THE UNIX PROGRAMMING ENVIRONMENT

Brian W. Kernighan, Rob Pike

PRENTICE HALL

HIGH TECH

UNIX

ПРОГРАММНОЕ ОКРУЖЕНИЕ

Брайан Керниган, Роб Пайк

Санкт-Петербург — Москва
2003

Брайан Керниган, Роб Пайк

UNIX. Программное окружение

Перевод П. Шера

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Козлихин*
Редактор	*В. Овчинников*
Художник	*В. Гренда*
Корректура	*С. Беляева*
Верстка	*Н. Гриценко*

Керниган Б., Пайк Р.
UNIX. Программное окружение. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 416 с., ил.
ISBN 5-93286-029-4

Книга представляет собой введение в программное окружение UNIX и адресована тем, кто хочет научиться программировать с помощью всех тех инструментов, которые поставляются с операционной системой. Рассматривается вход в систему, работа с файлами (cat, mv, cp, rm) и каталогами (cd, mkdir, ...), основы окружения (переменные, маски), фильтры (grep, sed, awk), программирование оболочки (циклы, сигналы, аргументы, стандартный ввод-вывод), введение в системные вызовы (read, write, open, creat, ...), введение в программирование с использованием lex, yacc и make, работа с документацией с помощью troff, tbl и eqn. Приводимые примеры не придуманы специально для этой книги, – некоторые из них впоследствии стали частью комплекта программ, используемых каждый день. Программы написаны на Си. Предполагается, что читатель знает или хотя бы изучает этот язык.

Прочтение этой книги как новичками, так и опытными пользователями поможет понять, как сделать работу с системой эффективной и приносящей удовольствие.

ISBN 5-93286-029-4
ISBN 0-13-937681-X (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 20.03.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 26 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

«Количество инсталляций UNIX возросло до 10, а ожидается еще больший рост».

(Справочное руководство по системе UNIX, 2-е издание, июнь 1972 года.)

Операционная система UNIX[1] стартовала на неиспользовавшейся DEC PDP-7 в Bell Laboratories в 1969 году. Кен Томпсон (Ken Thompson) при помощи и поддержке Руда Канадея (Rudd Canaday), Дага Мак-Илроя (Doug McIlroy), Джо Оссанны (Joe Ossanna) и Денниса Ритчи (Dennis Ritchie) написал небольшую универсальную систему с разделением времени, достаточно удобную для того, чтобы привлечь пользователей, и в конечном счете создавшую достаточный кредит доверия для покупки более мощной машины – PDP-11/20. Одним из первых пользователей стал Ритчи, который помогал перенести систему на PDP-11 в 1970 году. Кроме того, Ритчи спроектировал и написал компилятор для языка программирования Си. В 1973 году Ритчи и Томпсон переписали ядро UNIX на Си, прервав таким образом традицию написания системного программного обеспечения на языке ассемблера. И после этой переделки система, по существу, стала тем, чем она и является сегодня.

Примерно в 1974 году система была разрешена для использования в университетах «в учебных целях», а через несколько лет стали доступны ее коммерческие версии. В это время системы UNIX процветали в Bell Laboratories – они проникли в лаборатории, проекты по разработке программного обеспечения, центры обработки текстов и системы поддержки операций в телефонных компаниях. С того времени UNIX распространилась по всему миру – десятки тысяч систем установлено на различное оборудование, от микрокомпьютеров до самых крупных мэйнфреймов.

Почему UNIX имела такой успех? Можно привести несколько причин. Во-первых, благодаря тому что она написана на языке Си, она перено-

---

[1] UNIX – это торговая марка Bell Laboratories. «UNIX» – это *не* акроним, это намек на MULTICS, операционную систему, над которой Томпсон и Ритчи работали до UNIX.

сима – UNIX-системы работают на всевозможных компьютерах, от микропроцессоров до мэйнфреймов; это важное коммерческое преимущество. Во-вторых, исходный код доступен и написан на языке высокого уровня, что делает простым адаптирование системы для специфических требований пользователей. Наконец, и это самое важное, UNIX – *хорошая* операционная система, особенно для программистов. Среда программирования UNIX поразительно богата и продуктивна.

Хотя UNIX и представляет ряд новаторских программ и технологий, действенность системы не определяется какой-то одной программой или концепцией. Эффективность UNIX определяется применением особого подхода к программированию, философией использования компьютера. Для того чтобы описать эту философию, одного предложения, конечно же, недостаточно, но вот ее основная идея – мощность системы в огромной степени определяется взаимодействием программ, а не их наличием. Многие UNIX-программы по отдельности выполняют довольно тривиальные задачи, но будучи объединенными с другими, образуют полный и полезный инструментарий.

Цель этой книги в том, чтобы представить философию программирования в UNIX. Основа этой философии – взаимодействие программ, поэтому, хотя основное место отводится обсуждению отдельных инструментов, на протяжении всей книги обсуждается и тема комбинирования программ, а также их использования для построения других программ. Чтобы правильно работать с системой UNIX и ее компонентами, надо не только понимать, как применять программы, но и знать, как они взаимодействуют с окружением.

По мере распространения UNIX становилось все меньше тех, кто квалифицированно использовал бы ее приложения. Нередко случалось так, что опытным пользователям, в том числе и авторам этой книги, удавалось найти только «топорное» решение задачи, или же они писали программы для выполнения заданий, которые без проблем обрабатывались уже существующими средствами. Конечно же, чтобы найти красивое решение, необходимы опыт и понимание системы. Хотелось бы надеяться, что прочтение этой книги поможет как новичкам, так и «бывалым» пользователям понять, как сделать работу с системой наиболее эффективной и приносящей удовольствие. Используйте систему UNIX правильно!

Мы адресуем книгу программистам-одиночкам в надежде, что, сделав их труд более продуктивным, мы тем самым сделаем более эффективной и групповую работу. Хотя книга предназначена в основном для специалистов, содержание первых четырех или даже пяти глав можно понять, не имея опыта программирования, так что они могут иметь ценность и для других пользователей.

Везде, где возможно, приводятся не специально придуманные, а реальные примеры. Случалось даже, что программы, написанные как примеры для этой книги, впоследствии становились частью комплек-

та программ, используемых каждый день. Все примеры были протестированы непосредственно из текста,[1] который представлен в машинно-считываемой форме.

Книга организована следующим образом. Глава 1 представляет собой введение в самые основы работы с системой. Она описывает процесс регистрации, почту, файловую систему, наиболее употребительные команды и содержит начальную информацию о командном процессоре. Опытные пользователи могут пропустить эту главу.

Глава 2 посвящена обсуждению файловой системы UNIX. Файловая система является центральным звеном в функционировании ОС и в ее использовании, поэтому вы должны хорошо понимать ее для правильной работы в UNIX. Обсуждаются файлы и каталоги, привилегии и права доступа к файлам, а также индексные дескрипторы. В конце главы дан обзор иерархии файловой системы и рассказано о файлах устройств.

Командный процессор, или *оболочка,* – это основной инструмент как для исполнения программ, так и для их написания. Глава 3 показывает, как использовать оболочку для создания новых команд, работы с аргументами команд, переменными оболочки, элементарной управляющей логикой и перенаправлением ввода-вывода.

Глава 4 рассказывает о фильтрах – программах, которые выполняют некоторые простые преобразования данных по мере «пропускания» последних через себя. Первый раздел знакомит читателей с командой поиска по шаблону `grep` и ее сородичами, в следующем разделе обсуждаются некоторые наиболее распространенные фильтры, такие как `sort`, а оставшаяся часть главы посвящена двум универсальным программам преобразования данных: `sed` и `awk`. Программа `sed` – это потоковый редактор, осуществляющий редактирование потока данных по мере его поступления. А `awk` – это язык программирования, обеспечивающий простой информационный поиск и генерирование отчетов. В ряде случаев применение этих программ (возможно, во взаимодействии с оболочкой) позволяет полностью избежать программирования в его традиционном виде.

В главе 5 рассматривается применение оболочки для создания программ, которые будут использоваться другими людьми. Обсуждается более сложная управляющая логика и переменные, обработка системных сигналов. Примеры в этой главе составлены с использованием как `sed` и `awk`, так и оболочки.

Со временем вы достигнете предела того, что может быть реализовано при помощи оболочки и других, уже существующих программ. Глава 6

[1] В процессе редактирования русского издания книги все предлагаемые авторами примеры также проверялись на работоспособность и возможность их компиляции, если она необходима, в современном окружении систем Solaris, Linux и FreeBSD. – *Примеч. науч. ред.*

посвящена написанию новых программ с использованием стандартной библиотеки ввода-вывода. Программы написаны на Си. Предполагается, что читатель знает или хотя бы изучает этот язык одновременно с прочтением книги. Приведены разумные методики разработки и организации новых программ, описано поэтапное проектирование, показано, как использовать уже существующий инструментарий.

Глава 7 знакомит с системными вызовами – фундаментом, на котором построены все остальные слои программного обеспечения. Обсуждаемые темы: ввод-вывод, создание файла, обработка ошибок, каталоги, индексы, процессы и сигналы.

Глава 8 рассказывает об инструментах разработки программ, таких как: `yacc` – генератор грамматического анализатора; `make`, контролирующий процесс компиляции больших программ, и `lex`, генерирующий лексические анализаторы. Описание основано на создании большой программы – Си-подобного программируемого калькулятора.

В главе 9 обсуждаются средства подготовки документов, проиллюстрированные описанием (на уровне пользователя) и страницей руководства для калькулятора из главы 8. Данная глава может быть прочитана независимо от остальных.

Приложение A – это краткое изложение материалов о стандартном редакторе `ed`. Вероятно, некоторые читатели предпочитают какой-либо другой редактор для каждодневной работы, но, тем не менее, `ed` – это общедоступный, действенный и производительный редактор. Его регулярные выражения являются основой других программ (например, `grep` и `sed`), и уже по этой причине он заслуживает изучения.

Приложение B представляет собой справочное руководство по языку калькулятора из главы 8.

В приложении C содержится листинг окончательной версии программы калькулятора, для удобства весь код собран в одном месте.

Приведем несколько фактов. Во-первых, система UNIX приобрела большую популярность и в настоящий момент широко используется несколько ее версий. Например, седьмая версия происходит от первоисточника создания системы UNIX – Computing Science Research Center, Bell Laboratories (Центра исследования вычислительных систем лабораторий Белла). System III и System V – это версии, официально поддерживаемые Bell Laboratories. Университет Беркли, Калифорния, распространяет системы, являющиеся производной седьмой версии, известные как UCB 4.xBSD. Кроме того, существует множество вариантов, также полученных на основе седьмой версии, в частности для микрокомпьютеров.

Чтобы справиться с этим разнообразием, авторы рассматривают только те аспекты системы, которые (насколько этого можно ожидать) одинаковы во всех вариантах. Значительная часть представленного материала не зависит от версии системы, а в тех случаях, когда детали

реализации отличаются, предпочтение отдается седьмой версии, так как именно она распространена наиболее широко. Примеры также выполнялись на System V в Bell Laboratories и на 4.1BSD Университета Беркли; лишь в небольшом количестве случаев потребовались незначительные изменения. Ваш компьютер будет работать с ними вне зависимости от версии операционной системы, а обнаруженные различия будут минимальными.

Во-вторых, несмотря на то что в книге представлено много материала, это не справочное руководство. При создании этой книги более важным представлялось описание подхода и способа применения, а не конкретные детали. Стандартным источником информации является справочное руководство по UNIX. В нем можно найти ответы на вопросы, которых вы не получили в данной книге, там же можно прочитать об отличиях конкретной системы от той, которая описана здесь.

В-третьих, по мнению авторов, лучший способ научиться чему бы то ни было состоит в том, чтобы сделать это. Эту книгу надо читать за компьютером, чтобы иметь возможность экспериментировать, проверять или опровергать написанное, исследовать пределы возможностей. Прочитайте немного, попробуйте сделать то, что написано, потом вернитесь и читайте дальше.

UNIX – это, конечно же, не совершенная, но удивительная и непостижимая вычислительная среда. Надеемся, что читатели придут к этому же выводу.

Хотелось бы выразить многим людям благодарность за конструктивные замечания и критику, а также за помощь в усовершенствовании кода. В особенности Джону Бентли (Jon Bentley), Джону Линдерману (John Linderman), Дагу Мак-Илрою (Doug McIlroy) и Питеру Вейнбергеру (Peter Weinberger), которые с огромным вниманием читали многочисленные черновики. Мы признательны Алу Ахо (Al Aho), Эду Бредфорду (Ed Bradford), Бобу Фландрене (Bob Flandrena), Дейву Хансону (Dave Hanson), Рону Хардину (Ron Hardin), Марион Харрис (Marion Harris), Джерарду Хользманну (Gerard Holzmann), Стиву Джонсону (Steve Johnson), Нико Ломуто (Nico Lomuto), Бобу Мартину (Bob Martin), Ларри Рослеру (Larry Rosler), Крису Ван Вику (Chris Van Wyk) и Джиму Вейтману (Jim Weytman) за их замечания по первому варианту этой книги. Мы также благодарим Мика Бьянчи (Mic Bianchi), Элизабет Биммлер (Elizabeth Bimmler), Джо Карфагно (Joe Carfagno), Дона Картера (Don Carter), Тома Де Марко (Tom De Marco), Тома Дафа (Tom Duff), Дэвида Гея (David Gay), Стива Махани (Steve Mahaney), Рона Пинтера (Ron Pinter), Денниса Ритчи (Dennis Ritchie), Эда Ситара (Ed Sitar), Кена Томпсона (Ken Thompson), Майка Тилсона (Mike Tilson), Поля Туки (Paul Tukey) и Ларри Вера (Larry Wehr) за их ценные предложения.

Брайан Керниган (Brian Kernighan)
Роб Пайк (Rob Pike)

1

UNIX для начинающих

Что такое «UNIX»? В самом узком смысле слова – это ядро операционной системы с разделением времени – программа, которая управляет ресурсами компьютера и распределяет их между пользователями. UNIX позволяет пользователям запускать их программы; он управляет периферийными устройствами (дисками, терминалами, принтерами и т. п.), соединенными с машиной; кроме того, UNIX предоставляет файловую систему, которая обеспечивает долгосрочное хранение информации: программ, данных и документов.

В более широком смысле под «UNIX» понимается не только ядро системы, но и основные программы, такие как компиляторы, редакторы, командные языки, программы для копирования и печати файлов и т. д.

В еще более широком смысле «UNIX» может включать в себя программы, созданные вами или другими пользователями для запуска на вашей системе, например средства подготовки документов, программы статистического анализа или графические пакеты.

Какой из этих смыслов слова «UNIX» правилен, зависит от того, о каком уровне системы идет речь. Какое именно значение «UNIX» подразумевается в том или ином разделе данной книги, будет следовать из контекста.

Система UNIX может показаться более сложной, чем есть на самом деле, – новичку трудно разобраться в том, как использовать доступные средства наилучшим образом. Но, к счастью, начало пути не такое уж трудное – достаточно изучить всего несколько программ, и дело пойдет. Цель этой главы – помочь как можно быстрее начать работу с системой. Это скорее обзор, чем учебник; в следующих главах информация будет представлена более подробно.

Здесь речь пойдет о следующих важных областях:

- об основах – входе и выходе из системы, простых командах, исправлении ошибок ввода с клавиатуры, почте, межтерминальной связи;
- о повседневной работе – файлах и файловой системе, печати файлов, каталогах, наиболее употребительных командах;
- о командном процессоре, или *оболочке* (*shell*) – шаблонах (масках) имен файлов, перенаправлении ввода-вывода, конвейерах, или программных каналах (pipes), установке символов удаления (erase) и аннулирования ввода (kill), задании пути поиска команд.

Для тех, кто уже работал с системой UNIX, большая часть информации, представленная в главе 1, окажется уже знакомой, и они могут сразу перейти к главе 2.

Уже во время прочтения этой главы вам потребуется экземпляр справочного руководства по UNIX (UNIX Programmer's Manual);[1] во многих случаях легче сослаться на информацию руководства, чем пересказывать его содержание. Эта книга предназначена не для того, чтобы заменить руководство, а для того, чтобы показать, как лучше всего использовать команды, описанные в нем. Более того, возможны отличия между тем, что написано в этой книге, и тем, что справедливо для вашей системы. В начале руководства есть пермутационный указатель команд, описанных в руководстве, незаменимый для того, чтобы найти программу, подходящую для решения задачи; учитесь им пользоваться.

И в заключение один совет: не бойтесь экспериментировать. Новичок, даже случайно, мало что может сделать такого, что нанесло бы вред ему самому или другим пользователям. Так что изучайте работу системы на практике. Это длинная глава, и лучше всего читать ее порциями по несколько страниц, сразу пробуя делать то, о чем вы читаете.

1.1. Давайте начнем

Начальные сведения о терминалах и вводе с клавиатуры

Чтобы не рассказывать с самого начала все о компьютерах, будем считать, что вы в общих чертах знаете, что такое терминал и как им пользоваться. Если все же какое-либо из утверждений, приведенных ниже, будет непонятным, проконсультируйтесь с местным специалистом.

Система UNIX является *полнодуплексной* – символы, набранные на клавиатуре, посылаются в систему, которая, в свою очередь, посылает

[1] Под «справочным руководством по UNIX» имеется в виду интерактивная справочная система UNIX (так называемые man pages). Для ее применения надо просто иметь доступ к UNIX-машине. – *Примеч. науч. ред.*

их на терминал для вывода на экран. Как правило, такой эхо-процесс копирует символы прямо на экран, так что пользователь может видеть то, что он ввел; но в некоторых случаях, например при вводе пароля, эхо отключается, и символы на экран не выводятся.

Большинство символов на клавиатуре – это обычные печатные символы, у которых нет какого-либо специального значения, но есть и клавиши, которые сообщают компьютеру, как интерпретировать ввод. Безусловно, самая важная из таких клавиш – это *Return*. Нажатие клавиши *Return* означает окончание строки ввода; система реагирует на это перемещением курсора на терминале в начало следующей строки экрана. Для того чтобы система приступила к интерпретации вводимых символов, необходимо нажать *Return*.

Return – это пример *управляющего символа,* невидимый символ, который управляет некоторыми аспектами ввода с терминала и вывода на него. На любом нормальном терминале есть специальная клавиша *Return*, в отличие от большинства остальных управляющих символов. Вместо этого они должны вводиться следующим образом: нажимается и удерживается клавиша *Control* (может также называться *Ctl*, *Cntl* или *Ctrl*), а затем нажимается другая клавиша, обычно с буквой. Например, чтобы ввести *Return*, можно нажать клавишу *Return*, а можно, удерживая клавишу *Control*, ввести букву «m», так что *ctl*-m представляет собой альтернативное имя для *Return*. Другие управляющие символы включают: *ctl*-d, который сообщает программе, что ввод закончен; *ctl*-g, который воспроизводится на терминале как звуковой сигнал; *ctl*-h, обычно называемый *Backspace* (возврат на одну позицию), с помощью которого исправляют ошибки; и *ctl*-i – символ табуляции, который перемещает курсор на следующую позицию табуляции почти так же, как на обычной пишущей машинке. Позиции табуляции в системах UNIX разделены восемью пробелами. Символы табуляции и возврата на одну позицию имеют собственные клавиши на многих терминалах.

Еще две клавиши со специальным значением: *Delete*, иногда называемая *Rubout*[1] или какой-нибудь аббревиатурой, и *Break*, иногда называемая *Interrupt*. В большинстве систем UNIX нажатие клавиши *Delete* немедленно останавливает программу, не ожидая ее завершения. В некоторых системах эту функцию выполняет *ctl*-c. А также на некоторых системах, в зависимости от того, как подключены терминалы, *Break* – это также синоним *Delete* или *ctl*-c.

Сессия UNIX

Давайте начнем с диалога между пользователем и его системой UNIX, который мы будем комментировать. Во всех примерах этой книги при-

[1] Rub out – стирать, вычищать. – *Примеч. ред.*

няты следующие обозначения: то, что печатает пользователь, записывается *наклонными буквами*, ответы компьютера – обычными символами, а комментарии – *курсивом*.

```
Установите соединение: наберите телефонный номер или включите
терминал. Система должна сказать
login: you                  Введите имя и нажмите Return
Password:                   Пароль не появится на экране, когда будет введен
You have mail.              У вас есть непрочитанные письма
$                           Теперь система готова к выполнению команд
$                           Нажмите Return несколько раз
$ date                      Который час и какое сегодня число?
Sun Sep 25 23:02:57 EDT 1983
$ who                       Кто сейчас пользуется системой?
jlb      tty0    Sep 25 13:59
you      tty2    Sep 25 23:01
mary     tty4    Sep 25 19:03
doug     tty5    Sep 25 19:22
egb      tty7    Sep 25 17:17
bob      tty8    Sep 25 20:48
$ mail                      Читать почту
From doug Sun Sep 25 20:53 EDT 1983
give me a call sometime Monday

?                           Return – перейти к следующему сообщению
From mary Sun Sep 25 19:07 EDT 1983     Следующее сообщение
Lunch at noon tomorrow?

? d                         Удалить это сообщение
$                           Больше нет сообщений
$ mail mary                 Отправить почту пользователю mary
lunch at 12 is fine
ctl-d                   Конец письма
$                       Повесьте трубку или выключите терминал. Это все
```

Иногда сеанс на этом заканчивается, хотя время от времени люди делают заодно какую-нибудь работу. Оставшаяся часть этого раздела будет посвящена обсуждению приведенного выше сеанса и некоторых других программ, которые могут оказаться полезными.

Вход в систему

У пользователя должны быть имя и пароль, которые можно получить у системного администратора. Система UNIX поддерживает работу с разными терминалами, но она строго ориентирована на устройства, имеющие *нижний регистр*. Регистр имеет большое значение! Так что если терминал работает только в верхнем регистре (как некоторые видео- и портативные терминалы), жизнь пользователя превратится в такую пытку, что ему придется поискать другой терминал.

Убедитесь в том, что все установки выполнены соответствующим образом: верхний и нижний регистр, полный дуплекс и другие параметры,

которые может посоветовать местный специалист, например быстродействие или *скорость соединения* (*baud rate*). Установите соединение, используя те заклинания, которые нужны вашему терминалу; это может означать набор телефонного номера или просто щелчок тумблером. В любом случае система должна написать

```
login:
```

Если она пишет что-то непонятное, вероятно, установлена неправильная скорость; проверьте это значение и другие установки. Если это не поможет, несколько раз, не торопясь, нажмите клавишу *Break* или *Interrupt*. Если сообщение о начале сеанса так и не появляется, остается только позвать на помощь.

Когда сообщение `login:` получено, введите свое имя пользователя в *нижнем регистре*, затем нажмите *Return*. Если нужен пароль, система попросит его ввести и отключит на это время вывод на экран.

Кульминация усилий по входу в систему – это *приглашение*, обычно одиночный символ, который указывает, что система готова принимать команды. Наиболее часто в строке приглашения выводится знак доллара `$` или процента `%`, но можно заменить его любым, наиболее понравившимся; дальше будет рассказано, как это сделать. Приглашение на самом деле печатается программой, называемой *командным процессором*, или *оболочкой* (*shell*)[1]; она является основным интерфейсом пользователя для системы.

Непосредственно перед приглашением может присутствовать уведомление о наличии почты или «сообщение дня». Также может быть задан вопрос о типе подключенного терминала; ответ поможет системе учитывать специфические свойства терминала.

Ввод команд

Как только получено приглашение, можно начинать вводить *команды*, которые являются просьбой к системе выполнить некое действие. Слово *программа* будет употребляться как синоним команды. Итак, когда получено приглашение (будем считать, что это `$`), введите `date` и нажмите клавишу *Return* (*Enter*). Система должна ответить, выдав дату и время, а затем вывести новое приглашение, таким образом, вся транзакция будет выглядеть на терминале следующим образом:

```
$ date                    Который час и какое сегодня число?
Sun Sep 25 23:02:57 EDT 1983
$
```

[1] В профессиональном разговоре вы вряд ли услышите слово «оболочка». По всей вероятности, ваш собеседник скажет просто «шелл». – *Примеч. науч. ред.*

Не забудьте нажать клавишу *Return* и не вводите символ `$`. Если кажется, что система не обращает на вас внимания, нажмите *Return*, – что-нибудь должно произойти. *Return* больше не будет упоминаться, но не забывайте нажимать эту клавишу в конце каждой строки.

Теперь опробуем команду `who`, которая сообщает о том, кто в настоящее время находится в системе:

```
$ who
rlm       tty0     Sep 26 11:17
pjw       tty4     Sep 26 11:30
gerard    tty7     Sep 26 10:27
mark      tty9     Sep 26 07:59
you       ttya     Sep 26 12:20
```

Первая колонка содержит имена пользователей. Во второй находятся системные имена для соединений (tty означает «teletype» (телетайп) – устаревший синоним «терминала»). Все остальное – это информация о том, когда пользователь вошел в систему. Можно также попробовать

```
$ who am i
you       ttya     Sep 26 12:20
```

Если при вводе названия команды была допущена ошибка или введена несуществующая команда, то в ответ система выдаст сообщение о том, что такое имя не найдено:

```
$ whom                Команда введена с ошибкой...
whom: not found       ...поэтому система не знает, как ее выполнить
$
```

Если же случайно введено имя некой другой существующей команды, она выполнится, и результат, вероятно, будет трудно постижимым.

Странное поведение терминала

Однажды терминал может начать вести себя странно, например каждая буква выводится дважды или же *Return* не перемещает курсор на начало следующей строки. Обычно можно исправить положение, выключив и заново включив терминал или же выйдя из системы и войдя снова. Можно также прочесть описание команды `stty` (set terminal options – задание установок терминала) в разделе 1 руководства (man1). Для правильной обработки знаков табуляции (если сам терминал их не обрабатывает) введите команду

```
$ stty -tabs
```

и система преобразует символ табуляции в соответствующее количество пробелов. Если терминал позволяет устанавливать шаг табуляции, то это можно сделать посредством команды `tabs`. (На самом деле может потребоваться ввести

```
$ tabs    тип-терминала
```

для того, чтобы все заработало (обратитесь к описанию команды `tabs` в руководстве.)

Ошибки ввода

Если при вводе сделана ошибка и она обнаружена до нажатия *Return*, то существуют два способа исправления: *стереть* символы один за другим или всю строку целиком, а потом набрать ее заново.[1]

Если вводится символ *стирания строки* (*line kill* character) (по умолчанию символ `@`), то вся строка будет проигнорирована, как будто она никогда и не была введена, и ввод будет продолжен с новой строки:

```
$ ddtae@                    Абсолютно неправильно, начинаем заново
date                        с новой строки
Mon Sep 26  12:23:39 EDT 1983
$
```

Символ `#` (знак диеза) стирает последний введенный символ; каждый новый `#` стирает еще один символ по направлению к началу строки (но не за ее пределами). Так что ошибочный ввод можно исправить следующим образом:

```
$ dd#atte##e                Исправляем прямо в процессе ввода
Mon Sep 26  12:24:02 EDT 1983
$
```

На какие именно символы возложены функции стирания символа (забоя) и удаления строки, в большой степени зависит от системы. Во многих системах (включая ту, с которой работают авторы) знак забоя заменен на символ возврата на одну позицию (backspace), что удобно для видеотерминалов. Можно без труда проверить, как обстоят дела в конкретной системе:

```
$ datee←                    Проверим ←
datee←: not found           Это не ←
$ datee#                    Попробуем #
Mon Sep 26  12:26:08 EDT 1983   Это #
$
```

(Символ возврата на одну позицию изображен как ← для большей наглядности). В качестве символа удаления строки также часто используется *ctl*-`u`.

В данном разделе символ забоя будет обозначаться как диез `#`, поскольку это отображаемый символ, но помните, что ваша система может отличаться. В разделе «Настройка окружения» будет показано, как

[1] Материал этого раздела вряд ли имеет отношение к тем UNIX-системам, с которыми будет иметь дело читатель. Приведенная информация касается «настоящих» UNIX-систем с «настоящими» терминалами. В PC-шных версиях UNIX все попроще и поудобнее. – *Примеч. науч. ред.*

можно раз и навсегда определить знаки забоя и удаления строки по усмотрению пользователя.

Что делать, если требуется ввести знак забоя или удаления строки как часть текста? Если предварить символы # или @ обратной косой чертой \, они потеряют свое особое значение. Поэтому, чтобы ввести # или @, наберите \# или \@. Система может переместить курсор на следующую строку после прочтения @, даже если перед ним стояла обратная косая черта. Не беспокойтесь, «коммерческое at» было записано.

Обратная косая черта, иногда называемая *escape-символом*, широко используется для указания того, что следующий за ней символ имеет специальное значение. Чтобы стереть обратную косую черту, надо ввести два символа исключения: \##. Понятно почему?

