317
вывода, 203
с переводом строки, 198
присваивания, 212
пустой, 279
составной, 225
цикла, 228
операторные скобки, 225
операционная система, 54, 60
операция
бинарная, 131
блочного чтения и записи, 333
ввода, 214
взятия адреса, 351
деления с остатком, 210
исключающего или, 132
конкатенации строк, 310
логического и, 131

логического или, 131
отрицания, 131
побитовая, 255
сравнения, 223
унарная, 131
целочисленного деления, 209
описание переменной, 211
основание системы счисления, 122
отладка, 429
отладочная информация, 442
отладочная печать, 439
отладчик, 441
относительное имя файла, 334
ошибка округления, 174

пакетный режим, 63
память
оперативная, 53
ячейки, 53
параметр
значение, 247
переменная, 249
параметры командной строки, 315
Паскаля арифмометр, 35
Паскаля треугольник, 119
переменная, 210
глобальная, 250
динамическая, 349, 353
значение, 211
начальное, 212
имя, 210
локальная, 242
неинициализированная, 212
обращение, 211
файловая, 333
цикла, 234
переменная окружения, 99
перенос
в несуществующий разряд, 168
переполнение, 171
периферийное устройство, 187
побочный эффект, 247
подпрограмма, 240
вызов, 240
декларация, 389
подстановка имён файлов, 77
поколение ЭВМ
второе, 45
первое, 41
третье, 46
четвёртое, 47
поле записи, 301
порядок, 261
поток ввода-вывода, 85
дескриптор, 86
пошаговая детализация, 263
преобразование типов, 305
неявное, 306
явное, 307
принцип хранимой программы, 41
приоритет арифметической операции, 207
проблема останова, 155
программа, 143
пространство видимости имён, 448
процедура, 240
конструктивная, 143
процесс, 81, 99
код завершения, 266

раздельная трансляция, 447
единица, 447
регистр центрального процессора, 54, 188
редактор связей, 448
рекурсия, 149

секция инициализации модуля, 450
секция интерфейса модуля, 449
секция описания
констант, 257
переменных, 211
типов, 288
секция реализации модуля, 449
сигнал, 82, 273
система команд, 55, 193
система счисления
восьмеричная, 126
двоичная, 123, 126, 127, 130
десятичная, 122, 124
позиционная, 122
шестнадцатеричная, 126
скрипт, 68
состояние, 147
список
двусвязный, 379
звено, 356
односвязный, 356
строковый литерал, 198

структурные отступы, 205
суперкомпьютер, 47
суффикс имени файла, 196
сцепленность модулей, 455
счётчик команд, 55, 188
счётчик цикла, 229

таблица истинности, 133
тавтология, 133
текстовый редактор, 196
тело цикла, 228
терминал, 64
тест, 435
тестирование, 434
тип
    анонимный, 287
    перечислимый, 290
    пользовательский, 288
    порядковый, 235, 292
    файловый, 333
тип выражения, 208, 211
тип переменной, 211
типизированный файл, 334
точка останова, 444
транзистор, 45
    полевой, 45
трит, 164
Тьюринга машина, 147
Тьюринга тезис, 147

узел дерева, 385
указатель, 350
    нетипизированный, 352
    типизированный, 351
условие, 221, 223
условный переход, 55
утечка памяти, 356

файл, 60
    исполняемый, 201
    нетипизированный, 344
        размер блока, 344
    объектный, 201
    текстовый, 338
    типизированный, 341
факториал, 114
фильтр (программа), 281
функция
    вычислимая, 142, 143
    математическая, 47
    примитивно-рекурсивная, 157
    частично-рекурсивная, 144
функция (подпрограмма), 240, 244

хеш-таблица, 387

целостность состояния, 456
центральный процессор, 52, 187
цикл, 227
    `while`, 228
    арифметический, 233
    вложенный, 237
    с постусловием, 232
    с предусловием, 232
цикл обработки команд, 55, 189
цифры
    арабские, 122, 125
    двоичные, 126

Чёрча тезис, 145
число с плавающей точкой, 172
    денормализованное, 173
    научная нотация, 209, 261

шина, 52, 56, 187
    адресов, 52
    данных, 52
    управления, 52

экспорт имён из модуля, 448
элемент массива, 293

язык
    алгоритмически полный, 178
    ассемблера, 192
        директива, 193
    программирования, 176
        диалект, 194
        компилируемый, 196
        строго типизированный, 211
    разметки, 177
    Тьюринг-полный, 178
    формальный, 176
ячейка памяти, 187

ГЛАВНЫЕ СПОНСОРЫ ПРОЕКТА

список наиболее крупных пожертвований

I: 45763 (19972+25791), *спонсор пожелал остаться неизвестным*

II: 41500, *спонсор пожелал остаться неизвестным*

III: 25000, **unDEFER**

IV: 21600, *спонсор пожелал остаться неизвестным*

V: 15000 (3000+7000+5000), *спонсор пожелал остаться неизвестным*

VI: 10524 (5262+5262), **os80**

VII: 10000, **Аня «canja» Ф.**

VIII: 10000 (2000+8000), **Сергей Сетченков**

IX: 8080, **Дергачёв Борис Николаевич**

X: 8072, *спонсор пожелал остаться неизвестным*

XI: 8000, **Смирнов Денис**

СТОЛЯРОВ Андрей Викторович

ПРОГРАММИРОВАНИЕ: ВВЕДЕНИЕ В ПРОФЕССИЮ
1: АЗЫ ПРОГРАММИРОВАНИЯ
Учебно-методическое издание

Рисунок и дизайн обложки Елены Доменновой
Корректор Екатерина Ясиницкая

Напечатано с готового оригинал-макета

Подписано в печать 02.03.2016 г.
Формат 60x90 1/16. Усл.печ.л. 29. Тираж 300 экз. Изд.№ 046.

Издательство ООО “МАКС Пресс”
Лицензия ИД № 00510 от 01.12.99 г.

11992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891

Отпечатано в ППП «Типография «Наука»
121099, Москва, Шубинский пер., 6
Заказ № 333

Андрей Викторович Столяров (род. 1974) – кандидат физико-математических наук, кандидат философских наук, доцент; работает на кафедре алгоритмических языков факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова.

http://www.stolyarov.info