Вводимые символы, прежде чем они доберутся до места назначения, рассматриваются и интерпретируются целым рядом программ, а способ их интерпретации зависит не только от места назначения, но и от того, каким образом они туда попали.

Каждый введенный символ незамедлительно отображается на терминале, если только эхо не выключено, что бывает достаточно редко. Пока не нажата клавиша *Return*, символы временно хранятся ядром, поэтому опечатки можно поправить при помощи символов забоя и аннулирования строки. Если символ забоя или удаления строки предварен обратной косой чертой, то ядро отбрасывает черту и сохраняет последующий символ без интерпретации.

При нажатии клавиши *Return* сохраненные символы посылаются в программу, которая занимается чтением с терминала. Эта программа, в свою очередь, может интерпретировать символы специальным образом; например, оболочка (командный процессор) не воспринимает как имеющие особое значение символы, перед которыми стоит обратная косая черта. Об этом будет рассказано в главе 3. Пока же запомните, что ядро обрабатывает удаление строки, забой и обратную косую черту, только если черта стоит перед знаком удаления строки или забоя; оставшиеся же символы могут быть особым образом интерпретированы другими программами.

Упражнение 1.1. Объясните, что произойдет с

```
$ date\@
```

□

Упражнение 1.2. Большинство командных процессоров (в отличие от оболочки UNIX System 7) интерпретируют знак диеза # как начало комментария и игнорируют весь текст от # до конца строки. Учитывая это, поясните запись, представленную ниже, считая, что # – это знак исключения:

```
$ date
Mon Sep 26  12:39:56 EDT 1983
```

```
$ #date
Mon Sep 26  12:40:21 EDT 1983
$ \#date
$ \\#date
#date: not found
$
```

□

Опережающий ввод с клавиатуры

Ядро считывает ввод с клавиатуры по мере поступления, даже если оно одновременно занято чем-то еще, так что можно печатать сколь угодно быстро, в любой момент, даже если какая-то команда выполняет печать. Если ввод с клавиатуры выполняется в то время, пока система печатает, введенные символы появятся на экране вперемешку с выводимыми, но они сохранятся отдельно и будут интерпретированы корректно. Можно вводить команды одну за другой, не дожидаясь их завершения или даже старта.

Остановка программы

Большинство команд можно остановить, введя символ *Delete*.[1] Клавиша *Break*, которая есть на большинстве терминалов, тоже может остановить программу, но это зависит от конкретной системы. В некоторых случаях, например в текстовых редакторах, *Delete* останавливает любое действие, выполняемое программой, но оставляет вас внутри программы. Большинство программ будут остановлены при выключении терминала или разрыве телефонного соединения.

Если требуется лишь приостановить вывод, например, чтобы сохранить на экране выводимые данные, введите *ctl*-s. Вывод остановится практически сразу же; программа будет находиться в «подвешенном» состоянии до тех пор, пока ее не запустят вновь. Чтобы возобновить вывод, введите *ctl*-q.

Выход из системы

Правильный способ выхода из системы – это ввод *ctl*-d вместо команды; так оболочка получает сообщение о том, что ввод закончен. (В следующей главе будет подробно описано, как именно это происходит.) Обычно можно просто выключить терминал или повесить телефонную трубку, но осуществляется ли при этом на самом деле выход, зависит от системы.

---

[1] Для большинства распространенных UNIX-систем это код нажатия клавиш *ctl*-C. – *Примеч. науч. ред.*

Почта

В системе имеется почтовая служба, посредством которой пользователи могут общаться друг с другом, поэтому, войдя в систему, вы можете увидеть сообщение

```
You have mail.
```

перед первым приглашением на ввод команды. Чтобы прочитать почту, введите

```
$ mail
```

Сообщения будут показаны по одному, начиная с самого свежего. После каждого элемента `mail` ждет указания, как поступить с этим сообщением. Два основных ответа системе: `d` – удалить сообщение и *Return* – не удалять (то есть оно еще будет доступно, когда вы в следующий раз захотите почитать почту). Возможны варианты: `p` – распечатать сообщение, `s` *имя-файла* – сохранить сообщение в файле с указанным именем и `q` – выйти из `mail`. (Если вы не знаете, что такое файл, считайте, что это такое место, где можно сохранить информацию под выбранным именем и взять ее оттуда позже. Файлы – это тема раздела 1.2, да и большинства других разделов этой книги.)

Программа `mail` – это одна из тех программ, которая может существовать в различных вариантах. Она вполне может отличаться от приведенного здесь описания. Подробная информация представлена в руководстве.

Отправить почту другому пользователю очень просто. Пусть надо послать сообщение человеку с регистрационным именем `nico`. Самый простой способ сделать это выглядит так:

```
$ mail nico
Теперь введите текст письма, заполнив
столько строчек, сколько потребуется...
Завершив последнюю строку письма,
нажмите control-d
ctl-d
$
```

Символ *ctl*-`d` указывает на конец письма, сообщая команде `mail`, что больше данные вводиться не будут. Если на полпути вы передумаете отправлять письмо, нажмите *Delete* вместо *ctl*-`d`. Наполовину сформированное письмо будет не отправлено, а сохранено в файле с именем `dead.letter`.

Для тренировки отправьте письмо самому себе, затем введите `mail`, чтобы прочитать его. (На самом деле это не такое уж безумие, каким может показаться, – это простой и удобный способ напомнить себе о чем-нибудь важном.)

Есть и другие способы отправки сообщений – можно послать заранее подготовленное письмо, можно отправить одно сообщение сразу нескольким адресатам, можно также отправить сообщение пользователям, работающим на других компьютерах. Команда `mail` подробно описана в разделе 1 справочного руководства по UNIX. С этого момента будем использовать условное обозначение `mail(1)`, подразумевая страницу, описывающую команду `mail` в man1. Все команды, обсуждаемые в этой главе, также описаны в man1.

Кроме этого, в вашей системе может оказаться сервис напоминаний, который называется календарем (см. `calendar(1)`); в главе 4 будет описано, как его настроить, если это не было сделано ранее.

Общение с другими пользователями

Если в вашей UNIX-системе работают несколько пользователей, то однажды, как гром среди ясного неба, на вашем экране появится что-то вроде

```
Message from mary tty7...  Сообщение от mary
```

в сопровождении замечательного звукового сигнала. Мэри хочет написать вам что-то, но пока вы не выполните определенные действия, не сможете ответить ей тем же. Чтобы ответить, введите

```
$ write mary
```

Так устанавливается двусторонний канал связи. Теперь все, что Мэри введет на своем терминале, отобразится на вашем, и наоборот. Надо отметить, что происходит это очень медленно, как будто вы разговариваете с Луной.

Если вы работаете с какой-то программой, надо перейти в такое состояние, в котором будет возможен ввод команды. Обычно, какая бы программа ни была запущена, она должна остановиться или быть остановлена, но в некоторых программах, как, например, в каком-нибудь редакторе и в самой `write`, существует команда `!` для временного выхода в оболочку (табл. 2 в приложении 1).

Команда `write` не накладывает каких-либо ограничений, поэтому если вы не хотите, чтобы печатаемые вами символы перемешались с тем, что печатает Мэри, вам необходим некий протокол – соглашение об обмене сообщениями. Одно из соглашений заключается в том, чтобы соблюдать очередность, обозначая конец каждого фрагмента с помощью `(o)` – от английского «over» (окончено), а также сообщать о своем намерении закончить диалог при помощи `(oo)` – «over and out» (заканчиваю и выхожу).

Терминал Мэри:	*Ваш терминал:*
`$ write you`	
	`$ Message from mary tty7...`
	`write mary`

```
Message from you ttya...
did you forget lunch? (o)
                                    did you forget lunch? (o)
                                    five@
                                    ten minutes (o)
ten minutes (o)
ok (oo)
                                    ok (oo)
                                    ctl-d
EOF
ctl-d
                                    $ EOF
$
```

Еще один способ выйти из `write` – нажать *Delete*. Обратите внимание на то, что ошибки, сделанные при вводе, не отображаются на терминале Мэри.

Если предпринимается попытка пообщаться с кем-то, кто в данный момент не находится в системе или не хочет, чтобы его беспокоили, будет выдано соответствующее сообщение. Если адресат находится в системе, но не отвечает в течение достаточно большого промежутка времени, вероятно, он занят или отошел от терминала; тогда введите *ctl*-`d` или *Delete*. Если не хотите, чтобы вас беспокоили, выполните команду `mesg(1)`.

Новости

Многие UNIX-системы предоставляют сервис получения новостей, информируя пользователей о более или менее интересных событиях. Чтобы вызвать этот сервис, введите

```
$ news
```

Существует также обширная сеть UNIX-систем, которые поддерживают контакт друг с другом посредством телефонных звонков, расспросите специалиста о команде `netnews` и о USENET.[1]

Руководство по UNIX

Справочное руководство по UNIX (man) содержит большинство необходимой информации о системе. Раздел 1 знакомит читателя с командами, в том числе с представленными в данной главе. Раздел 2 описывает системные вызовы, обсуждаемые в главе 7, а раздел 6 – это информация об играх. Остальные разделы рассказывают о функциях, которые могут использовать программисты на Си, форматах файлов и

[1] Прошло уже два десятка лет с момента написания этой книги, и теперь можно так же осторожно расспросить специалиста об Интернете. – *Примеч. науч. ред.*

о сопровождении системы. (Нумерация разделов меняется от системы к системе.) Не забудьте и пермутационный указатель для начала; беглого просмотра достаточно для того, чтобы найти команды, которые могут быть полезными для выполнения конкретной задачи. Есть также введение, содержащее обзор работы системы.

Руководство часто представляет собой в системе оперативную справку, так что его можно читать прямо за терминалом. Если возникает какая-то проблема и рядом нет специалиста, способного помочь, можно вывести на терминал любую страницу руководства, набрав команду `man` *имя-команды*.

Так, чтобы прочитать справку о команде `who`, введите

```
$ man who
```

и

```
$ man man
```

чтобы прочитать о самой команде `man`.

Автоматизированное обучение

В систему может быть включена команда `learn`, которая предоставляет возможность получения информации о файловой системе и основных командах, о редакторе, подготовке документов и даже о программировании на Си. Попробуйте ввести

```
$ learn
```

Если команда `learn` присутствует в системе, она скажет, что делать дальше. Если с `learn` ничего не выходит, попробуйте команду `teach`.

Игры

Одно из лучших средств почувствовать себя свободно наедине с компьютером и терминалом (хотя это и не признается официально) – это компьютерные игры. В комплект поставки системы UNIX входит не много игр, но можно пополнить запасы на месте. Поспрашивайте товарищей или обратитесь к разделу 6 руководства (man 6).

1.2. Повседневная работа: файлы и основные команды

Информация в системе UNIX хранится в *файлах*, которые по сути своей очень похожи на обычные офисные папки для документов. У каждого файла есть имя, содержимое, место для хранения и некоторая административная информация о его владельце и размере. Файл

может содержать письмо, список имен или адресов, исходные тексты программы, данные для обработки программой или даже саму программу в исполняемой форме и другие нетекстовые материалы.

Файловая система UNIX организована таким образом, что вы можете работать со своими файлами, не вмешиваясь в работу других пользователей и не позволяя другим мешать себе. Несть числа командам, осуществляющим различные манипуляции с файлами; для начала рассмотрим только самые часто используемые. Глава 2 предлагает систематическое описание файловой системы и знакомит с многими другими командами, имеющими отношение к работе с файлами.

Создание файлов – редактор

Если надо напечатать письмо, документ или программу, каким образом извлечь информацию, хранящуюся в компьютере? Большинство задач такого рода выполняется с помощью *текстового редактора* – программы для хранения и обработки информации в компьютере. Практически в каждой UNIX-системе есть *экранный редактор* – редактор, который использует способность современных терминалов отражать редакционные изменения, как только они сделаны. Два наиболее популярных экранных редактора – это `vi` и `emacs`.

Мы не будем описывать здесь конкретные редакторы, частично из-за ограниченного объема книги, а частично потому, что ни один из них не является стандартным.[1] Однако есть более старый редактор `ed`, который наверняка доступен в каждой системе. Он не поддерживает специальные возможности, присущие некоторым терминалам, поэтому может работать на любом. Он также является основой других важнейших программ (в том числе нескольких экранных редакторов), так что стоит того, чтобы его изучили. Краткое описание этого редактора приводится в приложении 1.

Вне зависимости от того, какой редактор вы лично предпочитаете, его следует изучить достаточно хорошо для того, чтобы создавать файлы. В этой книге – для конкретизации и для гарантии работоспособности предложенных примеров на любой машине – обсуждается `ed`, но вы, безусловно, можете остановить свой выбор на другом редакторе – том, какой вам больше нравится.

Чтобы создать с помощью `ed` файл с именем `junk`, содержащий некоторый текст, выполните следующее:

```
$ ed                    Вызывает текстовый редактор
a                       команда ed для добавления текста
теперь вводите
любой текст ...
```

[1] Хотя сейчас, конечно, можно сказать, что редактор `vi` все-таки является стандартным. – *Примеч. науч. ред.*

```
.                    Введите отдельно «.» – завершение ввода текста
w junk               Записать текст в файл с именем junk
39                   ed выводит количество записанных символов
q                    Выйти из ed
$
```

Команда a (append – добавить) сообщает ed о начале ввода текста. Символ «.», обозначающий конец текста, должен быть введен в начале отдельной строки. Не забудьте его – до тех пор пока этот символ не будет введен, ни одна из последующих команд ed не будет распознана – все, что вводится, будет восприниматься как продолжение текста.

Команда редактора w (write – записать) сохраняет введенную информацию; w junk сохраняет ее в файле с именем junk. Любое слово может выступать в качестве имени файла; в данном примере было выбрано слово junk (мусор), чтобы показать, что файл не очень-то важный.

В ответ ed выдает количество символов, помещенных в файл. До ввода команды w ничего записано не будет, так что если отключиться и уйти, то информация не будет сохранена в файле. (Если отключиться во время редактирования, то данные, которые обрабатывались, сохраняются в файл ed.hup, с ним можно продолжать работать в следующей сессии.) Если во время редактирования происходит отказ системы (то есть из-за сбоя аппаратного или программного обеспечения система внезапно останавливается), то файл будет содержать только те данные, которые были записаны последней командой write. После же выполнения w информация записана навсегда; чтобы получить к ней доступ, введите

```
$ ed junk
```

Конечно же, введенный текст можно редактировать: исправлять орфографические ошибки, изменять стиль формулировок, перегруппировывать абзацы и т. д. Закончив, введите команду q (quit – выход), чтобы выйти из редактора.

Что там за файлы?

Создадим два файла, junk и temp, чтобы знать, чем мы располагаем:

```
$ ed
a
To be or not to be
.
w junk
19
q
$ ed
a
That is the question.
.
w temp
```

```
22
q
$
```

Счетчики количества символов в `ed` учитывают и символ конца каждой строки, называемый разделителем строк (newline), с помощью которого система представляет *Return*.

Команда `ls` выводит список имен (не содержимого) файлов:

```
$ ls
junk
temp
$
```

Это действительно два только что созданных файла. (В списке могли быть и другие файлы, которых вы не создавали.) Имена автоматически сортируются в алфавитном порядке.

Как и у многих других команд, у `ls` есть параметры, предназначенные для изменения поведения команды по умолчанию. Параметры вводятся в командной строке после имени команды, обычно они составлены из знака минус «-» и какой-то одной буквы, которая и определяет конкретное значение. Например, команда `ls -t` задает вывод файлов, упорядоченный по времени (time) их последнего изменения, начиная с самого свежего.

```
$ ls -t
temp
junk
$
```

Параметр `-l` обеспечивает расширенный (long) список, предоставляющий больше информации о каждом файле:

```
$ ls -l
total 2
-rw-r--r-- 1 you          19 Sep 26 16:25 junk
-rw-r--r-- 1 you          22 Sep 26 16:26 temp
$
```

`total 2` сообщает, сколько блоков дискового пространства занято файлами; блок – это обычно 512 или 1024 символа. Строка `-rw-r--r-` информирует о том, у кого есть права на чтение файла и запись; в данном случае владелец `you` может читать файл и писать в него, а остальные пользователи могут только читать. Следующий за строкой прав доступа символ `1` – это количество ссылок (links) на файл; пока не обращайте на него внимания, поговорим о нем в главе 2. Владелец файла (то есть пользователь, который его создал) – это `you`. `19` и `22` – это значения количества символов в соответствующих файлах; они совпадают

со значениями, выданными ed. Далее следуют дата и время последнего изменения файла.

Параметры могут быть сгруппированы: ls -lt выдает те же данные, что и ls -l, но отсортированные по времени, начиная с самого нового. Параметр -u предоставляет информацию о том, когда файлы использовались: ls -lut выдает расширенный (-l) список, отсортированный в порядке времени использования, начиная с последнего. Параметр -r изменяет порядок вывода на обратный, так что ls -rt выводит файлы в порядке, обратном времени их последнего изменения. Можно также после команды указать имена файлов, тогда ls выведет информацию только по указанным файлам:

```
$ ls -l junk
-rw-r--r-- 1 you          19 Sep 26 16:25 junk
$
```

Строки, следующие за именем команды в командной строке, как -l и junk в примере выше, называются *аргументами* команды. Обычно аргументы – это параметры или имена файлов, которые должны использоваться командой.

Запись параметра при помощи знака минус и одной (например, -t) или нескольких букв (например, -lt) – это общепринятое соглашение. Обычно если команда допускает использование таких необязательных параметров, то они должны предшествовать именам файлов, но возможен и другой порядок. Хотя надо отметить, что UNIX-программы достаточно капризны в том, что касается обработки нескольких параметров. Например, стандартная UNIX System 7 ls не воспримет

```
$ ls -l -t              Не работает в UNIX System 7
```

как синоним ls -lt, тогда как другие программы требуют разделения параметров.

По мере изучения UNIX станет понятно, что какая-то регулярность или система в отношении необязательных аргументов практически отсутствует. У каждой команды есть только ей присущие особенности, и каждая сама определяет, что будет обозначать какая буква (так что одна и та же функция в разных командах может быть обозначена разными буквами). Такая непредсказуемость приводит в замешательство, и именно ее часто называют главным недостатком системы. И хотя ситуация постепенно улучшается – в новых версиях уже больше единообразия, единственное, что можно порекомендовать – это стараться держать руководство поблизости, когда будете писать собственные программы.

Печать файлов: cat и pr

Итак, мы создали несколько файлов. Как просмотреть их содержимое? Есть много программ, способных выполнить эту задачу, может

быть, даже больше, чем нужно. Одна из возможностей состоит в том, чтобы запустить редактор:

```
$ ed junk
19                      ed сообщает о том, что в junk 19 символов
1,$p                    Напечатать строки с первой по последнюю
To be or not to be      В файле всего одна строка
q                       Все сделано
$
```

Сначала ed выводит количество символов в junk; команда 1,$p сообщает редактору, что надо напечатать все строки этого файла. Изучив ed, вы сможете печатать только необходимые фрагменты, а не целый файл.

Бывают случаи, когда редактор не подходит для печати. Например, существует предельный размер файла (несколько тысяч строк), который ed может обработать. Более того, он печатает только один файл за раз, а возможны ситуации, когда требуется печать нескольких файлов – одного за другим, без остановок. Итак, есть несколько других вариантов.

Первый вариант – это cat, самая простая из всех печатающих программ; она печатает содержимое всех файлов, имена которых указаны в ее аргументах:

```
$ cat junk
To be or not to be
$ cat temp
That is the question.
$ cat junk temp
To be or not to be
That is the question.
$
```

Названные в аргументах файлы или файл выводятся на терминал вместе (отсюда имя cat – catenate[1] – соединять), один за другим, без каких-либо разделителей.

С небольшими файлами проблем не возникает, а вот если файл большой и соединение с компьютером высокоскоростное, то надо иметь очень хорошую реакцию, чтобы успеть нажать *ctl*-s и остановить вывод cat, пока он еще не ускользнул с экрана. Нет «стандартной» команды, позволяющей выводить файл на видеотерминал порциями, помещающимися на одном экране, но во многих UNIX-системах такая возможность есть. В вашей системе такая команда может называться pg или more. Наша называется p; о ней будет рассказано в главе 6.

Как и cat, команда pr печатает содержимое всех файлов, перечисленных в списке, но в форме, пригодной для построчнопечатающих устройств: каждая страница содержит 66 строк (11 дюймов в высоту), к

[1] Catenate – это редко употребляемый синоним слова concatenate (связывать, соединять).

которым добавлены дата и время изменения файла, имя файла и номер страницы, а также несколько пустых строк на месте сгиба бумаги. Итак, чтобы аккуратно распечатать junk, потом перейти на начало новой страницы и так же распечатать temp, надо выполнить команду:

```
$ pr junk temp

Sep 26 16:25 1983  junk Page 1

To be or not to be
        (60 more blank lines)

Sep 26 16:26 1983  temp Page 1

That is the question.
        (60 more blank lines)
$
```

Программа pr также может осуществлять вывод в несколько колонок:

```
$ pr -3 имена-файлов
```

Каждый файл будет распечатан в 3 колонки. Вместо «3» можно написать любое разумное число, и pr постарается сделать все наилучшим образом. (На месте слов *имена-файлов* должен быть введен список файлов для печати.) Команда pr -m выведет каждый файл в отдельном столбце. См. описание pr(1).

Отметим, что pr *не является* программой форматирования текста в том, что касается перегруппировки строк или выравнивания краев. Настоящие средства форматирования – это nroff и troff, которые будут описаны в главе 9.

Существуют также команды, печатающие файлы на высокоскоростном принтере. Почитайте в man о командах lp и lpr или найдите статью «printer» в пермутационном указателе.[1] Выбор программы зависит от оборудования, подключенного к компьютеру. Команды pr и lpr часто используются вместе – после того как pr отформатирует информацию надлежащим образом, lpr запускает механизм вывода на построчно печатающий принтер. Вернемся к этому чуть позже.

Перемещение, копирование и удаление файлов – mv, cp, rm

Давайте посмотрим еще на какие-нибудь команды. Первое, что можно сделать, – это изменить имя файла. Переименование осуществляется посредством «перемещения» файла из одного имени в другое, например:

```
$ mv junk precious
```

[1] В большинстве современных UNIX для этой цели подойдут команды man -k printer или apropos printer. Попробуйте и увидите, что будет. – *Примеч. науч. ред.*

Файл, который назывался `junk`, теперь называется `precious`; содержимое при этом не изменилось. Если теперь запустить `ls`, список будет выглядеть по-другому: в нем больше нет `junk`, зато есть `precious`.

```
$ ls
precious
temp
$ cat junk
cat: can't open junk
$
```

Будьте осторожны: если переименовать файл в уже существующий, то файл назначения будет заменен.

Чтобы создать *копию* файла (то есть чтобы иметь две версии чего-то), используйте команду `cp`:

```
$ cp precious precious.save
```

Дубликат `precious` создается в `precious.save`.

И наконец, когда вам надоест создавать и перемещать файлы, команда `rm` удалит все указанные файлы:

```
$ rm temp junk
rm: junk nonexistent
$
```

Если один из файлов, который надо удалить, не существует, программа выдаст соответствующее сообщение, в остальных же случаях `rm`, как и большинство команд UNIX, работает «молча». Нет ни подсказок, ни пустой болтовни, сообщения об ошибках лаконичны и, бывает, бесполезны. Новичков такая краткость может привести в замешательство, а вот опытных пользователей раздражают «словоохотливые» программы.

Каким должно быть имя файла?

До сих пор мы использовали имена файлов, не обсуждая, какими они вообще могут быть, так что теперь пришло время привести несколько правил. Во-первых, длина имени файла ограничена 14 символами.[1] Во-вторых, хотя практически все символы разрешены в именах файлов, здравый смысл подсказывает, что разумнее придерживаться отображаемых символов и по возможности избегать символов, которые могут иметь другое значение. Например, в команде `ls`, рассмотренной ранее, `ls -t` означает вывод списка, упорядоченного по времени. Так что

[1] Современные версии операционных систем семейства UNIX позволяют присваивать файлам и каталогам имена, длина которых достигает 255 символов. – *Примеч. науч. ред.*

если дать файлу имя -t, тяжело будет добиться его вывода по имени. (Кстати, как это сделать?) Кроме знака минус в начале слова есть и другие символы со специальным значением. Чтобы избежать недоразумений, пока не разберетесь во всем досконально, лучше ограничиться буквами, цифрами, точками и знаками подчеркивания. (Точками и символами подчеркивания принято разделять имя файла на части, как в примере про precious.save.) И последнее – не забудьте о том, что регистр также имеет значение: junk, Junk и JUNK – это три разных имени.

Несколько полезных команд

Теперь, когда нам известны основы создания файлов, просмотра их имен и вывода их содержимого, изучим еще полдюжины команд, занимающихся обработкой файлов.

Чтобы конкретизировать обсуждение, будем использовать файл с именем poem, который содержит знаменитое стихотворение Огастеса Де Моргана (Augustus De Morgan). Создадим файл с помощью ed:

```
$ ed
a
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.
w poem
263
q
$
```

Первой рассмотрим команду wc, подсчитывающую количество строк, слов и символов в одном или нескольких файлах; она получила имя за свою деятельность – подсчет слов (word-counting):

```
$ wc poem
       8      46     263 poem
$
```

Это означает, что в poem 8 строк, 46 слов и 263 символа. Понятие «слово» определяется чрезвычайно просто: любая строка символов, не содержащая пробелов, знаков табуляции и разделителей строк.

Команда wc может обсчитать несколько файлов (и вывести общие результаты), может также исключить подсчет какой-либо категории. См. описание wc(1).

Следующая команда называется grep; она просматривает файл в поиске строк, соответствующих шаблону. (Имя команды происходит от названия команды g/*regular-expression*/p (g/*регулярное-выражение*/p) редактора ed, о которой рассказано в приложении 1.) Предположим, что требуется найти слово «fleas» в файле poem:

```
$ grep fleas poem
Great fleas have little fleas
And little fleas have lesser fleas,
And the great fleas themselves, in turn,
  have greater fleas to go on;
$
```

Если указать параметр -v, то grep будет искать строки, *не* соответствующие шаблону. (Параметр назван «v» по имени команды редактора; можно считать, что он меняет смысл соответствия на обратный.)

```
$ grep -v fleas poem
  upon their backs to bite 'em,
  and so ad infinitum.
While these again have greater still,
  and greater still, and so on.
$
```

Команда grep может быть использована для просмотра нескольких файлов; в этом случае перед началом каждой строки, удовлетворяющей шаблону, будет выведено имя файла, чтобы было понятно, где найдено соответствие. Существуют также параметры для подсчета, нумерации и т. д. Также grep может обрабатывать шаблоны гораздо более сложные, чем просто слово (как «fleas»), но оставим обсуждение этой возможности до главы 4.

Третья команда, sort, сортирует введенные данные по алфавиту, строка за строкой. Для стихотворения это выглядит достаточно странно, но все-таки попробуем ее применить, чтобы посмотреть, что получится:

```
$ sort poem
  and greater still, and so on.
  and so ad infinitum.
  have greater fleas to go on;
  upon their backs to bite 'em,
And little fleas have lesser fleas,
And the great fleas themselves, in turn,
Great fleas have little fleas
While these again have greater still,
$
```

Упорядочение выполнено построчно, при этом порядок сортировки по умолчанию выглядит следующим образом: сначала пробелы, потом буквы в верхнем регистре и затем в нижнем регистре, так что алфавитный порядок соблюдается не строго.

Команда `sort` имеет огромное количество параметров, определяющих порядок сортировки: обратный, по числовым значениям, в лексикографическом порядке, игнорируя начальные пробелы, по полям внутри строки и т. д. – обычно лучше просмотреть все эти параметры, чтобы быть уверенным в их значении. Перечислим наиболее часто используемые из них:

`sort -r`	*Порядок, обратный обычному*
`sort -n`	*Сортировать по значениям чисел*
`sort -nr`	*По значениям чисел в обратном порядке*
`sort -f`	*Игнорировать различие регистров*[1]
`sort +n`	*Сортировать начиная с n+1-го поля*

Более подробная информация о команде `sort` представлена в главе 4.

Еще одна команда для работы с файлом – это `tail`. Она выводит последние 10 строк файла. Это слишком для стихотворения из 8 строчек, но для больших файлов это хорошо. Кроме того, у `tail` есть параметр, задающий количество строк для просмотра, поэтому чтобы напечатать последнюю строку стихотворения:

```
$ tail -1 poem
  and greater still, and so on.
$
```

`tail` также может печатать файл, начиная с заданной строки:

```
$ tail +3 имя-файла
```

Вывод начнется со строки 3. (Обратите внимание на естественную инверсию соглашения о знаке минус в аргументе.)

И последняя пара команд, которая служит для сравнения файлов. Предположим, что вариант `poem` хранится в файле `new_poem`:

```
$ cat poem
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$ cat new_poem
Great fleas have little fleas
  upon their backs to bite them,
And little fleas have lesser fleas,
  and so on ad infinitum.
```

[1] При этом b сортируется как B: b раньше, чем D, но B всегда будет выведено раньше, чем b. – *Примеч. науч. ред.*

```
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$
```

Эти два файла мало чем отличаются друг от друга, и человеку тяжело обнаружить отличия. На помощь приходят команды сравнения файлов. Команда `cmp` находит первое несовпадение:

```
$ cmp poem new_poem
poem new_poem differ: char 58, line 2
$
```

Это означает, что первое отличие встречается во второй строке (что действительно так), но при этом нет никакой информации о том, в чем это отличие заключается и есть ли еще различия, кроме первого.

Вторая команда, выполняющая сравнение файлов, сообщает обо всех измененных, удаленных или добавленных строках и называется `diff`:

```
$ diff poem new_poem
2c2
<   upon their backs to bite 'em,
---
>   upon their backs to bite them,
4c4
<   and so ad infinitum.
---
>   and so on ad infinitum.
$
```

Это означает, что строка 2 первого файла (`poem`) должна быть заменена на строку 2 второго файла (`new_poem`), аналогично для строки 4.

Вообще говоря, `cmp` применяется, когда надо убедиться, что файлы действительно имеют одинаковое содержимое. Эта команда быстро выполняется и подходит для любых (не только текстовых) файлов. А команда `diff` применяется, когда есть подозрение в том, что у файлов есть какие-то отличия, и надо узнать, какие именно строки не совпадают. При этом `diff` обрабатывает только текстовые файлы.[1]

Команды обработки файлов: резюме

Краткий перечень рассмотренных команд обработки файлов представлен в табл. 1.1.

[1] Сравнив двоичные файлы, `diff` скажет: «Binary files *первый* and *второй* differ» или, если они не отличаются, не скажет ничего. – *Примеч. науч. ред.*

Таблица 1.1. Основные команды, относящиеся к файловой системе

Команда	Действие
`ls`	вывести список имен всех файлов в текущем каталоге
`ls` *имена-файлов*	вывести только указанные файлы
`ls -t`	вывести список, упорядоченный по времени, начиная с самого нового
`ls -l`	вывести список с более подробной информацией; также `ls -lt`
`ls -u`	вывести список, упорядоченный по времени последнего использования; также `ls -lu`, `ls -lut`
`ls -r`	вывести список в обратном порядке; также `ls -rt`, `ls -rlt` и т. д.
`ed` *имена-файлов*	редактировать указанные файлы
`cp` *file1 file2*	копировать *file1* в *file2*, перезаписывая старый *file2*, если он существовал
`mv` *file1 file2*	переместить *file1* в *file2*, перезаписывая старый *file2*, если он существовал
`rm` *имена-файлов*	безвозвратно удалить указанные файлы
`cat` *имена-файлов*	вывести содержимое указанных файлов
`pr` *имена-файлов*	вывести содержимое с заголовком, 66 строк на странице
`pr -n` *имена-файлов*	вывести в *n* колонок
`pr -m` *имена-файлов*	вывести указанные файлы рядом друг с другом (несколько колонок)
`wc` *имена-файлов*	сосчитать количество строк, слов и символов в каждом файле
`wc -l` *имена-файлов*	сосчитать количество строк в каждом файле
`grep` *шаблон имена-файлов*	вывести строки, соответствующие *шаблону*
`grep -v` *шаблон имена-файлов*	вывести строки, *не* соответствующие *шаблону*
`sort` *имена-файлов*	вывести строки файла в алфавитном порядке
`tail` *имена-файлов*	вывести 10 последних строк файла
`tail -n` *имена-файлов*	вывести *n* последних строк файла
`tail +n` *имена-файлов*	вывести, начиная с *n*-й строки
`cmp` *file1 file2*	вывести положение первого отличия
`diff` *file1 file2*	вывести все отличия между файлами

1.3. Снова о файлах: каталоги

Система отличает ваш файл `junk` от любого другого с таким же именем. Распознавание возможно благодаря тому, что файлы сгруппированы в *каталоги* (подобно тому, как книги в библиотеке расставлены по полкам), и файлы в разных каталогах могут иметь одинаковые имена, не создавая никаких конфликтов.

Обычно у каждого пользователя есть личный, или *домашний каталог*, который содержит информацию, принадлежащую только этому пользователю (иногда его еще называют каталогом регистрации[1] (login directory)). Войдя в систему, пользователь попадает в домашний каталог. Можно изменить каталог, в котором вы работаете, обычно он называется рабочим, или *текущим каталогом*, но домашний каталог всегда один и тот же. Если специально не указано иное, то при создании файла он помещается в текущий каталог. Поскольку изначально это домашний каталог, то файл не имеет никакого отношения к файлу с таким же именем, который может существовать в каталоге другого пользователя.

Каталог может включать в себя не только файлы, но и каталоги («Great directories have lesser directories...» – «У больших каталогов есть маленькие каталоги...», как в стихотворении Огастеса Де Моргана). Естественный способ представления структуры файловой системы – это дерево каталогов и файлов. Внутри этого дерева можно перемещаться в поиске файла от корня дерева по ветвям и, наоборот, можно начать с того места, где вы находитесь, и двигаться обратно к корню.

Начнем с реализации последней описанной возможности. Основным инструментом будет команда `pwd` (print working directory – показать рабочий каталог), которая выводит имя текущего каталога:

```
$ pwd
/usr/you
$
```

Это означает, что в настоящее время пользователь находится в каталоге `you`, который находится в каталоге `usr`, который, в свою очередь, находится в *корневом каталоге*, который условно обозначается как /. Символы / разделяют компоненты имени; длина каждого компонента такого имени ограничена 14 символами, как и имя файла.[2] Во многих системах `/usr` – это каталог, содержащий каталоги всех обычных пользователей системы. (Даже если домашний каталог и не `/usr/you`, команда `pwd` все равно выведет нечто подобное, так что приведенные ниже рассуждения подойдут и для этого случая.)

---

[1] Хотя на самом деле так его не называют никогда. – *Примеч. науч. ред.*

[2] Была ограничена. См. выше. – *Примеч. науч. ред.*

Если теперь ввести

```
$ ls /usr/you
```

должен быть выведен точно такой же список имен файлов, который выводился командой `ls` без параметров. Если параметры не указаны, то `ls` выводит содержимое текущего каталога, если же имя каталога указано, то команда выдает содержимое именно этого каталога.

Теперь попробуем

```
$ ls /usr
```

Будет выведен длинный список имен, среди которых будет и регистрационный каталог `you`.

Следующим шагом будет попытка вывести содержимое корневого каталога. Ответ должен быть примерно таким:

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

(Не путайте два смысла символа /; это и обозначение корневого каталога, и разделитель компонентов имени файла.) Почти все перечисленные элементы представляют собой каталоги, но вот `unix` – это файл, содержащий ядро UNIX в исполняемом виде. Подробно поговорим об этом в главе 2.

Введем команду

```
$ cat /usr/you/junk
```

(если `junk` еще присутствует в домашнем каталоге). Имя

```
/usr/you/junk
```

называется *путем* к файлу. «Путь к файлу» – это название с интуитивно понятным смыслом, означающее полное имя пути от корня по дереву каталогов к конкретному файлу. В системе UNIX существует универсальное правило – везде, где может использоваться обычное имя файла, может использоваться и путевое имя.

Структура файловой системы напоминает генеалогическое дерево. Попробуем пояснить (рис. 1.1).

Файл `junk` пользователя `you` не имеет никакого отношения к файлу с таким же именем, принадлежащему Полу (`paul`) или Мэри (`mary`).

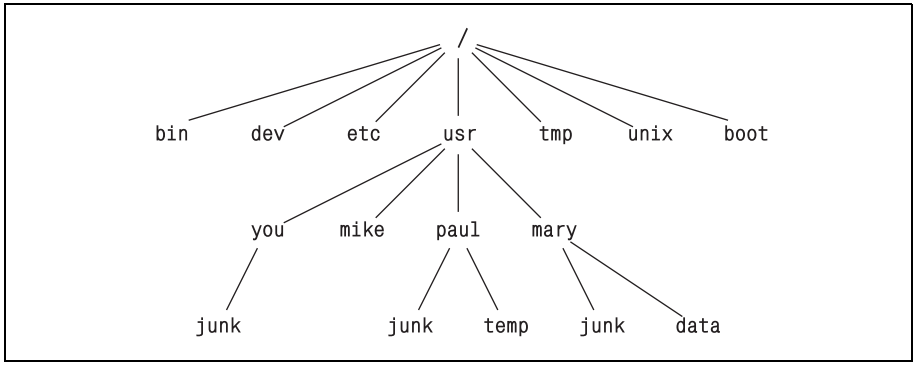

Рис. 1.1. *Дерево каталогов*

Если все интересующие пользователя файлы находятся в его личном каталоге, то путевые имена не имеют особого значения, а вот если он работает вместе с кем-то или занимается несколькими проектами одновременно, тут путевые имена становятся действительно полезными. Например, друзья могут распечатать ваш файл `junk`, просто введя команду

```
$ cat /usr/you/junk
```

Аналогично и вы можете, например, просмотреть, какие файлы есть у Мэри:

```
$ ls /usr/mary
data
junk
$
```

или сделать себе копию одного из ее файлов:

```
$ cp /usr/mary/data data
```

или даже отредактировать ее файл:

```
$ ed /usr/mary/data
```

Если пользователь не хочет, чтобы другие копались в его файлах, он может ограничить доступ к ним. Для каждого каталога и файла задаются права на чтение-запись-выполнение для владельца, группы и всех остальных, с помощью которых можно контролировать доступ. (Вспомните `ls -l`.) Пользователи наших локальных систем видят больше преимуществ в открытости, а не в секретности, но для других систем политика может быть иной. Вернемся к обсуждению этого вопроса в главе 2.

Итак, последняя серия экспериментов с путевыми именами:

```
$ ls /bin /usr/bin
```

Похожи ли полученные названия на что-то уже знакомое? Когда вы вызываете команду, печатая ее имя в командной строке (после соответствующего приглашения со стороны системы), система ищет файл с таким именем. Обычно она сначала просматривает текущий каталог (где, вероятно, файл не будет найден), затем каталог `/bin` и, наконец, `/usr/bin`. В командах, подобных `cat` или `ls`, нет ничего особенного, кроме того, что они собраны в нескольких каталогах для простоты поиска и администрирования. Чтобы проверить это, запустите какие-нибудь из подобных программ, указывая их полные путевые имена:

```
$ /bin/date
Mon Sep 26 23:29:32 EDT 1983
$ /bin/who
srm      tty1    Sep 26 22:20
cvw      tty4    Sep 26 22:40
you      tty5    Sep 26 23:04
$
```

Упражнение 1.3. Введите

```
$ ls /usr/games
```

и сделайте, что первым придет в голову. В нерабочее время некоторые вещи выглядят забавнее. □

Переход в другой каталог – cd

Если регулярно возникает необходимость работы с информацией из каталога другого пользователя, можно сказать, например: «Я хочу работать не со своими файлами, а с файлами Мэри». Такое пожелание реализуется с помощью изменения текущего каталога командой `cd`:

```
$ cd /usr/mary
```

Если теперь в качестве аргументов команд `cat` или `pr` использовать имена файлов без указания полного пути, то подразумеваться будут файлы из каталога, принадлежащего Мэри. Переход к другому каталогу никак не влияет на права доступа к файлу – если файл был недоступен из домашнего каталога, то переход в другой каталог этого факта не изменит.

Обычно бывает удобно организовать свои файлы таким образом, чтобы все, относящееся к одному проекту, находилось в одном каталоге, отдельно от других проектов. Например, кто-то, кто пишет книгу, может поместить все тексты в каталог под названием `book` (книга). Команда `mkdir` создаст новый каталог.

```
$ mkdir book        Создать каталог
$ cd book           Перейти в него
$ pwd               Убедиться, что переход осуществлен правильно
/usr/you/book
```

```
...                    Написать книгу (займет некоторое время)
$ cd ..                Подняться на один уровень файловой системы
$ pwd
/usr/you
$
```

Две точки `..` обозначают родительский каталог для текущего каталога, то есть каталог, находящийся уровнем выше (ближе к корню). Одна точка `.` обозначает текущий каталог.

```
$ cd                   Вернуться в домашний каталог
```

вернет вас в домашний (регистрационный) каталог.

Как только книга опубликована, все файлы можно стереть. Чтобы удалить каталог `book`, удалите все содержащиеся в нем файлы (позже будет рассказано о том, как сделать это быстро), потом перейдите (`cd`) в родительский каталог для `book` и введите:

```
$ rmdir book
```

Команда `rmdir` удаляет только пустые каталоги.

1.4. Оболочка

Когда система выводит приглашение на ввод команды `$` и пользователь вводит команды для выполнения, с ним общается не ядро, а специальный посредник, называемый командным интерпретатором, или оболочкой. Оболочка – это обычная программа, как `date` или `who`, хотя она и может выполнять некоторые замечательные действия. В том, что между пользователем и ядром находится оболочка, есть ряд несомненных преимуществ, некоторые из которых будут рассматриваться в этом разделе. Приведем три основных:

- Маски файлов: посредством шаблона можно задать в качестве аргументов программы целый набор имен файлов – оболочка найдет файлы, имена которых соответствуют шаблону.
- Перенаправление ввода-вывода: можно указать, что вывод любой программы должен производиться в файл, а не на терминал, и аналогично, что ввод должен осуществляться не с терминала, а из файла. Ввод и вывод даже могут быть соединены с другими программами.
- Персонализация окружения: можно определить собственные команды и их сокращенные имена.

Сокращения в именах файлов

Начнем с шаблона имени файла. Предположим, требуется ввести большой документ, например книгу. Логически книга разделена на небольшие фрагменты: главы и, может быть, разделы. Физически ее также надо разделить, ведь большие файлы неудобно редактировать.

Поэтому документ должен быть записан по частям в несколько файлов. Можно создать отдельный файл для каждой главы и назвать их ch1, ch2 и т. д. Если же каждая глава разбита на разделы, то можно создать файлы:

```
ch1.1
ch1.2
ch1.3
...
ch2.1
ch2.2
...
```

Именно такую структуру имеет эта книга. Если называть файлы систематически, руководствуясь соглашением об именах файлов, то можно без труда определить место каждого отдельного файла в целом.

Какие действия надо предпринять, чтобы напечатать книгу целиком? Можно сказать:

```
$ pr ch1.1 ch1.2 ch1.3 ...
```

но вводить имя за именем скоро надоест, к тому же, в длинном списке легко наделать ошибок. На помощь приходит маска имени файла. Вводим:

```
$ pr ch*
```

Для оболочки символ * означает любую строку символов, так что ch* – это шаблон, которому соответствуют имена всех файлов в текущем каталоге, начинающиеся с ch. Оболочка создает список таких файлов в алфавитном порядке[1] и передает этот список программе pr. Программа pr не видит никаких *, ей передается список строк, которые оболочка нашла в текущем каталоге по совпадению с шаблоном.

Решающим является то, что маска файла – это не свойство команды pr, а сервис оболочки. То есть ее можно использовать для образования последовательности имен файлов в любой команде. Например, чтобы сосчитать количество слов в первой главе:

```
$ wc ch1.*
    113     562    3200 ch1.0
    935    4081   22435 ch1.1
    974    4191   22756 ch1.2
    378    1561    8481 ch1.3
   1293    5298   28841 ch1.4
     33     194    1190 ch1.5
```

[1] Еще раз обратим ваше внимание на то, что порядок сортировки не строго алфавитный, буквы в верхнем регистре предшествуют буквам в нижнем. См. ascii(7) относительно упорядочения сортируемых символов.

```
      75    323   2030 ch1.6
    3801  16210  88933 total
$
```

Есть такая команда `echo`, которая чрезвычайно полезна для проверки значений символов маски. Легко догадаться, что делает эта команда – она «отражает» (выводит) свои аргументы:

```
$ echo hello world
hello world
$
```

Но ведь аргументы могут быть заданы шаблоном:

```
$ echo ch1.*
```

выводит имена всех файлов главы 1,

```
$ echo *
```

выводит имена *всех* файлов текущего каталога в алфавитном порядке,

```
$ pr *
```

распечатывает все файлы (в алфавитном порядке) и, наконец,

```
$ rm *
```

удаляет *все файлы* в текущем каталоге. (Надо было убедиться, что это *действительно* то, что вы хотели!)

Символ `*` не обязательно должен занимать последнюю позицию в имени файла, таких символов может быть несколько и они могут стоять в любом месте. Так,

```
$ rm *.save
```

удаляет все файлы, имена которых оканчиваются на `.save`.

Учтите, что имена файлов отсортированы в алфавитном, а не в числовом порядке. Поэтому если в книге 10 глав, то порядок вывода может оказаться не таким, как вы ожидали, – `ch10` предшествует `ch2`:

```
$ echo *
ch1.1 ch1.2 ... ch10.1 ch10.2 ... ch2.1 ch2.2 ...
$
```

Хотя шаблон, образованный с помощью `*`, и применяется чаще всего, имейте в виду, что оболочка может обрабатывать и другие шаблоны. Шаблону `[...]` удовлетворяет любой из символов, находящийся в скобках. Ряд последовательных букв или цифр может быть записан в краткой форме:

```
$ pr ch[12346789]*      Печатать главы 1,2,3,4,6,7,8,9, но не 5
$ pr ch[1-46-9]*        То же, что и в первом случае
$ rm temp[a-z]          Удалить любые существующие tempa, ..., tempz
```

Шаблону ? соответствует любой отдельный символ:

```
$ ls ?          Перечисляет файлы с именами из одного символа
$ ls -l ch?.1   Выводит ch1.1 ch2.1 ch3.1 и т. д., но не ch10.1
$ rm temp?      Удаляет файлы temp1, ..., tempa и т. д.
```

Обратите внимание, что шаблоны относятся только к *существующим* именам файлов. В частности, нельзя при помощи шаблонов создавать новые имена. Например, если надо в названии каждого файла заменить сокращение `ch` на слово `chapter`, то нельзя сделать это командой:

```
$ mv ch.* chapter.*      Не работает!!!
```

потому что не существует файлового имени, соответствующего шаблону `chapter.*`.

Шаблонные символы, такие как `*`, могут применяться как в простых именах файлов, так и в путевых; в этом случае соответствие определяется для каждого компонента пути, содержащего специальный символ. То есть `/usr/mary/*` совпадает со всем содержимым `/usr/mary`, а `/usr/*/calendar` генерирует список путевых имен файлов `calendar` всех пользователей.

Если символы `*`, `?` и т. д. интересуют вас сами по себе, в отрыве от своего специального значения, заключите весь аргумент в одинарные кавычки, как в нижеследующем примере

```
$ ls '?'
```

Также можно предварить специальный символ обратной косой чертой:

```
$ ls \?
```

(Поскольку `?` – это не символ аннулирования строки и не знак исключения, то такая обратная косая черта интерпретируется не ядром, а оболочкой.) Применение кавычек подробно обсуждается в главе 3.

Упражнение 1.4. В чем состоят отличия между этими командами?

```
$ ls junk       $ echo junk
$ ls /          $ echo /
$ ls            $ echo
$ ls *          $ echo *
$ ls '*'        $ echo '*'
```

□

Перенаправление ввода-вывода

Большинство рассмотренных команд осуществляют вывод на терминал; некоторые, например редактор, забирают данные тоже с терминала. Практически всегда в качестве терминала может выступать файл ввода или вывода или оба сразу. Например:

```
$ ls
```

выводит список файлов на терминал. Если же ввести

```
$ ls >filelist
```

то тот же самый список будет выведен не на терминал, а в файл filelist. Символ > означает «поместить вывод в указанный файл вместо терминала». Если такой файл не существовал, он будет создан, а если существовал, то перезаписан. На терминал ничего не выводится. Другой пример: можно объединить несколько файлов в один, собрав вывод команды cat в файл:

```
$ cat f1 f2 f3 >temp
```

Символ >> обозначает почти такое же действие, как и >, с той лишь разницей, что он «дописывает в конец». То есть команда

```
$ cat f1 f2 f3 >>temp
```

копирует содержимое f1, f2 и f3 в конец файла temp вместо того, чтобы перезаписывать имеющееся в нем содержимое. Как и в случае >, если файл temp ранее не существовал, он будет создан изначально пустым.

Аналогично символ < показывает, что входные данные программа должна брать не с терминала, а из файла, указанного далее. Например, можно подготовить письмо в файле let, а потом отправить его нескольким адресатам

```
$ mail mary joe tom bob <let
```

Во всех вышеприведенных примерах пробелы с любой стороны от > или < не обязательны, мы же придерживаемся традиционного форматирования.

Перенаправление вывода с помощью > предоставляет возможность комбинировать программы, позволяя получать результаты, ранее недоступные. Например, можно напечатать список пользователей в алфавитном порядке:

```
$ who >temp
$ sort <temp
```

Так как who выводит по одной строке для каждого зарегистрированного пользователя, а wc -l считает количество строк (отменяя подсчет слов и символов), то можно сосчитать количество пользователей в системе:

```
$ who >temp
$ wc -l <temp
```

Можно сосчитать количество файлов в текущем каталоге при помощи

```
$ ls >temp
$ wc -l <temp
```

хотя в этом случае само имя temp также включается в подсчет. Можно вывести файлы в три колонки:

```
$ ls >temp
$ pr -3 <temp
```

Наконец, можно узнать, зарегистрирован ли в системе конкретный пользователь, скомбинировав who и grep:

```
$ who >temp
$ grep mary <temp
```

Во всех этих примерах важно помнить, что интерпретацию символов < и > осуществляет оболочка (как и интерпретацию символов типа * в шаблонах имен файлов), а не конкретные программы. «Централизация» этой возможности в оболочке означает, что перенаправление ввода и вывода может применяться в любых командах; сама программа и не знает, что происходит что-то необычное.

Из-за этого возникло важное соглашение. Команда

```
$ sort <temp
```

сортирует содержимое файла temp, как и команда

```
$ sort temp
```

но существует отличие. Строка <temp интерпретируется оболочкой, поэтому sort не рассматривает имя temp как аргумент, вместо этого она сортирует свой *стандартный ввод*, который оболочка перенаправила таким образом, чтобы он поступал из файла. Во втором же примере с командой sort имя файла temp передается команде как аргумент, sort читает файл и сортирует его. Команда sort может получать в качестве аргументов несколько файлов, например

```
$ sort temp1 temp2 temp3
```

а если имена файлов не заданы, то она сортирует стандартный ввод. Это важное свойство большей части команд: если имена файлов не указаны, то обрабатывается стандартный ввод. Это значит, что можно просто «переговариваться» с программами, чтобы увидеть, как они работают. Например:

```
$ sort
ghi
abc
def
ctl-d
abc
def
ghi
$
```

Использование этого принципа будет обсуждаться в следующем разделе.

Упражнение 1.5. Объясните, почему

```
$ ls >ls.out
```

включает `ls.out` в список имен. □

Упражнение 1.6. Поясните вывод команды

```
$ wc temp >temp
```

Если в имени команды будет сделана ошибка, как в

```
$ woh >temp
```

что произойдет? □

Программные каналы

Во всех примерах предыдущего раздела применялся один и тот же прием: помещение вывода одной программы на вход другой через временный файл. Но ведь у временного файла нет другого назначения; на самом деле это слишком грубый метод. Данное наблюдение привело к созданию концепции программного *канала* (*pipe*), одного из наиболее важных достижений системы UNIX. Канал – это способ подключения вывода одной программы на вход другой без каких бы то ни было временных файлов; а *конвейер* (*pipeline*) – это соединение двух или более программ посредством каналов.

Давайте обратимся к примерам, рассмотренным ранее, и используем каналы вместо временных файлов. Символ вертикальной черты | сообщает оболочке, что надо образовать конвейер:

```
$ who | sort        Печатает отсортированный список пользователей
$ who | wc -l       Считает количество пользователей
$ ls | wc -l        Считает количество файлов
$ ls | pr -3        Выводит имена файлов в 3 колонки
$ who | grep mary   Ищет указанного пользователя
```

Любая программа, которая читает с терминала, может читать вместо этого из канала; любая программа, которая выводит данные на терминал, может выводить их в канал. Именно в этой ситуации соглашение о том, что если программе не указаны имена файлов, то она читает стандартный ввод, окупается в полной мере: все программы, соблюдающие это соглашение, могут быть включены в конвейеры. В примерах, рассмотренных выше, команды `grep`, `pr`, `sort` и `wc` используются в конвейерах именно таким образом.

В конвейер можно включить любое количество программ:

```
$ ls | pr -3 | lpr
```

создает список файлов в 3 колонки на построчно печатающем принтере, а

```
$ who | grep mary | wc -l
```

подсчитывает, сколько раз Мэри зарегистрирована в системе.

В действительности все программы конвейера выполняются одновременно, а не одна за другой. Это означает, что программа конвейера может быть и интерактивной; ядро следит за тем, как правильно обеспечить синхронизацию и диспетчеризацию, чтобы все работало.

Как вы, наверное, уже догадываетесь, созданием каналов занимается оболочка; отдельные программы не имеют понятия о перенаправлении ввода-вывода. Конечно, для того чтобы иметь возможность совместной работы, программы должны придерживаться определенных соглашений. Большинство команд имеет одинаковый дизайн, так что они подходят для любой позиции в конвейере. Обычно вызов программы выглядит следующим образом:

команда необязательные-аргументы необязательные-имена-файлов

Если имена файлов не указаны, то команда читает свой стандартный ввод, который по умолчанию является терминалом (удобно для экспериментов), но может перенаправляться и поступать из файла или канала. Что же касается вывода, большая часть команд направляет его на *стандартный вывод*, который по умолчанию связан с терминалом, но и он также может быть перенаправлен в файл или канал.

А вот сообщения об ошибках исполнения команд должны обрабатываться по-другому, иначе они могут исчезнуть в файле или канале. Поэтому у каждой команды есть еще и стандартный вывод ошибок, который обычно направляется на терминал. Схема на рис. 1.2 иллюстрирует весь процесс.

Практически все команды, описанные ранее, работают по этой модели; исключение составляют такие команды, как `date` и `who`, которые не читают входные данные, и несколько команд типа `cmp` и `diff`, у которых фиксированное количество входных файлов. (Но обратите внимание на параметр «-» в этих командах.)

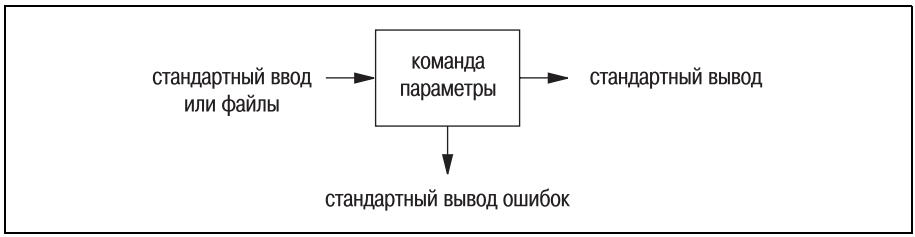

***Рис. 1.2.** Обработка сообщений об ошибках*

Упражнение 1.7. Объясните, в чем состоит отличие между

```
$ who | sort
```

и

```
$ who >sort
```

□

Процессы

Оболочка осуществляет немало других операций помимо организации конвейеров. Давайте ненадолго обратимся к основам одновременного выполнения нескольких программ, о котором упоминалось в предыдущем разделе. Например, можно запустить две программы в одной командной строке, разделив их точкой с запятой; оболочка распознает точку с запятой и разбивает строку на две команды:

```
$ date; who
Tue Sep 27 01:03:17 EDT 1983
ken      tty0    Sep 27 00:43
dmr      tty1    Sep 26 23:45
rob      tty2    Sep 26 23:59
bwk      tty3    Sep 27 00:06
jj       tty4    Sep 26 23:31
you      tty5    Sep 26 23:04
ber      tty7    Sep 26 23:34
$
```

Прежде чем оболочка выдала новое приглашение на ввод, выполнились обе команды (последовательно).

Несколько программ могут выполняться и одновременно. Предположим, надо сделать что-то, отнимающее много времени, например подсчитать количество слов в книге, при этом вам не хотелось бы ждать окончания выполнения команды `wc` для того, чтобы запустить какую-то другую программу. Чтобы добиться этого, введите

```
$ wc ch* >wc.out &
6944                    Идентификатор процесса, выведенный оболочкой[1]
$
```

Знак амперсанда `&` в конце командной строки сообщает оболочке, что следует «начать выполнение этой программы, а затем принимать новые команды с терминала незамедлительно», то есть не ждать завершения работы программы. Таким образом, команда начинает выполняться, и пока она выполняется, можно делать что-то еще. Если на-

[1] Скорее всего, будет напечатано что-то вроде `[1] 6944`. – *Примеч. науч. ред.*

править вывод в файл wc.out, то он не будет мешать вашей работе во время выполнения этой команды.

Экземпляр выполняющейся программы называется процессом. Число, выводимое оболочкой для команды, запущенной со знаком &, называется идентификатором процесса; с его помощью можно в других командах ссылаться на конкретную выполняющуюся программу.

Важно различать программы и процессы. Например, wc – это программа; каждый раз, когда она запускается, создается новый процесс. Если одновременно работают несколько экземпляров одной программы, то каждый из них представляет собой отдельный процесс со своим идентификатором (и все идентификаторы различны).

Если конвейер инициируется с применением &, например

```
$ pr ch* | lpr &
6951              Идентификатор процесса lpr
$
```

то все входящие в него процессы запускаются одновременно – знак & относится ко всему конвейеру. Но идентификатор выводится только один – для последнего процесса из последовательности.

Команда

```
$ wait
```

ждет, пока не выполнятся все процессы, инициированные с использованием &. Так что если она не возвращается тотчас же, значит ваши программы все еще работают. Команду wait можно прервать, нажав *Delete*.

Используя идентификатор, выведенный оболочкой, можно остановить процесс, инициированный со знаком &:

```
$ kill 6944
```

Если вы забыли значение идентификатора, используйте команду ps для получения сведений обо всех выполняющихся программах. Если ситуация выглядит безнадежной, kill 0 остановит все ваши процессы, кроме того экземпляра оболочки, в котором вы зарегистрировались. Если же любопытно узнать, чем занимаются другие пользователи, введите ps -ag[1] и получите информацию обо всех процессах, исполняющихся в этот момент. Приведем пример такого вывода:

```
$ ps -ag
   PID TTY TIME CMD
    36 co  6:29 /etc/cron
  6423 5   0:02 -sh
  6704 1   0:04 -sh
  6722 1   0:12 vi paper
```

[1] Скорее, ps -ax для BSD и ps -ef для System V. – *Примеч. науч. ред.*

```
  4430 2   0:03 -sh
  6612 7   0:03 -sh
  6628 7   1:13 rogue
  6843 2   0:02 write dmr
  6949 4   0:01 login bimmler
  6952 5   0:08 pr ch1.1 ch1.2 ch1.3 ch1.4
  6951 5   0:03 lpr
  6959 5   0:02 ps -ag
  6844 1   0:02 write rob
$
```

PID — это идентификатор процесса (process-id); TTY — терминал, связанный с процессом (как в who); TIME — время использования процессора в минутах и секундах; а в конце каждой строки выведена выполняющаяся команда. Команда ps в разных версиях системы выполняется по-разному, поэтому на некоторых компьютерах вывод может быть отформатирован иначе. Даже аргументы могут быть другими – см. страницу ps(1) в руководстве (man).

Процессы имеют такой же тип иерархической структуры, как и файлы: у каждого процесса есть порождающий (родительский) процесс, каждый процесс может порождать процессы. Ваш экземпляр оболочки был создан процессом, связанным с терминалом, с которого был осуществлен вход в систему. Когда вы запускаете команды, эти процессы становятся прямыми потомками оболочки. Если запустить программу из одного из них, используя, например, ! для выхода в оболочку из ed, порождается новый процесс, который, следовательно, является внуком оболочки.

Иногда процессу требуется так много времени на выполнение, что возникает желание запустить его, потом выключить терминал и уйти, не дожидаясь завершения. Однако если выключить терминал или прервать соединение, то процесс, как правило, будет убит, даже если указать знак &. Специально для таких ситуаций создали команду nohup (no hangup – не реагировать на разрыв линии). Если ввести

```
$ nohup      команда &
```

то выполнение команды будет продолжено даже после выхода из системы. Весь вывод команды сохраняется в файле nohup.out. Способа осуществить nohup команды задним числом не существует.

Если процесс занимает значительную часть ресурсов процессора, то по отношению к людям, использующим систему совместно с вами, будет вежливо запустить такой процесс с низким приоритетом; это делается при помощи программы nice:

```
$ nice      команда-требующая-огромных-ресурсов &
```

Программа nohup автоматически вызывает nice, ведь если пользователь все равно уходит, он может позволить программе выполняться чуть более медленно.

И последний вариант – можно сказать системе, чтобы она запустила процесс, например рано утром, когда нормальные люди еще спят, а не сидят за компьютером. Для этого существует команда `at(1)`:

```
$ at    время
любые
команды ...
ctl-d
$
```

Команды могут вводиться не только с терминала, но и из файла:

```
$ at 3am <file
$
```

Время может быть указано как в 24-часовом формате, например `2130`, так и в 12-часовом, например `930pm`.

Настройка окружения

Одна из приятных особенностей системы UNIX заключается в том, что существует несколько способов настраивать среду, приспосабливая ее к своим вкусам или к соглашениям локального окружения. Например, ранее упоминалась проблема разных стандартов для символа забоя и символа удаления строки; по умолчанию это обычно `#` и `@`. Каждый раз, когда это необходимо, можно заменить их

```
$ stty erase e kill k
```

где *e* – это символ, выбранный для обозначения забоя, а *k* – для удаления строки. Но осуществлять эту операцию при каждом входе в систему достаточно неудобно.

На помощь приходит оболочка. Если в каталоге регистрации есть файл `.profile`, то при входе пользователя в систему оболочка выполнит команды, содержащиеся в этом файле до того, как выдать первое приглашение на ввод. Так что поместите все команды по настройке среды в соответствии со своими предпочтениями в `.profile`, и они будут выполняться при каждом входе в систему.

Первая команда, которую большинство пользователей помещают в свой `.profile`:

```
stty erase <-
```

Обозначение `<-` использовано для наглядности, в `.profile` же можно поместить именно буквенное обозначение возврата на одну позицию. Команда `stty` воспринимает и такое представление, как, например, `^x` для *ctl*-`x`, так что аналогичного эффекта можно добиться при помощи:

```
stty erase '^h'
```

потому что *ctl*-h – это и есть возврат на одну позицию. (Символ ^ – это устаревший синоним для оператора |, образующего конвейер, поэтому следует заключать его в кавычки.)

Если терминал не поддерживает переход по табуляции, можно добавить -tabs в строку stty:

```
stty erase '^h' -tabs
```

Для того чтобы узнать о загруженности системы на момент входа, добавьте

```
who | wc -l
```

для подсчета пользователей. Если доступна служба новостей, можно добавить news. Некоторым нравится, когда в процессе регистрации на экран выводятся афоризмы:

```
/usr/games/fortune
```

Через некоторое время может показаться, что вход в систему занимает слишком много времени, тогда сократите .profile до необходимого минимума.

Некоторые свойства оболочки определяются так называемыми переменными окружения, значения которых доступны пользователю и могут быть им изменены. Например, строка приглашения к вводу, выводимая как $, на самом деле хранится в переменной оболочки PS1, для которой можно установить любое значение, например:

```
PS1='Yes dear? '
```

Кавычки необходимы, так как в строке приглашения есть пробелы. А вот пробелы вокруг знака = в этой конструкции не разрешены.

Также оболочка особым образом интерпретирует переменные HOME и MAIL. HOME – это имя домашнего каталога, обычно оно устанавливается правильно и не требует записи в .profile. Переменная MAIL определяет название стандартного файла, где хранится почта. Если определить его для оболочки, то уведомление о получении нового сообщения будет появляться после каждой команды:[1]

```
MAIL=/usr/spool/mail/you
```

[1] Это плохо реализовано в оболочке. Просмотр файла после каждой команды ощутимо увеличивает нагрузку на систему. К тому же, если долго работать в редакторе, уведомления о новых сообщениях не будут поступать, так как команды в оболочке при этом не запускаются. Лучше просматривать файл не после каждой команды, а каждые несколько минут. Реализация такой программы, проверяющей почту, обсуждается в главах 5 и 7. Третья возможность (правда, доступная не каждому) – это получение уведомлений от самой программы mail: она-то уж наверняка знает, когда приходит новое письмо.

(У файла почты может быть и другое имя; /usr/mail/you также очень распространено.)

Вероятно, самой используемой переменной окружения является та переменная, которая определяет, где оболочка ищет команды. Вспомните, что при поиске введенной команды оболочка сначала просматривает текущий каталог, потом каталог /bin и затем /usr/bin. Эта последовательность каталогов называется *путем поиска* и хранится в переменной окружения PATH. Если вас не устраивает значение пути поиска по умолчанию, можно изменить его, опять-таки в файле .profile. Например, добавим к стандартному пути еще /usr/games:

```
PATH=.:/bin:/usr/bin:/usr/games          Один из способов ...
```

Синтаксис немного необычный: последовательность каталогов, разделенных двоеточиями. Помните, что точка (.) – это обозначение текущего каталога. Этот символ можно опустить – пустой компонент в PATH подразумевает текущий каталог.

Еще один способ установить PATH для примера, приведенного выше, состоит в том, чтобы просто увеличить предыдущее значение переменной:

```
PATH=$PATH:/usr/games                    ... Другой способ
```

Значение любой переменной окружения можно получить, если предварить ее имя символом $. В только что рассмотренном примере выражение $PATH извлекает текущее значение, к которому добавляется новая часть, и результат снова присваивается переменной PATH. Можно проверить это при помощи программы echo:

```
$ echo PATH is $PATH
PATH is :/bin:/usr/bin:/usr/games
$ echo $HOME                             Регистрационный каталог
/usr/you
$
```

Если у вас уже есть собственные программы, можно собрать их в одном каталоге и также добавить его к пути поиска. В этом случае PATH может выглядеть следующим образом:

```
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
```

О создании собственных программ поговорим в главе 3.

Еще одна переменная, часто применяемая более замысловатыми, чем ed, текстовыми редакторами, – это TERM, которая определяет тип используемого терминала. Такая информация может позволить программе более эффективно распоряжаться экраном. Поэтому можно добавить, например

```
TERM=adm3
```

в файл .profile.

Переменные можно использовать для создания аббревиатур. Если часто приходится ссылаться на каталог с чрезвычайно длинным именем, то, вероятно, стоит добавить следующую строку

```
d=/horribly/long/directory/name
```

в свой `.profile`, тогда можно будет говорить просто

```
$ cd $d
```

Персональные переменные типа `d` принято записывать в нижнем регистре, чтобы отличать их от переменных самой оболочки, например `PATH`.

В заключение необходимо сообщить оболочке, что переменные должны использоваться и в других программах; это реализуется командой `export`, о которой будет рассказано в главе 3.

```
export MAIL PATH TERM
```

В итоге типичный файл `.profile` может выглядеть так:

```
$ cat .profile
stty erase '^h' -tabs
MAIL=/usr/spool/mail/you
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
TERM=adm3
b=$HOME/book
export MAIL PATH TERM b
date
who | wc -l
$
```

Мы рассмотрели далеко не все возможности, предоставляемые оболочкой. Одно из наиболее полезных свойств – это возможность создания собственных команд посредством комбинирования уже существующих команд в файле, обрабатываемом оболочкой. Удивительно, как много можно достичь с помощью этого замечательно простого механизма. Разговор о нем начнется в главе 3.

1.5. Оставшаяся часть системы UNIX

Система UNIX – это нечто гораздо большее, чем описано в этой главе, да и во всей книге. К настоящему моменту вы должны иметь представление о системе и, в частности, о руководстве (man). С конкретными вопросами о том, как и когда использовать команды, обращайтесь именно к нему.

Полезно также периодически пролистывать руководство, чтобы освежить знания об известных командах и познакомиться с новыми. Руководство описывает множество программ, не упомянутых в этой книге, в том числе компиляторы для таких языков, как ФОРТРАН 77; про-

грамму-калькулятор: bc(1); команды cu(1) и uucp(1) для межкомпьютерного общения; графические пакеты; статистические программы и такие эзотерические, как units(1).

Как уже говорилось, эта книга не заменяет собой руководство, а дополняет его. В последующих главах будут рассмотрены составляющие части системы UNIX и ее программы. Начнем с информации, представленной в руководстве, и будем двигаться дальше, изучая связи компонентов между собой. И хотя в руководстве нигде явно не говорится о взаимосвязях программ, именно они определяют структуру программной среды UNIX.

История и библиография

Первой книгой, посвященной системе UNIX, была «The UNIX Time-sharing System» (UNIX – система с разделением времени) Д. М. Ритчи (D. M. Ritchie) и К. Л. Томпсона (K. L. Thompson), изданная Communications of the ACM (CACM) в июле 1974 года и переизданная там же в январе 1983. (Страница 89 репринта включена в публикацию в марте 1983 года.) Это обзор системы для людей, интересующихся операционными системами, который стоит того, чтобы его прочитал каждый программист.

Специальный выпуск Bell System Technical Journal (BSTJ), посвященный системе UNIX (июль 1978), содержит много статей о дальнейшем развитии системы, а также ретроспективные материалы, в том числе обновленную редакцию оригинального документа CACM, написанного Ритчи и Томпсоном. Второй специальный выпуск BSTJ, содержащий новые материалы о UNIX, был опубликован в 1984.

В статье «The UNIX Programming Environment» (Среда программирования UNIX) Б. В. Кернигана (B. W. Kernighan) и Д. К. Мэши (J. R. Mashey), появившейся в апреле 1981 года в IEEE Computer Magazine, сделана попытка рассказать программистам об основных возможностях системы.

Любое издание справочного руководства по UNIX (UNIX Programmer's Manual) приводит список команд, стандартных функций и интерфейсов, форматов файлов и рассказывает о техническом обслуживании. Вам не удастся долго обходиться без него, хотя, чтобы начать программировать, достаточно прочитать только часть тома 1. Первый том руководства по седьмой версии системы опубликован издательством Holt, Rinehart and Winston.

Том 2 справочного руководства по UNIX называется «Documents for Use with the UNIX Time-sharing System» (Документы, используемые вместе с системой разделения времени UNIX), он содержит учебники и справочные руководства для основных команд. В частности, в нем довольно подробно описываются программы подготовки документов и

средства разработки программ. Со временем вам захочется прочесть обо всем этом.

«UNIX Primer» (UNIX для начинающих) Энн и Нико Ломато (Ann and Nico Lomuto), изданный в 1983 году Prentice-Hall, – это хороший учебник, знакомящий с системой «зеленых» новичков, главным образом, непрограммистов.

2

Файловая система

Все, что ни есть в системе UNIX, – это файл. И это не такое уж чрезмерное упрощение, как может показаться на первый взгляд. Когда разрабатывалась первая версия системы, еще до того, как она получила название, было решено сконцентрировать усилия на создании простой и удобной в использовании структуры файловой системы. И своему успеху и удобству применения UNIX в большой степени обязан именно файловой системе. Она представляет собой один из лучших примеров воплощения философии «простоты», показывая, какой мощи можно достичь аккуратной реализацией небольшого количества хорошо продуманных идей.

Чтобы уверенно чувствовать себя при изучении команд и их взаимосвязей, необходимо иметь представление об основах структуры и функционирования файловой системы. В этой главе обсуждается большинство аспектов использования файловой системы: что такое файлы, чем они представлены, каталоги и иерархия файловой системы, права доступа, индексные дескрипторы (внутренняя системная запись файлов, inodes) и файлы устройств. Так как практически всегда использование системы UNIX связано с обработкой файлов, существует множество команд для получения информации о файлах и их трансформации; самые употребительные из них представлены в этой главе.

2.1. Основы

Файл – это последовательность байтов. (Байт – это небольшой фрагмент информации, состоящий из 8 бит. В этой книге под байтом мы будем понимать эквивалент символа.) Система не навязывает файлу какую-то определенную структуру и никоим образом не пытается трактовать его

содержание, смысл байтов зависит только от программ, которые интерпретируют файл. Кроме того, будет показано, что это утверждение верно не только для дисковых файлов, но и для файлов периферийных устройств. Магнитные ленты, почтовые сообщения, символы, вводимые с клавиатуры, вывод принтера, поток данных в программном канале – каждый из этих файлов представляет собой просто последовательность байтов по отношению к системе и ее программам.

Лучший способ изучения файлов – практика, так что давайте начнем с создания небольшого файла:

```
$ ed
a
now is the time
for all good people
.
w junk
36
q
$ ls -l junk
-rw-r--r-- 1 you          36 Sep 27 06:11 junk
$
```

Создан файл `junk`, в нем 36 байт – 36 символов, которые были введены с клавиатуры в режиме добавления (естественно, не считая исправлений ошибок ввода). Чтобы просмотреть файл, введите

```
$ cat junk
now is the time
for all good people
$
```

Команда `cat` показывает, как выглядит файл. Команда `od` (octal dump – восьмеричный дамп) выводит видимое представление всех байтов файла:

```
$ od -c junk
 0000000   n   o   w       i   s       t   h   e       t   i   m   e  \n
 0000020   f   o   r       a   l   l       g   o   o   d       p   e   o
 0000040   p   l   e  \n
 0000044
 $
```

Параметр `-c` означает «интерпретировать байты как символы». Если задан параметр `-b`, то будет показано и восьмеричное представление байтов:[1]

[1] Каждый байт файла – это число, достаточно большое для кодирования печатных символов. В большинстве UNIX-систем принята кодировка ASCII, некоторые же машины, в частности выпущенные IBM, пользуются кодировкой EBCDIC. В этой книге везде подразумевается кодировка ASCII; введите `cat /usr/pub/ascii` или обратитесь к `ascii(7)`, чтобы просмотреть восьмеричные значения всех символов.

```
$ od -cb junk
0000000   n   o   w       i   s       t   h   e       t   i   m   e  \n
        156 157 167 040 151 163 040 164 150 145 040 164 151 155 145 012
0000020   f   o   r       a   l   l       g   o   o   d       p   e   o
        146 157 162 040 141 154 154 040 147 157 157 144 040 160 145 157
0000040   p   l   e  \n
        160 154 145  012
0000044
$
```

Семизначные числа слева обозначают местоположение в файле – порядковый номер следующего выведенного символа, в восьмеричной записи. Между прочим, то, что восьмеричным числам придано такое значение, – это наследие PDP-11, для которой предпочтительной была восьмеричная нотация. Для других машин больше подходит шестнадцатеричная запись – параметр `-x` позволяет выводить в таком виде.

Обратите внимание, что после каждой строки встречается символ с восьмеричным значением `012`. Это ASCII-символ *новой строки*, то есть то, что система помещает в поток ввода, когда пользователь нажимает клавишу *Return*. По соглашению, заимствованному из Си, представление символа новой строки выглядит как `\n`, но это просто условность, используемая такими программами, как `od`, для удобства чтения, на самом же деле величина, хранимая в файле, – это один байт `012`.

Символ новой строки – это наиболее часто употребляемый *специальный символ*. Другие символы, сопоставленные некоторым операциям управления терминалом, включают в себя возврат на одну позицию (восьмеричное значение `010`, выводится как `\b`), табуляцию (`011`, `\t`) и возврат каретки (`015`, `\r`).

Очень важно различать значение, сохраняемое для символа в файле, и то, как он интерпретируется в различных ситуациях. Например, когда с клавиатуры вводится возврат на одну позицию (*Backspace*), если этот символ в конкретной системе служит для удаления, ядро интерпретирует его как отказ от предыдущего введенного символа. В итоге и символ возврата, и символ, стоявший перед ним, исчезают, но введение возврата на одну позицию отражается на терминале, курсор перемещается на одну позицию назад.

Если же ввести последовательность

```
\←
```

(т. е. сначала `\`, а потом символ возврата на одну позицию), то ядро интерпретирует это не как введение специального символа возврата, а как желание поместить символ возврата в поток ввода, поэтому символ `\` отбрасывается, а байт `010` помещается в файл. При отображении символа возврата на терминале курсор перемещается на место символа `\`.

При *выводе* файла, содержащего символ возврата, последний не интерпретируется терминалом, а вызывает перемещение курсора назад.

Если файл выведен командой od, символ возврата представлен значением 010 или \b , если указан параметр -c.

Это же относится и к табуляции: при вводе символ табуляции отображается на терминале и пересылается программе, выполняющей чтение, при выводе он отправляется на терминал и там интерпретируется. Но есть и различие – *ядро* может быть настроено таким образом, что будет заменять при выводе символы табуляции соответствующим количеством пробелов. Символ табуляции устанавливает курсор в позициях 9, 17, 25 и т. д. Команда

```
$ stty -tabs
```

вызывает замену символов табуляции пробелами *при выводе на терминал* (stty(1)).

Аналогично обрабатывается и *Return*. Ядро отображает на терминале нажатие клавиши *Return* как символ возврата каретки и разделитель строк, а в файл отправляет только символ новой строки. При выводе перед символом новой строки добавляется возврат каретки.

Подход, реализованный в системе UNIX для представления управляющей информации, не традиционен, в частности в том, что касается использования символов новой строки в конце строк. Вместо этого во многих системах существуют «записи», по одной на строку, каждая из которых содержит не только введенные данные, но и подсчитанное количество символов в строке (и никаких разделителей строк). Другие системы заканчивают строку символами возврата каретки *и* новой строки, потому что именно такая последовательность требуется для вывода на большинстве терминалов. (Понятие «переход на другую строку» (linefeed) обозначает то же самое, что и разделитель строк, поэтому последовательность, упомянутую выше, часто называют «CR-LF» (carriage return line feed), что удобнее для произношения.)

Система UNIX не делает ни того, ни другого – нет ни записей, ни подсчета символов, и файл не содержит байты, которые пользователь или его программы не помещали в него. При выводе на терминал разделитель строк дополняется возвратом каретки, но программам достаточно обрабатывать только символ новой строки, т. к. они знают только о нем. В большинстве случаев требуется именно такое поведение. При необходимости на основе такой простой схемы можно построить и более сложную. Обратного, то есть создания простой схемы на базе более сложной, достичь затруднительно.

Поскольку конец каждой строки отмечен символом новой строки, можно было бы ожидать, что и файлы должны оканчиваться каким-либо специальным символом, скажем \e (от англ. end – конец). Но, изучая вывод команды od, вы не найдете символа конца файла, он просто заканчивается. Вместо использования специального кода система подписывает конец файла, просто сообщая, что данных больше

нет. Ядро хранит сведения о длинах файлов, поэтому программа обнаруживает конец файла, когда обработает его последний байт.

Программа получает данные файла, используя системный вызов (подпрограмму ядра) `read`. При каждом вызове `read` возвращает очередную порцию данных – следующую строку текста, вводимого с терминала, к примеру. Кроме того, она сообщает, сколько байтов файла было получено, так что, получив от `read` сообщение «получено ноль байтов», можно считать, что достигнут конец файла. Если же какие-либо байты еще остались, `read` вернет часть из них. Действительно, нет смысла вводить специальный символ конца файла, потому что, как было уже сказано, смысл байта зависит от способа его интерпретации. Тем не менее, *все* файлы должны где-нибудь кончаться, а так как все они должны быть доступны для чтения функцией `read`, то возврат нуля представляет собой независимый от интерпретации способ указания конца файла без использования дополнительных символов.

Когда программа читает данные с терминала, ядро передает ей каждую вводимую строку только после ввода символа новой строки (т. е. при нажатии *Return*). Поэтому, пока символ новой строки не введен, есть возможность вернуться и исправить замеченную опечатку. Если же ошибка обнаружена после нажатия клавиши *Return*, то исправлять ее поздно, так как строка уже прочитана программой.

С помощью команды `cat` можно наблюдать, как работает такой построчный ввод. Обычно эта команда для повышения эффективности *буферизует* вывод для повышения производительности, чтобы выполнять запись крупными блоками, но параметр `-u` позволяет отключить буферизацию записи, и тогда прочитанные строки немедленно отправляются на вывод:

```
$ cat                    Вывод с буферизацией
123
456
789
ctl-d
123
456
789
$ cat -u                 Вывод без буферизации
123
123
456
456
789
789
ctl-d
$
```

При нажатии клавиши *Return* `cat` получает очередную строку; при отключенной буферизации полученные данные выводятся сразу.

Теперь попробуем иначе: введем несколько символов, а вместо *Return* нажмем *ctl*-d:

```
$ cat -u
123ctl-d123
```

Символы выводятся сразу после нажатия комбинации *ctl*-d, которая означает «немедленно отправить напечатанные символы программе, выполняющей чтение данных с терминала». В отличие от символа новой строки, *ctl*-d не передается в программу. Теперь, не вводя ничего больше, нажмем *ctl*-d дважды:

```
$ cat -u
123ctl-d123ctl-d$
```

Оболочка выводит приглашение на ввод, поскольку команда cat, не получив символов, считает, что достигнут конец файла, и завершается. По нажатию *ctl*-d все, что было введено на терминале, отправляется программе, выполняющей чтение с терминала. Если ничего не введено, то программа, не получив очередной порции символов, воспримет это как конец файла. Именно поэтому нажатие *ctl*-d завершает сеанс – оболочка считает, что ввод окончен. Конечно, символ *ctl*-d обычно служит для извещения об окончании файла, но интересно и его более общее назначение.

Упражнение 2.1. Что произойдет при нажатии *ctl*-d в редакторе ed? Сравните с командой

```
$ ed <file
```

□

2.2. Что в файле?

Формат файла определяется использующей его программой; поэтому, вероятно, большое количество типов файлов определяется разнообразием программ. Но, поскольку файловые типы в файловой системе не определены, то и ядро ничего о них не знает. Команда file делает предположение о типе файла (вкратце объясним, как она это делает):

```
$ file /bin /bin/ed /usr/src/cmd/ed.c /usr/man/man1/ed.1
/bin:   directory
/bin/ed:        pure executable
/usr/src/cmd/ed.c:      c program text
/usr/man/man1/ed.1:     roff, nroff, or eqn input text
$
```

Здесь представлены четыре типичных файла, относящиеся к редактору: каталог, в котором они расположены (/bin), собственно двоичный исполняемый файл (/bin/ed), файл с исходным текстом на C (/usr/src/cmd/ed.c) и страница руководства (/usr/man/man1/ed.1).

Для определения типа файла команда `file` не использует его имя (хотя это возможно), так как соглашения об именовании – это всего лишь соглашения, и они не дают стопроцентной гарантии. Например, файлы с расширением `.c` почти всегда содержат исходные тексты на Си, но ничто не мешает создать такой файл с произвольным содержимым. Вместо этого `file` читает первые несколько сотен байт и ищет в них ключевые последовательности символов. (Как будет показано ниже, специальные системные файлы, такие как каталоги, могут быть определены путем запроса к системе, но команда `file` может идентифицировать каталог при его чтении.)

Иногда ключи очевидны. Исполняемая программа начинается с «магического числа» в двоичном представлении. Команда `od` без параметров выводит дамп файла 16-битными порциями или 2-байтовыми словами, позволяя увидеть это волшебное число:[1]

```
$ od /bin/ed
0000000 000410 025000 000462 011444 000000 000000 000000 000001
0000020 170011 016600 000002 005060 177776 010600 162706 000004
0000040 016616 000004 005720 010066 000002 005720 001376 020076
...
$
```

Восьмеричное значение `410` означает обычную исполняемую программу, такую, которая позволяет разделять исполняемый код нескольким процессам. (Конкретные значения магических чисел зависят от системы.) Восьмеричному значению `410` не соответствует никакой ASCII-символ, поэтому оно не может быть случайно создано в такой программе, как редактор. Но, разумеется, ничто не мешает создать такой файл с помощью собственной программы, и тогда система, в соответствии с принятым соглашением, будет считать его исполняемой программой.

В текстовых файлах ключи могут встречаться среди прочего текста, поэтому `file` определяет исходные тексты на Си по наличию таких слов, как `#include`, а входные данные программ `nroff` или `troff` – по наличию точки в начале строки.

Может возникнуть вопрос, почему система не следит за типами файлов более тщательно, чтобы, например, не допустить применения команды `sort` к файлу `/bin/ed`. Одна из причин – нежелание запрещать некоторые полезные действия. Хотя команда

```
$ sort /bin/ed
```

не имеет большого смысла, существует множество команд, обрабатывающих файлы любых типов, и нет оснований ограничивать их ис-

[1] В современных реализациях UNIX получили распространение более совершенные и сложные форматы двоичных исполняемых файлов, но, тем не менее, `find` для их определения следует изложенным здесь принципам. – *Примеч. науч. ред.*

пользование. Такие команды, как od, wc, cp, cmp, file и многие другие, работают с файлами независимо от их содержимого. Но идея бесформатных файлов еще глубже. Если бы, к примеру, входные файлы nroff отличались по формату от исходных текстов на Си, редактору пришлось бы вставлять эти различия при создании файла и обрабатывать их при чтении и повторном редактировании. А это определенно затруднило бы авторам создание примеров на Си в главах с 6 по 8!

Вместо того чтобы создавать различия, UNIX старается избегать их. Все текстовые файлы состоят из строк, заканчивающихся символом новой строки, и большинство программ понимают этот простой формат. В процессе написания этой книги авторы многократно запускали команды создания текстовых файлов, обрабатывали их командами, аналогичными упомянутым выше, затем использовали редактор для их объединения во входной файл программы troff. Листинги, встречающиеся почти на каждой странице, сформированы командами вида

```
$ od -c junk >temp
$ ed ch2.1
1534
r temp
168
...
```

Команда od выводит текст, который может быть прочитан везде, где только допустима работа с текстом. Такое единообразие несколько необычно; большинство систем даже для текста имеют несколько различных форматов и требуют дополнительных сведений от программы или пользователя для создания файла определенного типа. В системах UNIX существуют файлы лишь одного вида, и все, что требуется для доступа к файлу – это его имя.[1]

По большому счету отсутствие файловых форматов является преимуществом – программистам не надо заботиться о типах файлов, а все стандартные программы способны обрабатывать любые файлы, но есть и ряд недостатков. Программы сортировки, поиска и редактирования ожидают получить на входе текст: grep некорректно обрабатывает двоичные файлы, sort не может их сортировать, они не редактируются ни одним из стандартных редакторов.

Реализация большинства программ, работающих с текстом, накладывает свои ограничения. Авторы проверили ряд программ на текстовом файле длиной 30 000 байт, не содержащем символов новой строки, и выяснили, что лишь немногие из них работают правильно, в то время

[1] Даг Мак-Илрой (Doug McIlroy) предложил хороший тест для проверки единообразия файловой системы – UNIX прошла его без проблем. Может ли вывод программы, написанной на Фортране, быть использован как вход для компилятора Фортрана? Подавляющее большинство систем испытывают затруднения при прохождении этого теста.

как в большинстве из них сделаны неявные предположения о максимальной длине текстовой строки (см., например, раздел BUGS страницы руководства sort(1)).

Конечно, нетекстовые файлы тоже необходимы. Например, для повышения эффективности большие базы данных должны хранить дополнительную адресную информацию в двоичном виде. Но любой файловый формат, отличный от текстового, требует целого семейства программ, выполняющих действия, для которых в случае текста было бы достаточно стандартного инструментария. Меньшая эффективность обработки текстовых файлов компенсируется сокращением расходов на дополнительное программное обеспечение, поддерживающее специализированные форматы. При разработке нового формата файлов следует хорошенько подумать, выбирая нетекстовое представление. (Кроме того, обязательно должна быть предусмотрена обработка длинных входных строк.)

2.3. Каталоги и имена файлов

Все принадлежащие пользователю файлы имеют имена, начинающиеся с /usr/you, но если ваш единственный файл называется junk, то, выполнив команду ls, вы не увидите имени /usr/you/junk; имя файла напечатается без префикса:

```
$ ls
junk
$
```

Это происходит потому, что каждая выполняющаяся программа, то есть каждый процесс, имеет *текущий каталог* и подразумевается, что все имена файлов начинаются с имени этого каталога, если только в начале имени не задан явно символ косой черты /. Следовательно, и у оболочки, в которую пользователь попадает при входе в систему, и у команды ls есть текущий каталог. Команда pwd (print working directory – печатать текущий каталог) выводит имя текущего каталога:

```
$ pwd
/usr/you
$
```

Текущий каталог является атрибутом процесса, а не пользователя или программы – у пользователей есть регистрационные каталоги, а у процессов – текущие. Если процесс порождает другой процесс, то потомок наследует текущий каталог своего родителя. Но если дочерний процесс переходит в другой каталог, на родителе это никак не отражается – его текущий каталог останется прежним, что бы ни делали его потомки.

Понятие текущего каталога введено во многом для удобства обозначения, ведь оно позволяет значительно сократить объем вводимого текс-

та, но его основное назначение – организационное. Родственные файлы размещаются в одном каталоге. Обычно каталог /usr служит вершиной пользовательской файловой системы. (Слово user сокращено до usr в духе cmp, ls и т. п.) Каталог /usr/you представляет собой регистрационный, то есть текущий каталог при первом[1] входе в систему. В /usr/src находятся исходные тексты системных программ, в /usr/src/cmd – исходные тексты команд UNIX, в /usr/src/cmd/sh – тексты оболочки и т. д. Начиная новый проект или просто размещая группу файлов, например кулинарных рецептов, можно создать новый каталог командой mkdir и разместить там файлы.

```
$ pwd
/usr/you
$ mkdir recipes
$ cd recipes
$ pwd
/usr/you/recipes
$ mkdir pie cookie
$ ed pie/apple
...
$ ed cookie/choc.chip
...
$
```

Обратите внимание на то, как просто выполняется ссылка на подкаталог. Смысл pie/apple очевиден: рецепт яблочного пирога, расположенный в каталоге, /usr/you/recipes/pie.[2] Можно было бы поместить этот рецепт не в подкаталог каталога рецептов, а, скажем, в recipes/apple.pie, но логичнее все-таки хранить все рецепты пирогов в отдельном месте. Например, рецепт приготовления глазури можно поместить в recipes/pie/crust вместо того, чтобы дублировать его в рецепте каждого пирога.

Файловая система представляет собой мощное средство организации, но пользователь может забыть и то, куда он положил файл, и то, какие вообще файлы у него есть. Решить эту проблему поможет пара команд, просматривающих каталоги. Команда ls весьма полезна при поиске файлов, но она не просматривает подкаталоги.[3]

```
$ cd
$ ls
```

---

[1] На самом деле и при всех последующих входах тоже; почему автор написал «при первом», непонятно, речь идет явно о текущем каталоге *сразу* после регистрации на машине. – *Примеч. науч. ред.*

[2] В переводе с английского recipes – рецепты, cookie – печенье, pie – пирог, apple – яблоко, crust – глазурь. – *Примеч. перев.*

[3] Большинство современных реализаций UNIX допускают использование команды с параметром ls -R для рекурсивного просмотра подкаталогов. – *Примеч. науч. ред.*

```
junk
recipes
$ file *
junk:   ascii text
recipes:        directory
$ ls recipes
cookie
pie
$ ls recipes/pie
apple
crust
$
```

Эту часть файловой системы можно представить графически (рис. 2.1).

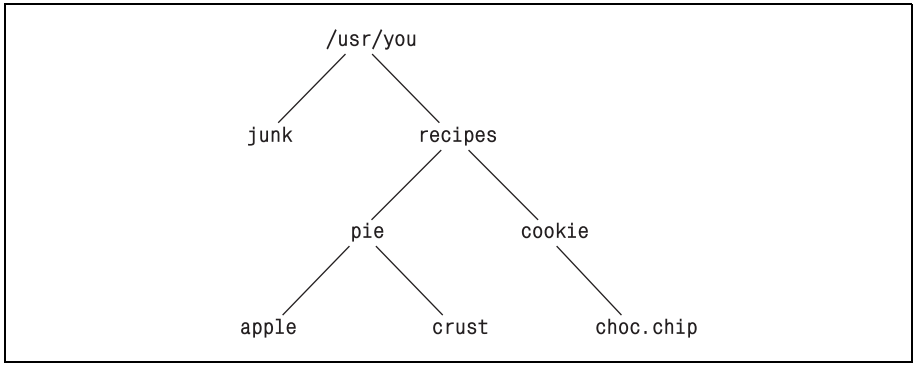

Рис. 2.1. *Графическое представление файловой системы*

Команда `du` (disc usage – использование диска) предназначена для определения дискового пространства, занимаемого файлами каталога, включая все подкаталоги.

```
$ du
6       ./recipes/pie
4       ./recipes/cookie
11      ./recipes
13      .
$
```

В столбце слева указано количество дисковых блоков (обычно размером 512 или 1024 байт каждый), занимаемых каждым из файлов. Значение в строке каталога показывает количество блоков, занятых всеми файлами каталога и его подкаталогами, включая и сам файл каталога.

Команда `du` имеет параметр `-a` (all – все), при задании которого выводятся все файлы каталога. Если среди файлов есть каталоги, то `du` обработает и их:

```
$ du -a
2       ./recipes/pie/apple
3       ./recipes/pie/crust
6       ./recipes/pie
3       ./recipes/cookie/choc.chip
4       ./recipes/cookie
11      ./recipes
1       ./junk
13      .
$
```

Вывод команды `du -a` можно перенаправить на `grep`, если требуется найти определенный файл:

```
$ du -a | grep choc
3       ./recipes/cookie/choc.chip
$
```

В главе 1 уже рассказывалось о том, что имя «`.`» относится к элементу каталога, ссылающемуся на сам каталог, что позволяет обращаться к каталогу, не зная его полного имени.

Команда `du` просматривает каталог в поисках файлов; если имя каталога не указано, то подразумевается «`.`», то есть текущий каталог. Таким образом, `junk` и `./junk` — это имена одного и того же файла.

Каталоги, несмотря на свои особые свойства с точки зрения ядра, располагаются в файловой системе, как обычные файлы. Они могут быть прочитаны, как и все файлы. Но в отличие от остальных, файлы каталогов не могут быть созданы или изменены обычными средствами; для сохранения целостности файловой системы ядро оставляет за собой контроль над содержимым этих файлов.

Настало время взглянуть на байты внутри каталога:

```
$ od -cb .
0000000   4   ;   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        064 073 056 000 000 000 000 000 000 000 000 000 000 000 000 000
0000020 273   (   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        273 050 056 056 000 000 000 000 000 000 000 000 000 000 000 000
0000040 252   ;   r   e   c   i   p   e   s  \0  \0  \0  \0  \0  \0  \0
        252 073 162 145 143 151 160 145 163 000 000 000 000 000 000 000
0000060 230   =   j   u   n   k  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        230 075 152 165 156 153 000 000 000 000 000 000 000 000 000 000
0000100
$
```

Видите имена файлов, «зашитые» внутри? Формат каталога совмещает двоичные и текстовые данные. Данные расположены порциями по 16 байт, последние 14 из которых содержат имя файла, дополненное ASCII-символами NUL (со значением `0`), а первые два сообщают системе, где находится служебная информация о данном файле (вернемся к

этому позже).[1] Каждый каталог начинается с пары записей: «.» (точка) и «..» (точка-точка).

```
$ cd                  Домашний каталог
$ cd recipes
$ pwd
/usr/you/recipes
$ cd ..; pwd          На один уровень вверх
/usr/you
$ cd ..; pwd          Еще на уровень вверх
/usr
$ cd ..; pwd          Еще на уровень вверх
/
$ cd ..; pwd          Еще на уровень вверх
/                     Выше идти некуда
$
```

Каталог / называется *корневым* каталогом системы. Все файлы в системе находятся либо в корневом каталоге, либо в его подкаталогах, при этом корневой каталог является родительским сам для себя.

Упражнение 2.2. На *основании* информации этого раздела можно в общих чертах представить себе механизм работы команды `ls`. Подсказка: `cat . >foo; ls -f foo`. □

Упражнение 2.3. (более сложное). Как работает команда `pwd`? □

Упражнение 2.4. Команда `du` была создана для слежения за использованием диска. Ее применение для поиска файлов в каталоге представляется довольно странным и, возможно, неправильным. В качестве альтернативы изучите страницу руководства `find(1)` и сравните эти две команды. В частности, сравните команду `du -a | grep ...` с соответствующим вызовом `find`. Какая из них работает быстрее? Что лучше – написать новую программу или использовать побочный эффект существующей? □

2.4. Права доступа

Каждый файл имеет связанный с ним набор *прав*, определяющих, кто и что может делать с этим файлом. Если вы настолько дисциплинированны, что храните в системе свою любовную переписку, может быть,

[1] В качестве примера автор приводит дамп файла каталога в том виде, какой мы получили бы в старых версиях UNIX, использующих s5fs (файловую систему, изначально применявшуюся в UNIX System V). Реализации файловых систем современных версий отошли от записей фиксированной длины (в них формат файла каталога более сложный) и не ограничивают длину имени файла 14-ю символами, как это было когда-то. Так, FFS (Fast File System), появившаяся впервые в BSD 4.2, допускает уже 255-символьные имена файлов. – *Примеч. науч. ред.*

даже рассортированную по каталогам, вы, вероятно, не захотите, чтобы она стала доступна коллегам. Тогда следует изменить права доступа ко всем письмам, чтобы пресечь распространение слухов (или не ко всем, – чтобы подогреть их), или же установить права доступа к каталогу, содержащему письма, чтобы уберечься от излишне любопытных сотрудников.

Но предупреждаем: в каждой системе существует специальный пользователь, называемый *суперпользователем* (*super-user*), который может читать и модифицировать *любые* файлы в системе. Права суперпользователя присвоены регистрационному имени `root`, под которым администраторы управляют системой.[1] Существует команда `su`, предоставляющая статус суперпользователя каждому, кто знает пароль. Любой обладатель такого пароля получит доступ к вашим любовным письмам, поэтому не стоит хранить в системе столь секретные материалы.

Для сохранения конфиденциальности можно применять команду `crypt` (`crypt(1)`), которая шифрует данные таким образом, что и суперпользователь будет не в состоянии их прочитать (или, по крайней мере, понять). Конечно, и команда `crypt` не абсолютно надежна. Суперпользователь может подменить ее, а шифр может быть взломан. Но первое является должностным преступлением, а второй способ весьма трудоемок, так что на практике эта команда достаточно безопасна.

На практике большинство брешей в системе безопасности возникает из-за применения слишком простых паролей или даже их отсутствия. Именно ошибки администрирования чаще всего дают возможность злоумышленнику получить права суперпользователя. В конце этой главы упомянут ряд работ, посвященных вопросам безопасности.

В процессе регистрации в системе пользователь вводит имя, а затем при помощи пароля подтверждает, что он именно тот, за кого себя выдает. Это имя называется *регистрационным идентификатором* (*login-id*). В действительности система опознает пользователя по номеру uid – *идентификатору пользователя*. Фактически, несколько регистрационных идентификаторов могут иметь одно и то же значение uid, что не позволяет системе отличить их друг от друга, хотя это случай довольно редкий и нежелательный с точки зрения безопасности.

---

[1] Если быть более точным, то права суперпользователя будут делегированы системой любому процессу пользователя, эффективный пользовательский идентификатор которого равен нулю. Но имя учетной записи пользователя и его идентификатор, получаемый системой из этой учетной записи при входе пользователя в систему, никак не связаны в действительности. Поэтому вполне может оказаться, что в какой-то системе администратор поменял имя регистрационной записи с нулевым идентификатором на что-то другое, отличное от `root`. Однако назначение такого имени для учетной записи суперпользователя – многолетняя традиция, а слова «root» и «суперпользователь» воспринимаются как синонимы. – *Примеч. науч. ред.*

Помимо идентификатора uid пользователю присваивается *идентификатор группы* (*group-id*), определяющий некоторый класс пользователей. Во многих системах обычные пользователи (в отличие от таких, как `root`) помещаются в одну группу, называемую «остальные» (other), но возможно, что в вашей системе приняты другие правила. Файловая система, а значит и система UNIX в целом, на основе значений uid и group-id, выданных пользователю, определяет, какие действия ему разрешены.

Пароли хранятся в файле `/etc/passwd`; там же находится вся регистрационная информация каждого из пользователей. Можно узнать свои значения uid и group-id так же, как это делает система, найдя свое имя в файле `/etc/passwd`:

```
$ grep you /etc/passwd
you:gkmbCTrJO4COM:604:1:Y.O.A.People:/usr/you:
$
```

Поля в файле паролей разделены двоеточиями и расположены в таком порядке (в соответствии с `passwd(5)`):

login-id:encrypted-password:uid:group-id:miscellany:login-directory:shell

Файл содержит текстовые данные, но определения полей и разделителя представляют собой предмет соглашения между программами, использующими информацию, полученную из этого файла.

Поле *shell,* определяющее оболочку, часто остается пустым, подразумевая оболочку по умолчанию, то есть `/bin/sh`. Поле *miscellany* может содержать произвольные данные, часто в нем хранят имя, адрес или номер телефона.

Обратите внимание на то, что пароль, занимающий второе поле, хранится в зашифрованном виде. Любой пользователь может прочитать этот файл, и если бы пароль хранился незашифрованным, узнать его мог бы кто угодно. Когда пользователь вводит свой пароль в программе `login`, та шифрует его и сравнивает результат с образцом из `/etc/passwd`. В случае совпадения разрешен вход в систему. Такой механизм эффективен благодаря тому, что алгоритм шифрования работает достаточно быстро, а дешифрование требует значительных усилий. Например, пароль `ka-boom` может в зашифрованном виде выглядеть как `gkmbCTrJO4COM`, но восстановить по такой строке оригинал весьма непросто.

Прочитать файл `/etc/passwd` оказалось возможным потому, что ядро, проверив права доступа к нему, выдало разрешение на чтение. Для каждого файла существуют три вида доступа: чтение (просмотр содержимого), запись (изменение содержимого) и исполнение (запуск программы). Кроме того, разные пользователи могут иметь различные права доступа. Один набор разрешений связан с владельцем файла, другой набор – с группой, в которую входит владелец, а третий определяет разрешения на доступ для всех остальных.

Команда ls, запущенная с параметром -l, выводит, кроме прочего, информацию о правах доступа:

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root       5115 Aug 30 10:40 /etc/passwd
$ ls -lg /etc/passwd
-rw-r--r-- 1 adm        5115 Aug 30 10:40 /etc/passwd
$
```

В этих двух строках содержится следующая информация: файл /etc/passwd принадлежит пользователю root из группы adm, имеет длину 5115 байт, время последнего изменения 10:40 30 августа, имеет одну ссылку (одно имя в файловой системе; ссылки обсуждаются в следующем разделе). Некоторые версии ls выводят информацию одновременно о пользователе и о группе.

В строке -rw-r--r-- представлены сведения о правах доступа к файлу. Первый знак - говорит о том, что это обычный файл. Если бы это был каталог, то в первой позиции находился бы символ d. Следующие три символа показывают права владельца (определяемые по его uid) на чтение, запись и выполнение. Символы rw- означают, что root (владелец файла) может читать и писать, но не может выполнять этот файл. У исполняемых файлов вместо прочерка должен стоять символ x.

Следующие три символа (r--) показывают разрешения для группы, в данном случае пользователи группы adm, по-видимому, системные администраторы, могут читать файл, но не имеют прав на запись и выполнение. Следующие три (также r--) определяют права доступа для остальных – всех прочих пользователей системы. Таким образом, на этой машине только root может изменять регистрационную информацию, а всем остальным она доступна для чтения. Одной из вероятных альтернатив было бы присвоение и группе adm привилегии на запись в /etc/passwd .

Файл /etc/group содержит имена групп и их идентификаторы и определяет принадлежность пользователей к группам. В /etc/passwd определена только первоначальная группа, которая назначается при регистрации; команда newgrp изменяет текущую группу пользователя.

Любой пользователь с помощью команды

```
$ ed /etc/passwd
```

может редактировать файл паролей, но только root может сохранить сделанные изменения. Возникает вопрос, как пользователь может изменить свой пароль, если для этого ему необходимо редактировать файл паролей? Это можно сделать при помощи команды passwd, находящейся обычно в каталоге /bin:

```
$ ls -l /bin/passwd
-rwsr-xr-x 1 root      8454 Jan  4  1983 /bin/passwd
$
```

(Учтите, что /etc/passwd представляет собой текстовый файл с регистрационной информацией, а /bin/passwd, расположенный в другом каталоге, является исполняемой программой, позволяющей изменить информацию о пароле.) В данном случае права доступа определяют, что команда доступна для выполнения всем, но только root может ее изменить. Символ s вместо x в поле прав на исполнение для владельца говорит о том, что команда во время выполнения приобретает права своего владельца, в данном случае root. А так как /bin/passwd получает при исполнении идентификатор пользователя root, любой пользователь с помощью этой команды может изменить файл паролей.

Назначение атрибута SUID (set-uid) – простая и изящная идея,[1] решающая ряд проблем безопасности. Например, автор игровой программы может присвоить ей исполнительный идентификатор владельца с тем, чтобы она могла обновлять текущий счет в файле, защищенном от прямого вмешательства других пользователей. Но концепция set-uid таит в себе и потенциальную опасность. Программа /bin/passwd должна работать правильно; в противном случае она может, действуя от имени суперпользователя, уничтожить системную информацию. Если права доступа к ней установлены в -rwsrwxrwx, то *любой* пользователь может заменить ее собственной программой. Это особенно критично для программ, имеющих атрибут SUID, так как пользователь root имеет права на доступ ко всем файлам системы. (Некоторые системы UNIX обнуляют бит SUID исполнительного идентификатора при модификации файла с тем, чтобы уменьшить риск появления бреши в системе безопасности.)

Бит SUID – мощное средство, но обычно оно используется лишь в некоторых системных программах, таких как passwd. Рассмотрим теперь обычный файл.

```
$ ls -l /bin/who
-rwxrwxr-x 1 root       6348 Mar 29  1983 /bin/who
$
```

Файл who доступен для исполнения всем, а для записи – владельцу root и группе владельца. Под «исполнением» здесь понимается следующее: при вводе команды

```
$ who
```

оболочка просматривает ряд каталогов, в том числе и /bin, в поисках файла who. Если она находит такой файл, и если файл имеет разрешение на выполнение, оболочка делает запрос на его выполнение к ядру. Ядро проверяет права доступа и, если они позволяют, запускает программу. Обратите внимание, что программа – это файл, имеющий разрешение на выполнение. В следующей главе мы рассмотрим програм-

[1] Идея атрибута set-uid принадлежит Деннису Ритчи (Dennis Ritchie).

мы, представляющие собой текстовые файлы, способные выполняться благодаря тому, что у них установлен соответствующий бит.

Права доступа на каталоги действуют немного иначе, хотя основная идея та же.

```
$ ls -ld .
drwxrwxr-x 3 you          80 Sep 27 06:11 .
$
```

Команда `ls` с параметром `-d` выводит сведения о самом каталоге, а не о его содержимом, а `d` в начале строки подтверждает, что «.» действительно является каталогом. Поле `r` означает, что каталог доступен для чтения, и его содержимое может быть просмотрено с помощью команды `ls` (или, при желании, `od`). Наличие `w` говорит о том, что возможно создание и удаление файлов в этом каталоге, поскольку эти действия требуют записи в файл каталога.

На самом деле вы не можете записывать данные прямо в файл каталога – это запрещено даже суперпользователю.

```
$ who >.              Попытка перезаписать «.»
.: cannot create      Невозможно
$
```

Для этой цели применяются системные вызовы, позволяющие создавать и удалять файлы, – только с их помощью можно вносить изменения в файл каталога. Но механизм прав доступа работает и здесь: поле `w` определяет, кто может использовать системные программы для изменения каталога.

Разрешение на удаление файла не связано с самим файлом. При наличии права на запись в каталог можно удалить файлы даже в том случае, если они защищены от записи. Команда `rm` запрашивает подтверждение перед тем, как удалить защищенный файл, – это один из тех редких случаев, когда UNIX просит пользователя подтвердить его намерения. (Флаг `-f` позволяет команде `rm` удалять файлы без лишних вопросов.)

Поле `x` в строке прав доступа к каталогу разрешает не исполнение, а поиск. Разрешение на исполнение для каталога означает, что в нем может быть выполнен поиск файла. Таким образом, можно создать каталог с правами `--x` для остальных пользователей, что позволит им получить доступ к тем файлам, которые им известны, но не даст выполнить команду `ls` для просмотра всего содержимого каталога. Аналогично, в каталоге с правами доступа `r-` пользователи могут просматривать его содержимое, но не могут с ним работать. В некоторых конфигурациях это свойство используется для отключения каталога `/usr/games` в рабочее время.

Команда `chmod` (change mode – изменить режим) изменяет права доступа к файлу.

```
$ chmod права-доступа имена-файлов ...
```

Правда, синтаксис *прав-доступа* несколько неуклюжий. Существуют два способа описания: в восьмеричном и в символьном виде. Восьмеричные значения легче в использовании, хотя иногда бывает удобно символьное представление, поскольку оно позволяет описать относительные изменения прав. Было бы удобно написать

```
$ chmod rw-rw-rw- junk          Этот способ не работает!
```

вместо

```
$ chmod 666 junk
```

но так нельзя. Восьмеричные значения образуются путем суммирования 4 для чтения, 2 для записи и 1 для исполнения. Тремя цифрами, как в ls, обозначаются разрешения для владельца, группы и остальных. Значения символьных кодов объяснить труднее, исчерпывающее описание имеется на странице руководства chmod(1). Для наших целей достаточно отметить, что знак + включает право доступа, а знак - его выключает. Например

```
$ chmod +x command
```

позволяет всем выполнять *command*, а

```
$ chmod -w file
```

отменяет право на запись для всех, включая владельца. Независимо от имеющихся разрешений, только владелец файла может изменить права доступа к нему – с учетом обычной оговорки о суперпользователях. Даже если кто-либо предоставит пользователю права на запись в файл, система не позволит ему изменить биты прав доступа

```
$ ls -ld /usr/mary
drwxrwxrwx 5 mary        704 Sep 25 10:18 /usr/mary
$ chmod 444 /usr/mary
chmod: can't change /usr/mary
$
```

В то же время, если каталог доступен для записи, пользователи могут удалять файлы независимо от прав доступа к этим файлам. Чтобы исключить возможность удаления файлов кем-либо, следует отменить разрешение на запись в каталог:

```
$ cd
$ date >temp
$ chmod -w .              Запретить запись в каталог
$ ls -ld .
dr-xr-xr-x 3 you          80 Sep 27 11:48 .
$ rm temp
rm: temp not removed      Невозможно удалить файл
$ chmod 775 .             Восстановить права доступа
$ ls -ld .
```

```
drwxrwxr-x 3 you          80 Sep 27 11:48 .
$ rm temp
$                         Запись в каталог разрешена
```

Теперь `temp` удален. Обратите внимание на то, что изменение прав доступа к каталогу не затрагивает дату его модификации. Дата модификации отражает изменения в содержании каталога, а не в его свойствах. Права доступа и даты хранятся не в самом файле, а в индексном дескрипторе (inode), которому посвящен следующий раздел.

Упражнение 2.5. Поэкспериментируйте с `chmod`. Попробуйте различные значения, такие как `0` и `1`. Будьте осторожны и не повредите свой регистрационный каталог! □

2.5. Индексные дескрипторы

Файл имеет несколько компонентов: имя, содержимое, служебную информацию, такую как права доступа и время модификации. Служебная информация хранится в индексном дескрипторе («inode», раньше было «i-node», но со временем дефис исчез) вместе с такой системной информацией, как размер файла, расположение его на диске, и прочими сведениями.

Индексный дескриптор хранит три значения времени: время последнего изменения (записи), время последнего использования (чтения или выполнения) и время последнего изменения самого индексного дескриптора, например при изменении прав доступа.

```
$ date
Tue Sep 27 12:07:24 EDT 1983
$ date >junk
$ ls -l junk
-rw-rw-rw- 1 you          29 Sep 27 12:07 junk
$ ls -lu junk
-rw-rw-rw- 1 you          29 Sep 27 06:11 junk
$ ls -lc junk
-rw-rw-rw- 1 you          29 Sep 27 12:07 junk
$
```

Как показывает команда `ls -lu`, изменение содержания файла не влияет на время его использования, а изменение прав доступа сказывается только на времени изменения индексного дескриптора, что видно из результатов выполнения команды `ls -lc`.

```
$ chmod 444 junk
$ ls -lu junk
-r--r--r-- 1 you          29 Sep 27 06:11 junk
$ ls -lc junk
-r--r--r-- 1 you          29 Sep 27 12:11 junk
$ chmod 666 junk
$
```

Команда `ls -t` сортирует файлы по времени, по умолчанию это время последнего изменения. Параметр `-t` может применяться совместно с параметром `-c` или `-u` для определения порядка изменения индексных дескрипторов или чтения файлов:

```
$ ls recipes
cookie
pie
$ ls -lut
total 2
drwxrwxrwx 4 you          64 Sep 27 12:11 recipes
-rw-rw-rw- 1 you          29 Sep 27 06:11 junk
$
```

Каталог `recipes` использовался последним, так как только что просматривалось его содержимое.

Понимание индексных дескрипторов необходимо не только с точки зрения параметров команды `ls`, но и потому, что можно сказать, что индексные дескрипторы – это *есть* файлы. Единственное назначение иерархии каталогов заключается в предоставлении удобных имен файлов. Внутреннее системное имя файла – это номер его индексного дескриптора (i-number), хранящего информацию о файле. Десятичные значения номеров индексных дескрипторов показывает команда `ls -i`:

```
$ date >x
$ ls -i
15768 junk
15274 recipes
15852 x
$
```

Перед именем файла, в первых двух байтах записи каталога, как раз и хранится номер его индексного дескриптора. Команда `od -d`, в отличие от восьмеричного побайтного, выводит десятичный дамп парами, позволяя увидеть значения i-number.

```
$ od -c .
0000000   4   ;   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 273   (   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040 252   ;   r   e   c   i   p   e   s  \0  \0  \0  \0  \0  \0  \0
0000060 230   =   j   u   n   k  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100 354   =   x  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000120
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 15852 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

Первые два байта каждой записи каталога являются единственной связью имени файла с его содержимым. Поэтому имя файла и называется *ссылкой*, ведь оно связывает имя в иерархии каталогов с индексным дескриптором и, следовательно, с данными. Одно и то же значение номера индексного дескриптора может встречаться в нескольких каталогах. Команда `rm` в действительности удаляет не индексные дескрипторы, а записи в каталоге, то есть ссылки. И только когда исчезнет последняя ссылка на файл, система удаляет индексный дескриптор, а значит и сам файл.

Если номер индексного дескриптора в элементе каталога равен нулю, это означает, что данная ссылка была удалена, чего нельзя сказать о содержимом файла – возможно, где-то еще остались ссылки на него. Убедимся в том, что значение `i-number` обнуляется при удалении файла:

```
$ rm x
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 00000 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

Следующий созданный в этом каталоге файл займет пустующее место, хотя, скорее всего, значение i-number будет другим.

Ссылка на существующий файл создается командой `ln`:

```
$ ln старый-файл новый-файл
```

Назначение ссылок заключается в том, чтобы дать файлу несколько имен, расположенных, возможно, в разных каталогах. Во многих системах существует ссылка `/bin/e` на файл `/bin/ed`, позволяющая вызывать редактор `e`. Ссылки, указывающие на один файл, имеют одинаковые значения номера индексного дескриптора:

```
$ ln junk linktojunk
$ ls -li
total 3
15768 -rw-rw-rw- 2 you          29 Sep 27 12:07 junk
15768 -rw-rw-rw- 2 you          29 Sep 27 12:07 linktojunk
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$
```

Число между строкой прав доступа и именем владельца показывает количество ссылок на файл. Поскольку ссылка есть лишь указатель на запись в индексном дескрипторе, все ссылки равноправны, нет разницы между первой из них и всеми последующими. (Имейте в виду, что из-за таких повторов `ls` неправильно подсчитывает итоговый объем дискового пространства.)

Если файл был изменен, то изменения будут видны по всем ссылкам, ведь это все тот же файл.

```
$ echo x >junk
$ ls -l
total 3
-rw-rw-rw- 2 you          2 Sep 27 12:37 junk
-rw-rw-rw- 2 you          2 Sep 27 12:37 linktojunk
drwxrwxrwx 4 you         64 Sep 27 09:34 recipes
$ rm linktojunk
$ ls -l
total 2
-rw-rw-rw- 1 you          2 Sep 27 12:37 junk
drwxrwxrwx 4 you         64 Sep 27 09:34 recipes
$
```

После удаления ссылки `linktojunk` счетчик ссылок вернулся к прежнему значению `1`. Как уже говорилось, удаление файла командой `rm` лишь стирает ссылку, а сам файл существует, пока на него есть хотя бы одна ссылка. Конечно, большинству файлов достаточно одной ссылки, но и здесь мы видим, как простая идея дает исключительную гибкость.

Предостережение для торопливых: как только удалена последняя ссылка на файл, его данные теряются. Удаленные файлы попадают в печь, а не в мусорную корзину, и восстановить их из пепла уже не удастся. (Есть, однако, призрачная надежда на восстановление. В большинстве крупных систем периодически выполняется резервное копирование для хранения изменившихся файлов в безопасном месте, обычно на магнитной ленте, откуда их можно восстановить. Для собственной безопасности и спокойствия поинтересуйтесь, как часто выполняется копирование в вашей системе. Если таковое отсутствует, будьте осторожны, сбой диска может обернуться катастрофой.)

Ссылки на файлы удобны, когда требуется одновременный доступ пользователей к файлу, но иногда целесообразно иметь *отдельную* копию – второй файл с той же информацией. Перед редактированием документа полезно сделать копию, чтобы иметь возможность отказаться от изменений и вернуться к оригиналу. Ссылка здесь не поможет, так как все ссылки ведут к уже измененному файлу. Копирование выполняется командой `cp`:

```
$ cp junk copyofjunk
$ ls -li
total 3
15850 -rw-rw-rw- 1 you          2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you          2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you         64 Sep 27 09:34 recipes
$
```

Номера индексных дескрипторов файлов `junk` и `copyofjunk` различны, так как это разные файлы, имеющие в данный момент одинаковое со-

держание. Советуем изменить права доступа к резервной копии, чтобы избежать случайного удаления.

```
$ chmod -w copyofjunk          Убрать разрешение на запись
$ ls -li
total 3
15850 -r--r--r-- 1 you           2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you           2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$ rm copyofjunk
rm: copyofjunk 444 mode n      Нет! Это ценно
$ date >junk
$ ls -li
total 3
15850 -r--r--r-- 1 you           2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you          29 Sep 27 13:16 junk
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$ rm copyofjunk
rm: copyofjunk 444 mode y      Ну, а это не так ценно
$ ls -li
total 2
15768 -rw-rw-rw- 1 you          29 Sep 27 13:16 junk
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$
```

Ни изменение копии, ни ее удаление не скажутся на оригинале. Обратите внимание, что запрет на запись в файл `copyofjunk` вынуждает команду `rm` запрашивать подтверждение на его удаление.

Есть еще одна распространенная команда для работы с файлами: `mv` перемещает (переименовывает) файлы, просто перестраивая ссылки. Синтаксис аналогичен используемому в командах `cp` и `ln`:

```
$ mv junk sameoldjunk
$ ls -li
total 2
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
15768 -rw-rw-rw- 1 you          29 Sep 27 13:16 sameoldjunk
$
```

Значение i-number файла `sameoldjunk` показывает, что это тот же самый файл, что и прежний `junk`, изменилось только его имя, то есть элемент каталога, связанный с индексным дескриптором, имеющим номер 15 768.

Было реализовано перемещение файла внутри одного каталога, но такие перестановки возможны и между каталогами. Часто несколько ссылок с одинаковыми именами, расположенные в разных каталогах, указывают на один файл, например, когда несколько пользователей работают с одной и той же программой или документом. Команда `mv` может перемещать каталог или файл из одного каталога в другой. Для

таких часто повторяющихся действий в командах mv и cp предусмотрен специальный синтаксис:

```
$ mv (или cp) файл1 файл2 ... каталог
```

Здесь один или несколько файлов перемещаются (или копируются) в каталог, определяемый последним аргументом. Копии и ссылки получают те же имена, что и исходные файлы. Например, чтобы усовершенствовать редактор (если возникнет такое желание), можете выполнить команду

```
$ cp /usr/src/cmd/ed.c .
```

и получить собственную копию исходного текста для экспериментов. А если собираетесь поработать над оболочкой, то создайте каталог

```
$ mkdir sh
$ cp /usr/src/cmd/sh/* sh
```

и командой cp скопируйте туда ее исходные файлы (здесь предполагается, что в /usr/src/cmd/sh нет подкаталогов, так как команда cp не сумеет их создать). В некоторых системах ln также может принимать несколько аргументов и последним тоже должно быть имя каталога. А в некоторых системах mv, cp и ln сами являются ссылками на один и тот же файл, который проверяет, по какому имени к нему обращаются, и в зависимости от этого выбирает нужное действие.

Упражнение 2.6. Почему команда ls -l сообщает о 4 ссылках на recipes? Подсказка: используйте

```
$ ls -ld /usr/you
```

Чем полезна эта информация? □

Упражнение 2.7. В чем разница между

```
$ mv junk junk1
```

и

```
$ cp junk junk1
$ rm junk
```

Подсказка: создайте ссылку на junk и проверьте ее значение. □

Упражнение 2.8. Команда cp копирует только файлы первого уровня иерархии и не может копировать подкаталоги. Что она сделает, если один из аргументов окажется каталогом? Удобно ли это? Сравните три способа: параметр команды cp для перехода в подкаталоги, отдельную команду rcp (рекурсивное копирование) и копирование каталогов командой cp по умолчанию. За дополнительной информацией о реализации этой возможности обратитесь к главе 7. Какую пользу другим программам принесла бы возможность прохода по дереву каталогов? □

2.6. Иерархия каталогов

В первой главе неформальное рассмотрение иерархии файловой системы начиналось с каталога /usr/you. Теперь рассмотрим ее более методично, с самого начала, от ее корня.

Корневым каталогом является /:

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

В файле /unix находится ядро UNIX: при запуске системы оно считывается в память и запускается. Точнее говоря, процесс состоит из двух шагов: первым загружается файл /boot, а он, в свою очередь, считывает /unix. Дополнительную информацию о процессе начальной загрузки можно получить из boot(8). Остальные файлы в /, по крайней мере в данном примере, – это каталоги, содержащие отдельные части файловой системы. Читая приведенный ниже краткий обзор иерархии каталогов, загляните в упомянутые там каталоги: поняв, как организована файловая система, вы сможете более эффективно ее использовать. В табл. 2.1 перечислены каталоги, с которыми полезно ознакомиться, правда некоторые из них могут иметь другие имена:

- /bin (binaries – двоичные файлы) – этот каталог уже встречался, в нем находятся основные программы, такие как who и ed;
- /dev (devices – устройства) – рассмотрим этот каталог в следующем разделе;
- /etc (et cetera – разное) – тоже уже встречался. Содержит различные служебные файлы, в частности файл паролей и некоторые программы, например /etc/getty, инициализирующую соединение с терминалом по запросу от /bin/login. Файл /etc/rc – командный, выполняемый после начальной загрузки системы. В /etc/group перечислены пользователи всех групп;
- /lib (library – библиотека) содержит главным образом компоненты компилятора Си, такие как препроцессор /lib/cpp, и стандартную библиотеку /lib/libc.a;
- /tmp (temporaries – временные файлы) – это хранилище файлов, создаваемых только на время выполнения программы.

Таблица 2.1. Представляющие интерес каталоги (см. также hier(7))

Каталог	Содержимое
/	корневой каталог файловой системы
/bin	основные исполняемые программы
/dev	файлы устройств
/etc	прочие системные файлы
/etc/motd	сообщение при регистрации
/etc/passwd	файл паролей
/lib	основные библиотеки и т. п.
/tmp	временные файлы; очищается при перезагрузке системы
/unix	исполняемый образ операционной системы
/usr	файловая система пользователей
/usr/adm	администрирование системы: учетная информация и т. п.
/usr/bin	двоичные файлы пользователя: troff и т. д.
/usr/dict	словарь (words) и файлы поддержки для spell(1)
/usr/games	игровые программы
/usr/include	файлы заголовков для программ на Си, например math.h
/usr/include/sys	файлы заголовков системных программ, например inode.h
/usr/lib	библиотеки Си, Фортрана и т. д.
/usr/man	справочное руководство
/usr/man/man1	страницы первого раздела руководства
/usr/mdec	диагностика оборудования, начальная загрузка и т. п.
/usr/news	сообщения службы news
/usr/pub	общедоступные файлы: см. ascii(7) и eqnchar(7)
/usr/src	исходные тексты утилит и библиотек
/usr/src/cmd	исходные тексты команд из /bin и /usr/bin
/usr/src/lib	исходные тексты библиотечных функций
/usr/spool	рабочие каталоги коммуникационных программ
/usr/spool/lpd	временный каталог построчно печатающего принтера
/usr/spool/mail	почтовые ящики
/usr/spool/uucp	рабочий каталог программ uucp
/usr/sys	исходные тексты ядра операционной системы
/usr/tmp	альтернативный временный каталог (редко используется)
/usr/you	регистрационный каталог пользователя
/usr/you/bin	личные программы пользователя

Например, редактор ed при запуске создает файл с именем вида /tmp/e00512 для хранения копии редактируемого файла и работает с ним, а не с оригиналом. Можно, конечно, создать его и в текущем каталоге, но использование /tmp имеет свои преимущества: маловероятно, но все же возможно, что файл с именем e00512 уже существует, а /tmp очищается автоматически при старте системы – в рабочем каталоге не будут скапливаться ненужные файлы при сбое системы; к тому же, каталог /tmp обычно оптимизирован для быстрого доступа.

Разумеется, может возникнуть проблема, когда несколько программ одновременно пытаются создать файлы в /tmp. Именно поэтому ed дает временным файлам специфические имена: они формируются таким образом, чтобы исключить совпадение с именами временных файлов других программ. В главах 5 и 6 будет рассказано о том, как этого достичь.

Каталог /usr называется «пользовательской файловой системой», хотя может и не иметь ничего общего с действительными пользователями системы. На машине авторов регистрационные каталоги называются /usr/bwk и /usr/rob, но на других машинах пользовательская область может находиться в другом месте, как было рассказано в главе 1. Независимо от того, находятся ли файлы пользователей в /usr или в другом месте, есть ряд вещей, которые там есть наверняка (хотя и эти настройки могут отличаться). Как и в корневом каталоге /, здесь есть каталоги /usr/bin, /usr/lib и /usr/tmp. Они имеют примерно то же назначение, что и их тезки из /, но содержат файлы, менее важные для системы. В частности, nroff обычно располагается в /usr/bin, а не в /bin, а библиотеки компилятора Фортрана находятся в /usr/lib. Конечно, представления о важности меняются от системы к системе. В некоторых, например в дистрибутиве седьмой версии, все программы сосредоточены в /bin, а /usr/bin вообще отсутствует; в других /usr/bin поделен на два каталога в соответствии с частотой использования программ.

Кроме того, в /usr имеются подкаталоги /usr/adm с учетной информацией и /usr/dict, содержащий небольшой словарь (см. spell(1)). Справочное руководство находится в /usr/man – это, например, /usr/man/man1/spell.1. Если в системе присутствуют исходные тексты, то они, скорее всего, находятся в /usr/src.

Советуем потратить некоторое время на исследование файловой системы, в особенности каталога /usr, чтобы понять, как она организована и где при случае можно найти требуемый файл.

2.7. Устройства

В обзоре из предыдущего раздела был пропущен каталог /dev, потому что файлы, находящиеся в нем, сами в некотором роде представляют

обзор всех существующих файлов. Как видно из названия, /dev содержит файлы устройств.

Одна из самых удачных идей в системе UNIX заключается в способе работы с *периферийными устройствами* – дисками, магнитными лентами, принтерами, терминалами и т. п. Вместо того чтобы использовать специальные подпрограммы для работы, например, с магнитными лентами, система обращается к файлу /dev/mt0 (как всегда, имя может отличаться). Внутри ядра обращения к этому файлу транслируются в команды работы с лентой, поэтому программа, выполняющая чтение /dev/mt0, получает данные со смонтированной в данный момент ленты. Например, команда

```
$ cp /dev/mt0 junk
```

копирует содержимое ленты в файл с именем junk. Команду cp не интересуют особенности файла /dev/mt0, для нее это просто последовательность байтов.

Файлы устройств – как обитатели зоопарка, каждый имеет свои особенности, но основные идеи файловой системы применимы ко всем. Вот небольшой фрагмент каталога /dev:

```
$ ls -l /dev
crw--w--w- 1 root      0,   0 Sep 27 23:09 console
crw-r--r-- 1 root      3,   1 Sep 27 14:37 kmem
crw-r--r-- 1 root      3,   0 May  6  1981 mem
brw-rw-rw- 1 root      1,  64 Aug 24 17:41 mt0
crw-rw-rw- 1 root      3,   2 Sep 28 02:03 null
crw-rw-rw- 1 root      4,  64 Sep  9 15:42 rmt0
brw-r----- 1 root      2,   0 Sep  8 08:07 rp00
brw-r----- 1 root      2,   1 Sep 27 23:09 rp01
crw-r----- 1 root     13,   0 Apr 12  1983 rrp00
crw-r----- 1 root     13,   1 Jul 28 15:18 rrp01
crw-rw-rw- 1 root      2,   0 Jul  5 08:04 tty
crw--w--w- 1 you       1,   0 Sep 28 02:38 tty0
crw--w--w- 1 root      1,   1 Sep 27 23:09 tty1
crw--w--w- 1 root      1,   2 Sep 27 17:33 tty2
crw--w--w- 1 root      1,   3 Sep 27 18:48 tty3
$
```

В первую очередь следует обратить внимание на то, что вместо длины файла присутствуют два целых числа, а первый символ в строке прав доступа всегда b или c. Таким способом команда ls показывает, что запись индексного дескриптора относится к устройству, а не к обычному файлу. Индексный дескриптор обычного файла содержит список дисковых блоков, в которых хранится содержимое файла. В индексном дескрипторе файла устройства содержится внутреннее имя устройства, состоящее из обозначения типа символа c для *символьного* или b для *блочного*, и пара чисел, называемых *старшим* и *младшим* номерами устройства. Диски и ленты являются блочными устройствами, а все ос-

тальные – терминалы, принтеры, телефонные линии и т. д. – символьными. Старший номер определяет тип устройства, а младший обозначает конкретный экземпляр устройства. Например, /dev/tty0 и /dev/tty1 – это два порта контроллера терминала и поэтому они имеют одинаковые старшие номера и разные младшие номера.

Файлы дисков обычно получают имена в соответствии с маркой устройства. Так, файлы устройств для диска DEC RP06 называются /dev/rp00 и /dev/rp01. В действительности это одно физическое устройство, логически разделенное на две файловые системы. Если в системе присутствует второй жесткий диск, то его файлы устройства получают имена /dev/rp10 и /dev/rp11. Первая цифра обозначает номер физического диска, а вторая – номер логического раздела.

Может возникнуть вопрос, почему используется не один файл дискового устройства, а несколько. Исторически сложилось так, что файловая система состоит из отдельных подсистем, кроме того, это облегчает ее сопровождение. Доступ к файлам подсистемы осуществляется через каталоги основной системы. Программа /etc/mount показывает соответствие файлов устройств и каталогов:

```
$ /etc/mount
rp01 on /usr
$
```

В данном случае корневая система размещается на устройстве /dev/rp00 (хотя /etc/mount об этом не сообщает), а пользовательская файловая система – файлы каталога /usr и его подкаталоги – на устройстве /dev/rp01.

Корневая файловая система необходима для работы операционной системы. Каталоги /bin, /dev и /etc всегда находятся в корневой файловой системе, так как это единственная файловая система, доступная при загрузке, и в ней хранятся файлы, участвующие в запуске операционной системы, такие как /bin/sh. В процессе начальной загрузки проверяется целостность всех файловых систем (см. icheck(8) или fsck(8)), после чего они присоединяются к корневой системе. Операция присоединения называется монтированием и представляет собой программный аналог установки нового диска в дисковод, для ее выполнения необходимы привилегии суперпользователя. После того как устройство /dev/rp01 смонтировано в каталог /usr, файлы пользовательской файловой системы доступны так же, как если бы они входили в корневую файловую систему.

Обычно пользователю нет необходимости задумываться о том, куда смонтирована каждая из файловых систем, ходя здесь есть два существенных момента. Во-первых, запрещено создание ссылок на файлы, расположенные в других файловых подсистемах, так как они монтируются и размонтируются независимо друг от друга. Например, невозможно в пользовательском каталоге /usr создать ссылку на про-

грамму, находящуюся в /bin, так как эти каталоги принадлежат разным файловым системам:

```
$ ln /bin/mail /usr/you/bin/m
ln: Cross-device link
$
```

Еще одно препятствие состоит в том, что различные файловые системы могут содержать совпадающие номера индексных дескрипторов.

Во-вторых, файловые системы имеют ограничения на размер (количество блоков, доступных для размещения файлов) и количество индексных дескрипторов. Если подсистема заполнена, то невозможно увеличить файл, не освободив предварительно дисковое пространство. Команда df (disc free space – свободное дисковое пространство) показывает количество свободного места в смонтированных файловых системах:

```
$ df
/dev/rp00 1989
/dev/rp01 21257
$
```

Здесь /usr содержит 21 257 свободных блоков. Много это или мало, зависит от того, как используется система, требования к свободному пространству могут очень сильно отличаться. Между прочим, из всех команд df, возможно, имеет наибольшее разнообразие выходных форматов. В других системах результат ее выполнения может выглядеть совершенно по-другому.

Обратимся теперь к более полезным на практике вещам. При регистрации в системе пользователю выделяется терминальная линия, а следовательно, и файл устройства в /dev, через который происходит обмен символами.

Команда tty выводит данные о терминале пользователя:

```
$ who am i
you      tty0    Sep 28 01:02
$ tty
/dev/tty0
$ ls -l /dev/tty0
crw--w--w- 1 you       1, 12 Sep 28 02:40 /dev/tty0
$ date >/dev/tty0
Wed Sep 28 02:40:51 EDT 1983
$
```

Обратите внимание, что устройство принадлежит вам, и только вы имеете разрешение на его чтение. Другими словами, никто не может следить непосредственно за тем, что вводится с клавиатуры. В то же время, вывод на терминал пользователя доступен всем. Этого можно избежать, изменив командой chmod права доступа к устройству, – тогда

другие пользователи не смогут послать вам сообщение командой `write`. Также для этой цели можно выполнить команду `mesg`.

```
$ mesg n              Выключить вывод сообщений
$ ls -l /dev/tty0
crw------- 1 you       1, 12 Sep 28 02:41 /dev/tty0
$ mesg y              Восстановить
$
```

Часто возникает необходимость обратиться к своему терминалу по имени. Чтобы не заниматься его поисками, можно использовать устройство `/dev/tty`, которое является синонимом текущего терминала, независимо от его настоящего имени.

```
$ date >/dev/tty
Wed Sep 28 02:42:23 EDT 1983
$
```

Особенно удобно устройство `/dev/tty` для организации взаимодействия с пользователем, когда стандартный ввод и вывод перенаправлены в файлы. Одной из программ, использующих `/dev/tty`, является `crypt`. Открытый текст считывается из стандартного ввода, зашифрованные данные выводятся в стандартный вывод, а ключ для шифрования `crypt` получает из `/dev/tty`:

```
$ crypt <cleartext >cryptedtext
Enter key:                    Введите ключ для шифрования
$
```

Здесь нет явного упоминания устройства `/dev/tty`, но, тем не менее, оно используется. Ключ не может быть получен из стандартного ввода, поскольку тот предназначен для чтения исходного файла. Поэтому команда `crypt` открывает файл `/dev/tty` и считывает оттуда ключ, отключив автоматический эхо-вывод, поэтому ваш шифрующий ключ не появляется на экране. В главах 5 и 6 будут рассмотрены еще несколько применений `/dev/tty`.

Предположим, пользователю требуется запустить программу, но ее вывод пользователя не интересует. Например, он уже знает сегодняшние новости и не хочет читать их снова.

Вывод команды `news` можно исключить, перенаправив его в устройство `/dev/null`:

```
$ news >/dev/null
$
```

Данные, выводимые в `/dev/null`, пропадают без каких-либо комментариев, а при попытке ввода из `/dev/null` программа немедленно получает признак конца файла, так как операция чтения возвращает ноль байт.

Часто /dev/null служит для того, чтобы выделить из выходных данных программы диагностические сообщения. Например, команда time (time(1)) показывает использование процессора программой. Эта информация направляется в стандартный вывод ошибок, что позволяет хронометрировать программы с большим объемом вывода, направляя его в /dev/null:

```
$ ls -l /usr/dict/words
-r--r--r-- 1 bin    196513 Jan 20  1979 /usr/dict/words
$ time grep e /usr/dict/words >/dev/null

real       13.0
user        9.0
sys         2.7
$ time egrep e /usr/dict/words >/dev/null

real        8.0
user        3.9
sys         2.8
$
```

Значения, выводимые командой time, показывают общее время выполнения, процессорное время, затраченное программой, и процессорное время, затраченное ядром. Команда egrep – это более быстрый вариант grep, который будет рассмотрен в главе 4; при поиске в больших файлах egrep работает примерно вдвое быстрее. Если вывод grep и egrep не перенаправить в файл или в /dev/null, то придется ждать окончания вывода на терминал сотен тысяч символов, чтобы получить нужную информацию.

Упражнение 2.9. Изучите остальные файлы каталога /dev в разделе 4 руководства. В чем разница между /dev/mt0 и /dev/rmt0? Объясните возможные преимущества использования подкаталогов в /dev для дисков, лент и т. п. □

Упражнение 2.10. Ленты, записанные в системах, отличных от UNIX, часто имеют разные размеры блоков, например 800 байт – для десяти 80-символьных перфокарт, но ленточное устройство /dev/mt0 рассчитано на блоки длиной 512 байт. Изучите команду dd (dd(1)), используемую при чтении таких лент. □

Упражнение 2.11. Почему устройство /dev/tty не реализовано как ссылка на терминал, с которого выполнялась регистрация? Что произойдет, если установить ему права доступа rw--w--w-, как у текущего терминала? □

Упражнение 2.12. Как работает write(1)? Подсказка: см. utmp(5). □

Упражнение 2.13. Как определить, что пользователь недавно работал с терминалом? □

История и библиография

Файловой системе посвящена одна из частей исследования «UNIX implementation» (Реализация UNIX) Кена Томпсона (Ken Thompson), изданного в BSTJ (Bell System Technical Journal) в июле 1978 года. Доклад Денниса Ритчи (Dennis Ritchie) «The evolution of the UNIX timesharing system» (Эволюция разделения времени в UNIX), сделанный на симпозиуме по разработке языков и методологии программирования (Symposium on Language Design and Programming Methodology) в Сиднее (Австралия) в сентябре 1979 года, представляет прекрасное описание того, как была спроектирована и реализована файловая система в первой ОС UNIX на PDP-7 и как она доросла до сегодняшнего состояния.

Файловая система UNIX вобрала в себя некоторые идеи из системы MULTICS. Эта система всесторонне рассмотрена в труде «The MULTICS System: An Examination of its Structure» (Система MULTICS: Исследование структуры) Органика (E. I. Organick), изданном MIT Press в 1972 году.

В статье «Password security: a case history» (Безопасность паролей: наглядная иллюстрация) Боба Морриса (Bob Morris) и Кена Томпсона приведено занимательное сравнение механизмов паролей на ряде систем; о нем рассказано во втором томе руководства по UNIX для программиста).

Помещенная в том же томе статья Денниса Ритчи «On the security of UNIX» (О безопасности в UNIX) объясняет, почему безопасность системы больше зависит от грамотного администрирования, чем от использования программ типа `crypt`.

3

Работа с оболочкой

Самой важной программой для большинства пользователей UNIX является оболочка – программа, которая интерпретирует запросы на исполнение команд; взаимодействие с ней занимает больше времени, чем со всеми другими программами, может быть, за исключением любимого текстового редактора. В этой главе и в главе 5 представлен большой объем информации, касающейся возможностей оболочки. Это сделано для того, чтобы показать, что очень многие задачи можно решить, не прилагая титанических усилий и, конечно же, не прибегая к программированию на традиционных языках типа Си.

Мы разделили рассказ об оболочке на две главы. Данная глава описывает более сложные, чем описанные в главе 1, но также широко используемые свойства оболочки: применение метасимволов, заключение в кавычки, создание новых команд, передача им аргументов, работу с переменными оболочки и элементарные управляющие конструкции. Это то, что следует знать для того, чтобы самостоятельно использовать оболочку. Материал же, представленный в главе 5, несколько сложнее – он предназначен для написания серьезных программ оболочки, «пуленепробиваемых» для других пользователей. Информация поделена между главами в некоторой степени случайным образом, так что в конечном счете надо прочесть обе.

3.1. Структура командной строки

Начнем с того, что попробуем понять, что такое команда и как она интерпретируется оболочкой. В этом разделе не очень много новой информации, основная часть посвящена формализации базовых сведений, представленных в главе 1.

Простейшая команда состоит из одного *слова*, обычно этим словом является имя файла, который должен быть выполнен (позже будут рассмотрены и другие виды программ):

```
$ who                    Исполнить файл /bin/who
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$
```

Обычно команда заканчивается символом разделителя строк, но часто для *завершения команды* применяется и точка с запятой ; :

```
$ date;
Wed Sep 28 09:07:15 EDT 1983
$ date; who
Wed Sep 28 09:07:23 EDT 1983
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$
```

Несмотря на то что для завершения команды можно использовать точку с запятой, обычно ничего не происходит до тех пор, пока не нажата клавиша *Return*. Обратите внимание на то, что после нескольких команд, введенных в одной строке, оболочка выводит только одно приглашение на ввод, но в остальном (не считая количества приглашений) ввод

```
$ date; who
```

идентичен вводу двух команд в разных строках. В частности, `who` не выполняется до тех пор, пока не завершится `date`.

Попытаемся передать вывод «`date; who`» в канал:

```
$ date; who | wc
Wed Sep 28 09:08:48 EDT 1983
      2      10      60
$
```

Вероятно, это не совсем то, чего вы ожидали. Дело в том, что только вывод `who` передается `wc`. Соединение команд `who` и `wc` посредством канала образует единую команду, которая называется *конвейером* (*pipeline*), она-то и выполняется после завершения `date`. При разборе командной строки оболочка учитывает то, что приоритет | выше, чем приоритет ; .

Можно сгруппировать команды при помощи скобок:

```
$ (date; who)
Wed Sep 28 09:11:09 EDT 1983
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$ (date; who) | wc
```

```
      3     16     89
$
```

Результат выполнения команд `date` и `who` конкатенируется в один поток, который может быть направлен в канал.

Поток данных по каналу может быть перехвачен и помещен в файл (но не в другой канал) с помощью команды `tee`, которая не входит в оболочку, но может быть полезна для работы с каналами. Одно из ее возможных применений – это сохранение промежуточного вывода в файле:

```
$ (date; who) | tee save | wc
      3     16     89                    Вывод из wc
$ cat save
Wed Sep 28 09:13:22 EDT 1983
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$ wc <save
      3     16     89
$
```

Команда `tee` копирует свой ввод в указанный файл или файлы, а также в свой вывод, так что `wc` получает те же входные данные, которые бы она получила, если бы `tee` не входила в конвейер.

В качестве указателя конца команды также может выступать амперсанд `&`. Он действует так же, как точка с запятой или разделитель строк, за исключением того, что этот символ указывает оболочке, что не следует ждать завершения команды. Амперсанд `&` обычно применяется для выполнения длительной команды «в фоновом режиме», притом, что в это же время вводятся интерактивные команды:

```
$ long-running-command &   Команда с большим временем исполнения
5273                       Идентификатор процесса такой команды
$                          Приглашение на ввод появляется сразу же
```

Можно и более интересно использовать фоновые процессы, если задействовать возможность группировки команд. Команда `sleep` ждет указанное количество секунд, прежде чем завершиться:

```
$ sleep 5
$                                  Приглашение появляется через 5 секунд
$ (sleep 5; date) & date
5278
Wed Sep 28 09:18:20 EDT 19830      Вывод второй команды date
$ Wed Sep 28 09:18:25 EDT 1983     Приглашение, через 5 секунд date
```

Фоновый процесс запускается и сразу же засыпает; тем временем вторая команда `date` печатает текущее время, и оболочка выдает приглашение на ввод новой команды. Через пять секунд `sleep` завершается, и первая команда `date` печатает новое время. Тяжело объяснить течение времени на бумаге, так что лучше попробуйте запустить этот пример на компьютере. (В зависимости от загруженности компьютера и неко-

торых других причин, промежуток времени между двумя событиями может составлять не ровно пять секунд.) Таким образом, не составляет труда запустить команду в будущем; можно, например, реализовать удобное напоминание

```
$ (sleep 300; echo Tea is ready) &  Чай будет готов через 5 минут
5291
$
```

(Если в строке, которая должна быть отражена на терминале, присутствует *ctl*-g, то при выводе раздастся звуковой сигнал.) Использование скобок в этих примерах необходимо, поскольку приоритет амперсанда выше, чем у точки с запятой.

Амперсанд как указатель конца применяется к командам, а так как конвейер – это команда, то для запуска конвейера в фоновом режиме наличие скобок в записи не требуется:

```
$ pr file | lpr &
```

Файл выводится на принтер, но это не заставляет вас ожидать завершения команды. Если заключить конвейер в скобки, эффект будет тем же самым, но придется вводить больше текста с клавиатуры:

```
$ (pr file | lpr) &       Аналогично предыдущему примеру
```

Большинство программ принимают в командной строке *аргументы*, как file (аргумент pr) в последнем примере. Аргументы – это слова, разделенные пробелами и знаками табуляции, обычно представляющие собой имена файлов для обработки, но, будучи строками, могут быть интерпретированы программой любым подходящим для нее способом. Например, pr принимает имена файлов для печати, echo отражает свои аргументы на терминал без интерпретации, а первый аргумент grep задает текстовый шаблон, который должен использоваться при поиске. И, конечно же, многие программы также имеют параметры, которые указываются аргументами, начинающимися со знака минус.

Различные специальные символы, интерпретируемые оболочкой (такие, как <, >, |, ; и &), не являются аргументами запускаемых программ. Вместо этого они контролируют то, как оболочка их запускает. Например,

```
$ echo Hello >junk
```

Здесь сообщается, что оболочка должна запустить echo с единственным аргументом Hello и поместить вывод в файл junk. Строка >junk не является аргументом команды echo; она интерпретируется оболочкой, и echo ее даже не видит. На самом деле эта строка не обязательно должна быть последней в команде:

```
$ >junk echo Hello
```

В этом случае происходит то же самое, но такая запись менее очевидна.

Упражнение 3.1. В чем отличия трех команд, приведенных ниже?

```
$ cat file | pr
$ pr <file
$ pr file
```

(С годами оператор перенаправления < уступил в популярности каналам, похоже, конструкция `cat file |` кажется людям более естественной, чем `<file`.) □

3.2. Метасимволы

Оболочка распознает и ряд других символов как специальные; чаще всего встречается звездочка `*`, сообщающая оболочке, что следует просмотреть каталог в поиске файлов с именами, в которых на месте `*` стоит любая символьная строка. Например,

```
$ echo *
```

это жалкое подобие `ls`. В главе 1 не упоминалось, что при проверке имен файлов на совпадение не рассматриваются имена, начинающиеся с точки, чтобы избежать проблем с именами «.» и «..», которые есть в каждом каталоге. Правило выглядит следующим образом: при поиске имен файлов, соответствующих шаблону, имена, начинающиеся с точки, считаются подходящими, только если в шаблоне точка присутствует явным образом. Как обычно, прояснить ситуацию поможет «здравомыслящая» команда `echo` или даже несколько таких команд:

```
$ ls
.profile
junk
temp
$ echo *
junk temp
$ echo .*
. .. .profile
$
```

Такие символы, как `*`, имеющие специальные свойства, называются *метасимволами*. Их очень много. Полный список метасимволов представлен в табл. 3.1 (но о некоторых из них будет рассказано только в главе 5).

Теперь, когда известно о существовании значительного количества метасимволов, надо найти какой-то способ сказать оболочке: «Оставь этот символ в покое». Самый простой и хороший способ защитить специальные символы от интерпретации оболочкой состоит в том, чтобы заключить их в одинарные кавычки:

```
$ echo '***'
***
$
```

Можно использовать и двойные кавычки "..." , но оболочка обычно заглядывает внутрь таких кавычек в поиске символов $, `...` и \, так что избегайте их, если обработка строки внутри кавычек нежелательна.

Третий способ состоит в том, чтобы ввести обратную косую черту \ перед *каждым* символом, который требуется защитить от обработки оболочкой, например

```
$ echo \*\*\*
```

Несмотря на то что *** не очень-то напоминает английское слово, в терминах оболочки это *слово* (любая отдельная строка, которую оболочка воспринимает как единое целое, включая пробелы, если они заключены в кавычки).

Кавычки одного типа защищают кавычки другого типа:

```
$ echo "Don't do that!"
Don't do that!
$
```

и не обязательно заключать в кавычки весь аргумент:

```
$ echo x'*'y
x*y
$ echo '*'A'?'
*A?
$
```

Таблица 3.1. Метасимволы оболочки

Метасимвол	Пояснение
>	*prog*>*file* направляет стандартный вывод в файл
>>	*prog*>>*file* добавляет стандартный вывод в конец файла
<	*prog*<*file* берет стандартный ввод из файла
|	p_1|p_2 направляет стандартный вывод p_1 на стандартный ввод p_2
<<*str*	*встроенный документ* (*here document*) – стандартный поток ввода начинается в строке непосредственно после символа, признак окончания – *str*
*	соответствует любой строке из нуля и более символов, входящей в имя файла
?	соответствует любому отдельному символу в имени файла
[*ccc*]	соответствует любому отдельному символу из *ccc* в имени файла, разрешено использование диапазонов, например 0-9 или a-z
;	указатель конца команды, p_1;p_2 выполняет p_1, затем p_2
&	аналогично ; , но без ожидания завершения p_1
`...`	выполняет команду (команды) в ...; вывод которых заменяет `...`

Метасимвол	Пояснение
`(...)`	выполняет команду (команды) в `...` в дочерней оболочке
`{...}`	выполняет команду (команды) в `...` в текущей оболочке (редко используется)
`$1`, `$2` *и т. д.*	`$0`…`$9` заменяются аргументами, переданными командному файлу
`$`*var*	значение переменной оболочки *var*
`${`*var*`}`	значение *var*, предотвращает перемешивание при объединении с текстом (см. также табл. 5.3)
`\`	`\`*c* воспринимает буквально символ *c*, `\`*разделитель-строк* игнорируется
`` `...` ``	воспринимает `...` буквально
`"..."`	воспринимает `...` буквально, после того как интерпретированы `$`, `` `...` `` и `\`
`#`	если слово начинается с `#`, то оставшаяся часть строки – это комментарий (нет в седьмой версии)
var=*value*	присваивает значение переменной *var*
p_1 `&&` p_2	запускает p_1; в случае успеха запускает p_2
p_1 `||` p_2	запускает p_1; в случае неудачи запускает p_2

В последнем примере `echo` видит единый аргумент без кавычек, поскольку после того, как они сделали свою работу, кавычки отбрасываются.

Строки, заключенные в кавычки, могут содержать разделители строк:

```
$ echo 'hello
> world'
hello
world
$
```

Строка «`> `» – это *второе приглашение на ввод*, которое оболочка выводит в тех случаях, когда она ожидает продолжения ввода для завершения команды. В примере, приведенном выше, кавычка, открытая в первой строке, должна быть закрыта. Второе приглашение хранится в переменной оболочки `PS2` и может быть изменено по желанию пользователя.

Во всех этих примерах заключение метасимвола в кавычки предотвращает его интерпретирование оболочкой. Команда

```
$ echo x*y
```

выводит все имена, начинающиеся с `x` и заканчивающиеся `y`. Как обычно, `echo` ничего не знает о файлах и метасимволах; интерпретацией символа * (если он есть), занимается оболочка.

Что произойдет, если не будут найдены файлы, соответствующие шаблону? Оболочка вместо того, чтобы жаловаться (так было в более ранних версиях), передает строку так, как будто она была заключена в кавычки. Вообще-то неразумно зависеть от такой линии поведения, хотя с помощью этого эффекта можно попробовать узнать о наличии файла, соответствующего шаблону.

```
$ ls x*y
x*y not found       Сообщение от ls: таких файлов нет
$ >xyzzy            Создать xyzzy
$ ls x*y
xyzzy               Файл xyzzy соответствует x*y
$ ls 'x*y'
x*y not found       ls не интерпретирует *
$
```

Обратная косая черта в конце строки означает, что строка будет продолжена; таким образом можно передать оболочке очень длинную строку.

```
$ echo abc\
> def\
> ghi
abcdefghi
$
```

Обратите внимание на то, что если разделителю строк предшествует обратная косая черта, то он отбрасывается, а если же он заключен в кавычки, то сохраняется.

Метасимвол # практически повсеместно используется для комментариев оболочки; если слово оболочки начинается с #, то оставшаяся часть строки игнорируется.

```
$ echo hello # there
hello
$ echo hello#there
hello#there
$
```

Символ # не входил в оригинальную седьмую версию, но он получил очень широкое распространение, и поэтому будет использоваться и в этой книге.

Упражнение 3.2. Объясните, каким будет вывод

```
$ ls .*
```

□

Экскурс в команду echo

Команда echo выводит заключительный символ разделителя строк, хотя в явном виде такое требование не существует. Конструкция коман-

ды была бы более разумной и «чистой», если бы echo выводила только то, что требуется. Тогда было бы проще выдавать приглашения на ввод:

```
$ pure-echo Enter a command:
Enter a command:$          Нет замыкающего разделителя строк
```

Но в таком варианте есть и недостаток – очень часто оказывается, что переход на новую строку после завершения команды желателен, а его отсутствие вызывает необходимость дополнительного ввода:

```
$ pure-echo 'Hello!
> '
Hello!
$
```

По умолчанию команда должна выполнять те функции, которые наиболее часто используются, поэтому настоящая команда echo автоматически добавляет заключительный разделитель строк.

Но что делать пользователю, которому не нужен этот разделитель? В седьмой версии есть специальный параметр -n, который убирает последний разделитель строк:

```
$ echo -n Enter a command:
Enter a command:$          Приглашение на ввод на той же строке
$ echo -
-                          Только -n имеет специальное значение
$
```

Существует и трюк, позволяющий вывести -n, а следом – разделитель строк:

```
$ echo -n '-n
> '
-n
$
```

Выглядит безобразно, но работает, к тому же такая ситуация очень редко встречается.

System V предлагает команде echo другой подход для уничтожения замыкающего разделителя строк – интерпретировать Си-подобные escape-последовательности, такие как \b для символа возврата на одну позицию (backspace) и \c (отсутствует в Си) для уничтожения последнего разделителя строк:

```
$ echo 'Enter a command:\c'          Версия System V
Enter a command:$
```

Хотя применение этого механизма не порождает путаницы с выводом на терминал знака минус, зато есть другие проблемы. Дело в том, что команда echo часто применяется для диагностики, а обратная косая

черта интерпретируется таким большим количеством команд, что если этим займется и echo, все только еще больше запутается.

Итак, обе конструкции имеют сильные и слабые стороны. В книге будет использоваться версия (-n) для седьмой версии, так что если ваша программа echo подчиняется другим условиям, некоторые программы могут потребовать незначительной доработки.

Есть и еще один философский вопрос: что команда echo должна делать, если у нее нет аргументов, точнее, должна ли она печатать пробел или не должна вообще ничего делать? Все современные реализации echo (известные авторам) печатают пробел, а вот более ранние версии не делали этого, когда-то по этому поводу даже велись жаркие споры. Даг Мак-Илрой (Doug McIlroy) привнес в обсуждение этой темы немного мистицизма.

UNIX и Эхо

Жила-была в королевстве Нью-Джерси девушка UNIX. Издалека съезжались ученые, чтобы посмотреть на нее, так она была восхитительна. Ослепленные ее чистотой, все пытались завоевать ее сердце; одних привлекали ее грация и непорочность, других – изысканные манеры, третьих – ловкость в выполнении трудных, обременительных заданий, которые редко выполнялись и в более благоприятных условиях. У девушки было такое доброе сердце и сговорчивый нрав, что UNIX приняла всех своих поклонников, кроме самых богатых. Вскоре появилось множество их потомков, они жили и процветали во всех уголках земли.

Сама природа улыбалась UNIX и отвечала ей охотнее, чем другим смертным. Простой народ, который мало что понимал в высших материях, наслаждался ее эхо (*echo*), оно было таким точным и кристально чистым, что люди с трудом верили в то, что девушке отвечают те же самые леса и горы, которые так искажают их собственные крики. А уступчивая UNIX получала совершенное эхо в ответ на все свои слова.

Когда один нетерпеливый обожатель сказал UNIX: «Я хочу услышать эхо молчания», девушка послушно разомкнула уста, и эхо ответило ей.

Молодой человек сказал: «Зачем ты открываешь рот? Никогда больше не делай этого, если подразумевается, что ты отразишь молчание!» И UNIX послушалась.

Другой юноша, очень впечатлительный, просил: «Мне бы так хотелось, чтобы исполнение было совершенным, даже когда ты вызываешь эхо в ответ на молчание, а совершенное эхо не может появиться из закрытого рта». UNIX не хотела никого обижать, поэтому она согласилась говорить разные «ничего» для чувстви-

тельного и нетерпеливого молодых людей. Она назвала «ничего» для чувствительного юноши «\n».

Но теперь, когда девушка говорила «\n», она уже на самом деле не молчала, так что ей приходилось открывать рот дважды: сначала – чтобы сказать «\n», а потом, чтобы сказать «ничего». Это не понравилось впечатлительному юноше, который сказал «Твое \n звучит для меня, как настоящее «ничего», но вот второе все портит. Мне бы хотелось, чтобы одного из них не было». Тогда UNIX, которая не могла никого оставить обиженным, согласилась отменить некоторые эхо, она назвала эту операцию «\c». Теперь чувствительный молодой человек мог услышать совершенное отражение молчания, попросив «\n» и «\c» вместе. Но, говорят, он умер от излишеств в условных обозначениях, так и не услышав ни одного совершенного эха.

Упражнение 3.3. Спрогнозируйте, что будет делать каждая из перечисленных ниже команд `grep`, затем проверьте свои предположения:

```
grep \$          grep \\
grep \\$         grep \\\\
grep \\\$        grep "\$"
grep '\$'        grep '"$'
grep '\"$'       grep "$"
```

Файл, содержащий сами эти команды, представляет собой хорошую совокупность тестовых данных, которая может пригодиться для дальнейших экспериментов. □

Упражнение 3.4. Как добиться, чтобы команда `grep` осуществляла поиск по шаблону, начинающемуся с «-»? Почему не помогает заключение аргумента в кавычки? Подсказка: исследуйте параметр `-e`. □

Упражнение 3.5. Рассмотрим

```
$ echo */*
```

Выводит ли такая команда все имена из всех каталогов? В каком порядке выводятся имена? □

Упражнение 3.6 (сложный вопрос). Каким образом ввести / в имя файла (так, чтобы этот символ не воспринимался как разделяющий компоненты имени)? □

Упражнение 3.7. Что произойдет в результате

```
$ cat x y >y
```

и

```
$ cat x >>x
```

Подумайте, прежде чем бросаться к компьютеру. □

Упражнение 3.8. Почему, если пользователь вводит

```
$ rm *
```

команда `rm` не может предупредить его о том, что он пытается удалить все свои файлы? □

3.3. Создание новых команд

Настало время обратиться к теме, заявленной в главе 1, и поговорить о том, как из старых команд создавать новые.

Если какую-то последовательность команд приходится выполнять достаточно часто, то было бы удобно превратить ее в «новую» команду, имеющую собственное имя, так чтобы ее можно было использовать в дальнейшем как обычную команду. Чтобы быть конкретными, предположим, что вам часто приходится подсчитывать количество пользователей с помощью конвейера

```
$ who | wc -l
```

Такая возможность описывалась в главе 1, теперь же хотелось бы создать новую программу `nu`, которая бы это делала.

Первый шаг заключается в создании обычного файла, содержащего «`who | wc -l`». Можно сделать это в любимом редакторе или же проявить творческий подход:

```
$ echo 'who | wc -l' >nu
```

(Если бы кавычек не было, то что было бы записано в `nu`?)

Как уже упоминалось в главе 1, оболочка – это программа, подобная редактору или командам `who` и `wc`; она называется `sh`. А раз это программа, можно запустить ее и перенаправить *ее* ввод. Итак, запустим оболочку, задав для нее ввод из файла `nu`, а не с терминала:

```
$ who
you      tty2    Sep 28 07:51
rhh      tty4    Sep 28 10:02
moh      tty5    Sep 28 09:38
ava      tty6    Sep 28 10:17
$ cat nu
who | wc -l
$ sh <nu
      4
$
```

Вывод выглядит так же, как если бы команда `who | wc -l` была введена с терминала.

Опять-таки, как и многие другие программы, оболочка забирает входные данные из файла, если он указан как аргумент; так что можно было написать

```
$ sh nu
```

и результат не изменился бы. Но вводить каждый раз «sh» неудобно, к тому же возникает отличие программ, написанных (например) на Си, от программ, созданных за счет соединения других программ с оболочкой.[1] Поэтому если файл является исполняемым и содержит текст, то оболочка считает, что это файл с командами оболочки. Такой файл называется *командным файлом оболочки*. Итак, все, что требуется – это сделать nu исполняемой один-единственный раз:

```
$ chmod +x nu
```

и тогда впоследствии ее можно будет вызывать следующим образом:

```
$ nu
```

С этого момента пользователи nu уже не смогут догадаться (просто запустив программу), что она реализована таким простым способом.

Как же оболочка на самом деле запускает nu? Создается новый процесс оболочки, точно так же, как если бы было введено

```
$ sh nu
```

Эта дочерняя оболочка называется *подоболочкой* и представляет собой процесс оболочки, запущенный текущей оболочкой. Стандартный ввод sh nu поступает с терминала, поэтому sh nu – это не то же самое, что sh <nu.

В своем теперешнем состоянии команда nu работает, только если она находится в текущем каталоге (если, конечно, текущий каталог присутствует в переменной PATH, что и будет предполагаться начиная с этого момента). Чтобы иметь возможность использовать nu всегда, вне зависимости от того, в каком каталоге пользователь работает, переместите ее в личный каталог /bin и добавьте /usr/you/bin в путь поиска:

```
$ pwd
/usr/you
$ mkdir bin              Создать bin, если он не существовал ранее
$ echo $PATH             Проверить PATH, чтобы знать наверняка
:/usr/you/bin:/bin:/usr/bin        Должно быть нечто похожее
$ mv nu bin              Установить nu
$ ls nu
nu not found             Ее действительно нет в текущем каталоге
```

[1] Однако такое разграничение существует в большинстве других операционных систем.

```
$ nu
      4                    Но оболочка находит ее
$
```

Естественно, переменная PATH должна быть правильно задана в файле .profile, тогда не надо будет повторять установки при каждом входе в систему.

Есть и другие простые команды, которые можно создать таким же способом, чтобы настроить окружение по своему вкусу.

Вот те из них, которые могут пригодиться:

- cs выводит собственную последовательность загадочных символов для очистки экрана терминала (чаще всего реализуется посредством вывода 24 символов новой строки);
- what запускает who и ps -a, выводя сообщение о тех, кто находится в системе, и о том, чем они занимаются;
- where печатает идентификационное имя системы UNIX; это удобно, если регулярно используется несколько систем. (Установка PS1 приводит к такому же результату.)

Упражнение 3.9. Просмотрите /bin и /usr/bin, чтобы узнать, сколько команд являются в настоящее время командными файлами. Можно ли сделать это при помощи одной команды? Подсказка: file(1). Насколько верны догадки, основанные на оценке длины файла? □

3.4. Аргументы и параметры команд

Большинство программ оболочки интерпретируют аргументы, поэтому когда команда запускается, можно указать, например, имена файлов и параметры (хотя некоторые команды, например nu, успешно работают и без аргументов).

Предположим, что требуется создать программу cx, которая изменяла бы код прав доступа к файлу на исполняемый, таким образом,

```
$ cx nu
```

представляет собой краткую запись

```
$ chmod +x nu
```

Вам уже известно почти все, что для этого необходимо. Нужен файл с именем cx, который содержал бы

```
chmod +x имя-файла
```

Единственное, о чем еще не говорилось, так это о том, каким образом сообщить команде cx имя файла, если при каждом запуске cx оно будет разным.

Когда оболочка выполняет командный файл, каждое вхождение $1 заменяется на первый аргумент, каждое вхождение $2 – на второй аргумент, и так до $9. Поэтому, если файл cx содержит

```
chmod +x $1
```

то когда запускается команда

```
$ cx nu
```

подоболочка заменяет $1 на первый аргумент – nu.

Давайте посмотрим на всю последовательность операций:

```
$ echo 'chmod +x $1' >cx          Сначала создать cx
$ sh cx cx                        Сделать файл cx исполняемым
$ echo echo Hi, there! >hello     Создать тестовую программу
$ hello                           Опробовать ее
hello: cannot execute
$ cx hello                        Сделать ее исполняемой
$ hello                           Попробовать еще раз
Hi, there!                        Работает
$ mv cx /usr/you/bin              Установить cx
$ rm hello                        Очистить
$
```

Обратите внимание, что было введено

```
$ sh cx cx
```

В точности то же самое автоматически бы сделала оболочка, если бы команда cx уже была исполняемой и было бы введено

```
$ cx cx
```

Что делать, если необходимо наличие нескольких аргументов, например, чтобы программа типа cx могла обрабатывать сразу несколько файлов? Первый, «топорный» способ, состоит в том, чтобы задать для программы оболочки девять аргументов:

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

(Работает только для 9 аргументов, потому что строка $10 воспринимается как «первый аргумент, $1, за которым следует 0»!). Если пользователь такого командного файла задает меньше, чем девять аргументов, место отсутствующих занимают пустые строки; в результате подоболочка передает chmod только те аргументы, которые действительно были определены. То есть этот способ работает, но имеет явные недостатки, к тому же он не подходит, если задано более девяти аргументов.

Предвосхищая возможный поворот событий, оболочка предоставляет шаблон $*, который означает «все аргументы». Тогда правильный способ определения cx выглядит так:

```
chmod +x $*
```

Сколько бы аргументов ни было задано, все работает.

Имея в своем репертуаре `$*`, можно создать несколько удобных командных файлов, например `lc` или `m`:

```
$ cd /usr/you/bin
$ cat lc
# lc: сосчитать количество строк в файлах
wc -l $*
$ cat m
# m: краткое обозначение для mail
mail $*
$
```

Обе программы осмысленно работают и без аргументов. Если аргументы не заданы, то `$*` будет пустым, и вообще никакие аргументы не будут переданы `wc` и `mail`. Но и без аргументов, и с ними команда вызывается правильно:

```
$ lc /usr/you/bin/*
      1 /usr/you/bin/cx
      2 /usr/you/bin/lc
      2 /usr/you/bin/m
      1 /usr/you/bin/nu
      2 /usr/you/bin/what
      1 /usr/you/bin/where
      9 total
$ ls /usr/you/bin | lc
      6
$
```

Эти команды и команды из других примеров данной главы относятся к разряду *личных* программ. Личными называются программы, которые пишутся пользователем для себя и помещаются в личный каталог `/bin`. К ним не предоставляется общий доступ, потому что они слишком сильно зависят от вкуса конкретного пользователя. В главе 5 будут представлены основы написания программ оболочки, которые подходят для общего использования.

Аргументами командного файла не обязательно должны быть имена файлов. Разберем пример просмотра персонального личного телефонного каталога. Если существует файл `/usr/you/lib/phone-book`, который содержит что-то вроде:

```
dial-a-joke  212-976-3838
dial-a-prayer  212-246-4200
dial santa  212-976-3636
dow jones report  212-976-4141
```

то просмотреть его можно командой `grep`. (Неплохим местом для хранения персональных баз данных является каталог `/lib`.) Формат информации не имеет значения для `grep`, поэтому можно искать имена,

адреса, почтовые индексы, то есть все, что угодно. Давайте создадим программу «справочная линия» и назовем ее 411 в честь номера телефонной справочной линии нашего района:

```
$ echo 'grep $* /usr/you/lib/phone-book' >411
$ cx 411
$ 411 joke
dial-a-joke  212-976-3838
$ 411 dial
dial-a-joke  212-976-3838
dial-a-prayer  212-246-4200
dial santa  212-976-3636
$ 411 'dow jones'
grep: can't open jones          Что-то не так
$
```

Последний пример приведен для того, чтобы показать возможные трудности: несмотря на то, что dow jones передан 411 как единый аргумент, он содержит пробел и уже не заключен в кавычки, поэтому подоболочка, которая интерпретирует команду 411, преобразует его в два аргумента для grep: получается, как будто было введено

```
$ grep dow jones /usr/you/lib/phone-book
```

а это, естественно, неправильно.

Есть средство, способное помочь в такой ситуации, оно использует особенность обработки оболочкой двойных кавычек. Все, что заключено в '...', остается нетронутым, а вот внутрь "..." оболочка заглядывает в поиске $, \ и `...`. Поэтому если изменить 411 таким образом:

```
grep "$*" /usr/you/lib/phone-book
```

то строка $* будет заменена аргументами, но команде grep будет передаваться как один аргумент, даже если в ней будут содержаться пробелы.

```
$ 411 dow jones
dow jones report  212-976-4141
$
```

Кстати, заодно можно сделать grep (а значит, и 411) не зависящей от регистра, для этого укажем параметр -y:

```
$ grep -y шаблон ...
```

Если указан параметр -y, то строчные буквы из шаблона будут сопоставляться с прописными буквами из входного потока. (Этот параметр включен в grep из седьмой версии , но отсутствует в некоторых других системах.)

В обсуждении аргументов команд есть пять пунктов, которые будут отложены до главы 5, но все же один из них заслуживает упоминания

прямо сейчас. Аргумент $0 является именем исполняемой программы: в cx $0 – это cx. Нестандартное использование $0 можно встретить в реализации программ 2, 3, 4, ..., которые печатают свой вывод в несколько колонок:

```
$ who | 2
drh      tty0    Sep 28 21:23       cvw      tty5    Sep 28 21:09
dmr      tty6    Sep 28 22:10       scj      tty7    Sep 28 22:11
you      tty9    Sep 28 23:00       jlb      ttyb    Sep 28 19:58
$
```

Реализации 2, 3, ... идентичны, на самом деле все они представляют собой ссылки на один и тот же файл:

```
$ ln 2 3; ln 2 4; ln 2 5; ln 2 6
$ ls -li [1-9]
16722 -rwxrwxrwx 5 you        51 Sep 28 23:21 2
16722 -rwxrwxrwx 5 you        51 Sep 28 23:21 3
16722 -rwxrwxrwx 5 you        51 Sep 28 23:21 4
16722 -rwxrwxrwx 5 you        51 Sep 28 23:21 5
16722 -rwxrwxrwx 5 you        51 Sep 28 23:21 6

$ ls /usr/you/bin | 5
2             3             4             411           5
6             cx            lc            m             nu
what          where
$ cat 5
# 2, 3, ...:  печать в n колонок
pr -$0 -t -l1 $*
$
```

Параметр -t подавляет вывод заголовка наверху страницы, а параметр -l*n* устанавливает длину страницы равной *n* строкам. Имя программы становится аргументом, задающим количество колонок для pr, так что вывод печатается построчно в количество колонок, определенное аргументом $0.

3.5. Вывод программы в качестве аргументов

Теперь перейдем от аргументов команды внутри командного файла к формированию аргументов. Конечно же, самый распространенный способ порождения аргументов состоит в расширении имени файла с помощью метасимволов типа * (не считая явного ввода аргументов), но есть и другой хороший способ – запуск программы. Вывод любой программы может быть помещен в командную строку, для этого команда должна быть заключена в обратные кавычки `...`:

```
$ echo At the tone the time will be `date`.
At the tone the time will be Thu Sep 29 00:02:15 EDT 1983.
$
```

Небольшое изменение, призванное показать, что обратные кавычки `...` интерпретируются оболочкой внутри двойных кавычек "...":

```
$ echo "At the tone
> the time will be `date`."
At the tone
the time will be Thu Sep 29 00:03:07 EDT 1983.
$
```

Приведем другой пример. Предположим, требуется отправить почту целому списку пользователей, регистрационные имена которых находятся в файле mailinglist. Можно, конечно, решить задачу, преобразовав mailinglist в соответствующую команду mail и предъявив ее оболочке (не очень красивый способ), но гораздо проще сказать:

```
$ mail `cat mailinglist` <letter
```

Выполняется cat, создавая список имен пользователей, которые и становятся аргументами mail. (Интерпретируя вывод, заключенный в обратные кавычки, как аргументы, оболочка воспринимает разделители строк как разделители слов, а не как символы завершения командной строки; эта тема будет подробно обсуждаться в главе 5.) Обратные кавычки достаточно просты и удобны в использовании, так что команда mail не нуждается в специальном параметре для списка рассылки.

Несколько другой подход заключается в преобразовании файла mailinglist из собственно списка имен в программу, которая выводит список имен.

```
$ cat mailinglist          Новая версия
echo don whr ejs mb
$ cx mailinglist
$ mailinglist
don whr ejs mb
$
```

Теперь рассылка писем пользователям по списку выглядит следующим образом:

```
$ mail `mailinglist` <letter
```

Если добавить еще одну команду, появится возможность изменять список пользователей в интерактивном режиме. Такая программа называется pick:

```
$ pick аргументы ...
```

Аргументы предъявляются один за другим, и после каждого программа ожидает ответа. Выводом pick являются те аргументы, в ответ на которые поступило y (yes); любой другой ответ означает, что аргумент будет отброшен. Например,

```
$ pr `pick *.c` | lpr
```

выводит одно за другим все имена файлов, оканчивающиеся на .c; те из них, которые будут выбраны (ответ «y»), печатаются программами pr и lpr. (Команда pick не входит в седьмую версию, но она настолько проста и удобна, что ее версии были включены в примеры глав 5 и 6.)

Предположим, что речь идет о второй версии mailinglist. Тогда посылка писем адресатам don и mb выглядит так:

```
$ mail `pick \`mailinglist\`` <letter
don? y
whr?
ejs?
mb? y
$
```

Обратите внимание на то, что в записи присутствуют вложенные обратные кавычки; наличие обратной косой черты предотвращает интерпретацию внутренних `...` при разборе внешних.

Упражнение 3.10. Что произойдет, если в

```
$ echo `echo \`date\``
```

убрать все символы обратной косой черты? □

Упражнение 3.11. Введите

```
$ `date`
```

и поясните результат. □

Упражнение 3.12.

```
$ grep -l шаблон имена-файлов
```

перечисляет имена файлов, которые соответствуют *шаблону*, но больше ничего не выводит. Попробуйте запустить несколько вариаций на тему

```
$ команда `grep -l шаблон имена-файлов`
```

□

3.6. Переменные оболочки

У оболочки есть такие же переменные, как и в большинстве языков программирования (в терминах оболочки их также называют *параметрами*). Такие строки, как $1, являются *позиционными параметрами* – это переменные, которые содержат аргументы, передаваемые в командный файл. Цифра указывает позицию в командной строке. Уже упоминались такие переменные оболочки, как PATH (список каталогов для поиска команд), HOME (регистрационный каталог) и другие. Отличие от переменных обычного языка заключается в том, что пере-

менная аргумента не может быть изменена. Хотя PATH – это переменная со значением $PATH, но не существует переменной 1, значение которой равнялось бы $1. Так что $1 – это просто компактная запись первого аргумента и ничего больше.

Переменные оболочки, не являющиеся позиционными параметрами, можно создавать, обращаться к ним и изменять их. Например,

```
$ PATH=:/bin:/usr/bin
```

присваивание, которое изменяет путь поиска. По обе стороны от знака равенства не должно быть пробелов, а присваиваемое значение должно быть единым словом (то есть необходимо заключить его в кавычки, если слово содержит метасимволы, которые не должны интерпретироваться оболочкой). Для того чтобы получить значение переменной, надо поставить знак доллара перед ее именем:

```
$ PATH=$PATH:/usr/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$ PATH=:/usr/you/bin:/bin:/usr/bin       Восстановление
$
```

Не все переменные оболочки являются предопределенными. Можно создавать новые переменные, присваивая им значения; по традиции переменные со специальным смыслом записываются прописными буквами, а обычные имена – строчными (в нижнем регистре). Часто переменные используются для хранения длинных строк, например путей:

```
$ pwd
/usr/you/bin
$ dir=`pwd`              Запомнить текущий каталог
$ cd /usr/mary/bin       Перейти в другое место
$ ln $dir/cx .           Использовать переменную в имени файла
$ ...                    Поработать некоторое время
$ cd $dir                Вернуться обратно
$ pwd
/usr/you/bin
$
```

Встроенная в оболочку команда set показывает значения всех определенных переменных. Чтобы увидеть одну-две переменные, разумнее использовать echo.

```
$ set
HOME=/usr/you
IFS=

PATH=:/usr/you/bin:/bin:/usr/bin
PS1=$
PS2=>
dir=/usr/you/bin
```

```
$ echo $dir
/usr/you/bin
$
```

Значение переменной сопоставлено оболочке, которая создала ее, и не передается автоматически потомку оболочки.

```
$ x=Hello            Создать x
$ sh                 Новая оболочка
$ echo $x
                     Только разделитель строк: x не определена в подоболочке
$ ctl-d              Покинуть эту оболочку
$                    Обратно в исходную оболочку
$ echo $x
Hello                x определена
$
```

Командный файл не может изменить значение переменной, потому что он запускается подоболочкой:

```
$ echo 'x="Good Bye"'        Создать командный файл из двух строк...
> echo $x' >setx                    ... для задания и печати x
$ cat setx
x="Good Bye"
echo $x
$ echo $x
Hello                        В исходной оболочке x — это Hello
$ sh setx
Good Bye                     В подоболочке x — это Good Bye...
$ echo $x
Hello                        ...но все еще Hello в этой оболочке
$
```

Однако бывают случаи, когда использование командного файла для изменения переменной оболочки возможно. Очевидным примером является файл для добавления нового каталога в PATH. В оболочке есть команда . (точка), которая исполняет команды в файле в текущей оболочке, а не в подоболочке. Изначально это было придумано для того, чтобы пользователю было удобно перезапускать свой .profile и не приходилось бы для этого выходить из системы и заново входить в нее, теперь же эта команда применяется и в других целях:

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games            Добавить /usr/games в PATH
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$
```

Поиск файла для команды . осуществляется при помощи механизма PATH, поэтому можно поместить его в личный каталог /bin.

Исполнение файла командой . только внешне напоминает запуск командного файла. В обычном смысле слова файл не «исполняется». Вместо этого команды файла интерпретируются точно таким образом, как если бы они вводились интерактивно – стандартный ввод оболочки временно перенаправлен и поступает из файла. Так как файл не исполняется, а только читается, ему не нужны права на исполнение. Еще одно отличие заключается в том, что файл не получает аргументов командной строки; вместо этого есть только $1 и $2, а остальное пусто. Было бы удобно, если бы передавались аргументы, но этого не происходит.

Другой способ установки значения переменной в подоболочке заключается в явном присваивании в командной строке *перед* самой командой:

```
$ echo 'echo $x' >echox
$ cx echox
$ echo $x
Hello                    Как и раньше
$ echox
                         В подоболочке x не установлена
$ x=Hi echox
Hi                       Значение x передано в подоболочку
$
```

(Первоначально присваивания, занимающие любое место в командной строке, передавались команде, но это мешало dd(1).)

Механизм . следует использовать для того, чтобы перманентно изменить значение переменной, а присваивания в командной строке – для временных изменений. Для примера еще раз поищем команды в каталоге /usr/games при условии, что каталог не содержится в PATH:

```
$ ls /usr/games | grep fort
fortune                        Команда, выводящая афоризмы
$ fortune
fortune: not found
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin   /usr/games не содержится в PATH
$ PATH=/usr/games fortune
Ring the bell; close the book; quench the candle.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin    PATH не изменена
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games          games команда все еще здесь
$ . games
$ fortune
Premature optimization is the root of all evil - Knuth
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games   На этот раз PATH изменена
$
```

Оба эти механизма можно использовать в одном командном файле. Немного измененная команда games может применяться для запуска игры без изменения PATH или может перманентно изменить PATH так, чтобы переменная содержала /usr/games:

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games $*                    Обратите внимание на $*
$ cx /usr/you/bin/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                 Не содержит /usr/games
$ games fortune
I'd give my right arm to be ambidextrous.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                 Все еще не содержит
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games      Теперь содержит
$ fortune
He who hesitates is sometimes saved.
$
```

Первое обращение к games запускает командный файл в подоболочке, в которой PATH была временно изменена с тем, чтобы включить в себя /usr/games. Вместо этого второй пример интерпретирует файл в текущей оболочке, с пустой строкой $*, так что в строке нет команды, и PATH изменяется. В применении этих двух способов для games есть определенное трюкачество, в результате же появилось естественное и удобное средство.

Значение переменной можно сделать доступным в подоболочках с помощью команды оболочки export. (Можете подумать над тем, почему нет способа экспортировать значение переменной из подоболочки в родительскую оболочку.) Приведем уже рассмотренный выше пример, на этот раз используем экспортированную переменную:

```
$ x=Hello
$ export x
$ sh                        Новая оболочка
$ echo $x
Hello                       x известна подоболочке
$ x='Good Bye'              Изменить ее значение
$ echo $x
Good Bye
$ ctl-d                     Покинуть эту оболочку
$                           Вернуться в исходную оболочку
$ echo $x
Hello                        x все еще Hello
$
```

Семантика команды export не очевидна, но для повседневной работы достаточно эвристического правила: не экспортируйте временные пе-

ременные, экспортируйте переменные, которые хотите задать во всех оболочках и подоболочках (в том числе, например, оболочки, запускаемые командой ! редактора ed). Поэтому следует экспортировать специальные переменные оболочки, такие как PATH и HOME.

Упражнение 3.13. Почему мы всегда включаем текущий каталог в переменную PATH? Куда он должен быть помещен? □

3.7. Снова о перенаправлении ввода-вывода

Стандартный вывод ошибок организован так, чтобы сообщения о них всегда появлялись на терминале:

```
$ diff file1 fiel2 >diff.out
diff: fiel2: No such file or directory
$
```

И в самом деле желательно, чтобы сообщения об ошибках обрабатывались именно так, ведь вряд ли можно назвать удачным способ, при котором сообщения исчезали бы в файле diff.out, оставляя пользователя в полной уверенности, что команда отработала правильно.

При запуске любой программы по умолчанию порождаются три файла, пронумерованные небольшими целыми числами, которые называются *дескрипторами файла* (вернемся к ним в главе 7). Стандартный ввод, 0, и стандартный вывод, 1, с которыми мы уже хорошо знакомы, часто перенаправляются в файлы и каналы. Последний файл с номером 2 – это *стандартный вывод ошибок*, который обычно попадает на терминал.

Иногда программы производят вывод в файл стандартного вывода ошибок, даже если работают правильно. Так поступает программа time, которая выполняет команду и записывает время исполнения в стандартный вывод ошибок:

```
$ time wc ch3.1
    931   4288   22691 ch3.1

real        1.0
user        0.4
sys         0.4
$ time wc ch3.1 >wc.out

real        2.0
user        0.4
sys         0.3
$ time wc ch3.1 >wc.out 2>time.out
$ cat time.out

real        1.0
user        0.4
sys         0.3
$
```

Конструкция 2>*имя-файла* (пробелы между 2 и знаком > недопустимы) направляет стандартный вывод ошибок в файл; она синтаксически тяжеловесна, но функцию свою выполняет. (Программа time в нашем коротком тесте выводит не очень точное время; но если проводить серию длительных тестов, то полученные цифры будут полезными и достаточно достоверными, так что может возникнуть желание сохранить их для последующего анализа; см. табл. 8.1.)

Также существует возможность объединения двух выходных потоков:

```
$ time wc ch3.1 >wc.out 2>&1
$ cat wc.out
         931   4288   22691 ch3.1

real        1.0
user        0.4
sys         0.3
$
```

Запись 2>&1 говорит оболочке, что следует поместить вывод стандартных ошибок в тот же поток, что и стандартный вывод. Это не какое-то дополнительное мнемоническое значение амперсанда, а просто идиома, которую надо запомнить. Можно использовать и 1>&2 – для того, чтобы добавить стандартный вывод к стандартному выводу ошибок:

```
echo ... 1>&2
```

выводит данные на стандартный вывод ошибок. В командных файлах таким способом предотвращается исчезновение сообщений в каналах или файлах.

В оболочке имеется механизм, позволяющий поместить стандартный ввод команды вместе с командой, а не в отдельном файле, так что командный файл может быть абсолютно независимым. Напишем еще раз программу справочной службы 411:

```
$ cat 411
grep "$*" <<End
dial-a-joke  212-976-3838
dial-a-prayer  212-246-4200
dial santa  212-976-3636
dow jones report  212-976-4141
End
$
```

На жаргоне оболочки такая конструкция называется *встроенным документом* (*here document*); это означает, что входные данные находятся там же, где и сама команда, а не в каком-то отдельном файле. Символ << обозначает эту конструкцию; слово, следующее за ним (в данном примере – End), служит для того, чтобы ограничить ввод, который понимается как все, что предшествует такому слову, находящемуся в отдельной строке. Оболочка заменяет символы $, `...` и \ во входных

данных встроенного документа, если только какая-то часть слова не заключена в кавычки или не использована обратная косая черта; в этом случае весь документ воспринимается буквально.

В конце главы мы вернемся к встроенным документам и приведем гораздо более интересные примеры их использования.

В табл. 3.2 приведен список различных обозначений для перенаправления ввода-вывода, понятных оболочке.

Упражнение 3.14. Сравните программу `411`, содержащую встроенный документ, с ее первоначальной версией. Какую из них легче поддерживать? Какую лучше использовать для создания всеобщей службы? □

Таблица 3.2. Перенаправление ввода-вывода в оболочке

Обозначение	Действие
>*file*	направляет стандартный вывод в *file*
>>*file*	добавляет стандартный вывод в *file*
<*file*	получает стандартный ввод из *file*
p_1|p_2	соединяет стандартный вывод программы p_1 со вводом p_2
^	устаревший синоним для |
n>*file*	прямой вывод из файлового дескриптора *n* в *file*
n>>*file*	добавляет вывод из файлового дескриптора *n* в *file*
n>&*m*	объединяет вывод из файлового дескриптора *n* с файловым дескриптором *m*
n<&*m*	объединяет ввод из файлового дескриптора *n* с файловым дескриптором *m*
<<*s*	встроенный документ: получает стандартный ввод до тех пор, пока в начале строки не встретится s; обрабатываются символы $, `...` и \
<<*s*	встроенный документ без подстановок
<<'*s*'	встроенный документ без подстановок

3.8. Циклы в программах оболочки

Фактически оболочку можно назвать языком программирования: в ней есть переменные, циклы, принятие решений и т. д. Основы организации циклов будут рассмотрены в этом разделе, а об управляющей логике программы поговорим в главе 5.

Организация цикла по множеству имен файлов применяется повсеместно, при этом единственным оператором управления в оболочке, который можно вводить на терминале, а не помещать в файл для последующего исполнения, является оператор `for`. Синтаксис `for`:

```
for var in список слов
```

```
do
        команды
done
```

Например, с помощью оператора for можно организовать вывод имен файлов по одному на строке:

```
$ for i in *
> do
>       echo $i
> done
```

На месте i может стоять любая переменная оболочки, но традиционно это i. Обратите внимание, что доступ к значению переменной обеспечивается посредством $i, но цикл for ссылается на переменную как на i. В данном примере был использован шаблон * – для того, чтобы охватить все файлы в текущем каталоге; можно использовать и любые другие списки аргументов. Обычно задачи бывают более интересными, чем просто вывод имен файлов. Часто приходится сравнивать множество файлов с более ранними версиями. Например, чтобы сравнить старую версию главы 2 (она хранится в каталоге /old) с текущей:

```
$ ls ch2.* | 5
ch2.1        ch2.2        ch2.3        ch2.4        ch2.5
ch2.6        ch2.7
$ for i in ch2.*
> do
>       echo $i:
>       diff -b old/$i $i
>       echo                Добавить пустую строку для читаемости
> done | pr -h "diff `pwd`/old `pwd`" | lpr &
3712                        Идентификатор процесса
$
```

Мы организовали конвейерное перенаправление в pr и lpr только для того, чтобы показать, что стандартный вывод команд внутри цикла for становится стандартным выводом самого for. С помощью параметра -h команды pr был создан красивый заголовок для вывода (два вложенных вызова pwd). И еще мы запустили всю эту последовательность команд в асинхронном режиме (&), так что нам не надо ожидать ее завершения; & применяется сразу ко всему циклу и конвейеру.

Авторы предпочитают такой формат оператора for, который использован в примере, но возможны и другие варианты. Приведем основное ограничение: do и done воспринимаются как зарезервированные слова, только если они появляются после символа новой строки или точки с запятой. В зависимости от размеров for, иногда бывает удобнее записать все в одной строке:

```
for i in список; do команды; done
```

Цикл `for` следует использовать для большого количества команд и в случаях, когда обработка встроенного аргумента отдельной командой неприемлема.

Но не применяйте его для одной команды, которая будет перебирать имена файлов:

```
# Плохая идея:
for i in $*
do
        chmod +x $i
done
```

хуже, чем

```
chmod +x $*
```

потому что цикл `for` выполняет отдельную команду `chmod` для каждого файла, что требует значительно больших компьютерных ресурсов. (Убедитесь, что вы понимаете различие между

```
for i in *
```

где перебираются все файлы в текущем каталоге, и

```
for i in $*
```

где перебираются все аргументы командного файла.)

Список аргументов для цикла `for` обычно образуется по шаблону для имен файлов, но может быть получен и другим способом. Это может быть

```
$ for i in `cat ...`
```

Но можно просто ввести аргументы с терминала. Например, ранее в этой же главе была создана группа программ для вывода в несколько колонок (они были названы `2`, `3` и т. д.). Они представляют собой просто ссылки (links) на один и тот же файл, которые могут быть сделаны (после того как единожды был записан файл `2`) при помощи команды

```
$ for i in 3 4 5 6; do ln 2 $i; done
$
```

Приведем более интересный пример применения цикла `for` — используем `pick`, чтобы выбрать файлы, которые будут сравниваться с файлами из каталога резервного копирования:

```
$ for i in `pick ch2.*`
> do
>         echo $i:
>         diff old/$i $i
> done | pr | lpr
```

```
ch2.1? y
ch2.2?
ch2.3?
ch2.4? y
ch2.5? y
ch2.6?
ch2.7?
$
```

Очевидно, этот цикл должен быть помещен в командный файл, чтобы в следующий раз не пришлось набирать все заново: если что-то проделано дважды, вероятнее всего, это придется повторить снова.

Упражнение 3.15. Если бы цикл `diff` был помещен в командный файл, стоило бы помещать в командный файл `pick`? Обоснуйте ответ. □

Упражнение 3.16. Что произошло бы, если бы последняя строка цикла, рассмотренного выше, выглядела так:

```
> done | pr | lpr &
```

то есть заканчивалась бы амперсандом? Попытайтесь догадаться, а потом проверьте себя за компьютером. □

3.9. Команда bundle: сложим все вместе

Для того чтобы осознать особенности создания командных файлов, давайте напишем большую программу. Предположим, что получено письмо от коллеги, работающего на другом компьютере (например, `somewhere!bob`[1]); он хотел бы скопировать командные файлы из вашего каталога `/bin`. Самый простой способ – это послать их ему в ответе на его письмо. Для начала введите:

```
$ cd /usr/you/bin
$ for i in `pick *`
> do
>       echo ============ This is file $i ============
>       cat $i
> done | mail somewhere!bob
$
```

Но давайте встанем на место `somewhere!bob`: он получит письмо, содержащее все файлы; они, конечно, будут разделены, но потребуется редактор, чтобы выделить каждый компонент в файл. Внутренний голос подсказывает, что правильно построенное почтовое сообщение могло бы автоматически самораспаковываться, так что получателю не при-

[1] Существуют различные обозначения для записи адресов удаленных машин. Самое распространенное из них – это *компьютер*!*пользователь* (см. `mail(1)`).

шлось бы ничего делать вручную. Это подразумевает создание командного файла, содержащего как файлы, так и инструкции по распаковке.

Интуиция подсказывает и второе соображение – удобным способом комбинирования вызова команды и данных для этой команды является встроенный документ. Теперь осталось только правильно расставить кавычки. Получилась работающая программа `bundle`, которая сгруппировывает файлы в автономный командный файл на своем стандартном выводе:

```
$ cat bundle
# bundle: сгруппировать файлы в дистрибутивный пакет

echo '# Чтобы распаковать, запустите этот файл'
for i
do
      echo "echo $i 1>&2"
      echo "cat >$i <<'End of $i'"
      cat $i
      echo "End of $i"
done
$
```

Заключение в кавычки `"End of $i"` гарантирует, что все метасимволы оболочки в файлах будут проигнорированы.

Конечно же, стоит опробовать программу, прежде чем использовать ее для отправки файлов `somewhere!bob`:

```
$ bundle cx lc >junk                    Создать пробную команду bundle
$ cat junk
# Чтобы распаковать, запустите этот файл
echo cx 1>&2
cat >cx <<'End of cx'
chmod +x $*
End of cx
echo lc 1>&2
cat >lc <<'End of lc'
# lc: сосчитать количество строк в файлах
wc -l $*
End of lc
$ mkdir test
$ cd test
$ sh ../junk                            Проверим ее
cx
lc
$ ls
cx
lc
$ cat cx
chmod +x $*
```

```
$ cat lc
# lc: сосчитать количество строк в файлах
wc -l $*                                    Похоже, все нормально
$ cd ..
$ rm junk test/*; rmdir test                Очистить
$ pwd
/usr/you/bin
$ bundle `pick *` | mail somewhere!bob      Отправить файлы
```

Наличие в одном из отправляемых файлов строки

```
End of имя-файла
```

может создать проблему, но вероятность такого события невелика. Чтобы сделать `bundle` крайне надежной, можно прибегнуть еще к нескольким приемам, описанным в следующих главах, но фактически программа может быть использована и в своем теперешнем состоянии.

Программа `bundle` представляет собой яркую иллюстрацию гибкости среды UNIX: она использует циклы оболочки, перенаправление ввода-вывода, встроенные документы и командные файлы, непосредственно взаимодействует с `mail`. Но, наверное, самое интересное заключается в том, что `bundle` – это программа, которая создает программу. По мнению авторов, `bundle` является одной из самых красивых программ оболочки – всего несколько строк кода, которые реализуют простое и элегантное решение важной задачи.

Упражнение 3.17. Как использовать `bundle` для отправки всех файлов каталога и его подкаталогов? Подсказка: командные файлы могут быть рекурсивными. □

Упражнение 3.18. Измените `bundle` так, чтобы вместе с каждым файлом отправлялась и информация, полученная от `ls -l`, в частности права доступа и дата последнего изменения. Сравните возможности `bundle` с возможностями программы сжатия данных `ar(1)`. □

3.10. Зачем нужна программируемая оболочка?

Оболочка UNIX не является типичным примером командного процессора: хотя она и позволяет запускать программы привычным способом, но, будучи языком программирования, она способна и на большее. Давайте ненадолго вернемся назад к тому, что уже было описано. Это стоит сделать, во-первых, потому что в этой главе представлено очень много материала, а во-вторых, потому что, пообещав рассказать о «наиболее часто используемых свойствах оболочки», авторы посвятили 30 страниц примерам программирования в оболочке. Но дело в том, что используя оболочку, вы все время пишете маленькие однострочные программы: конвейер – это программа, такая же как и при-

мер с «Чай готов». Оболочка работает следующим образом: пользователь постоянно программирует ее, но делать это настолько просто и естественно (когда научишься), что он не воспринимает этот процесс как программирование.

Оболочка сама занимается некоторыми вещами, например организацией циклов, перенаправлением ввода-вывода при помощи < и >, расширением имен файлов посредством *, так что программам не надо беспокоиться об этом, и, что еще важнее, применение этих средств единообразно для любых программ. Другие свойства (командные файлы, каналы) на самом деле предоставляются ядром, а оболочка обеспечивает естественный синтаксис для их создания. Они выходят за пределы простого обеспечения удобства пользователя, фактически расширяя возможности системы.

Оболочка обязана ядру UNIX большей частью своих свойств: например, хотя оболочка организует каналы, но перемещение данных по ним выполняется ядром. Благодаря тому что система обрабатывает исполняемые файлы специальным образом, можно создать командный файл так, чтобы он работал так же, как и откомпилированная программа. Пользователю не надо знать, что это командные файлы – для обращения к ним не требуется специальной команды типа `RUN`. К тому же оболочка сама является программой, а не частью ядра, поэтому ее можно настраивать, расширять и использовать, как и любые другие программы. Такая концепция воплощена не только в системе UNIX, но здесь она реализована наилучшим образом.

Разговор о программировании оболочки будет продолжен в главе 5, а пока запомните одну вещь: что бы вы ни делали в оболочке, вы ее программируете, именно этим в значительной степени объясняется то, что она так хорошо работает.

История и библиография

Оболочка была программируемой с давних пор. Первоначально существовали отдельные команды для `if`, `goto` и метки; при запуске команда `goto` сканировала входной файл с самого начала в поиске нужной метки. (Так как было невозможно заново прочитать канал, было невозможно и направить данные в командный файл, в котором использовалась управляющая логика).

Оболочка пользователя в седьмой версии была создана Стивом Бурном (Steve Bourne), Джон Мэши (John Mashey) помогал ему и подал некоторые идеи. В этой редакции содержится все, что нужно для программирования, вы увидите это в главе 5. Кроме того, усовершенствованы ввод и вывод: предоставлена возможность перенаправлять ввод-вывод в программы оболочки и из них без ограничений. Разбор метасимволов в именах файлов является внутренним свойством этой оболочки; в

предыдущих версиях существовала специальная программа, которая должна была работать на очень маленьких машинах.

Вам могла встретиться и другая оболочка (а может быть, именно ее вы и используете), `csh` – так называемая «Си-оболочка», которая была разработана в Беркли Биллом Джоем (Bill Joy) на основе оболочки шестой версии. «Си-оболочка» пошла дальше, чем оболочка Бурна, в направлении интерактивной поддержки. Наиболее примечательным является механизм сохранения предыстории, который делает возможным повторение (вероятно, с незначительным редактированием) ранее выполненных команд. Синтаксис также несколько отличается. Поскольку «Си-оболочка» основана на более ранней оболочке, в ней меньше возможностей для программирования; она является скорее интерактивным командным процессором, чем языком программирования. В частности, невозможно осуществлять направление данных в управляющие операторы и из них.

Программа `pick` была создана Томом Даффом (Tom Duff), а команда `bundle` была независимо придумана Аланом Хьюитом (Alan Hewett) и Джеймсом Гослингом (James Gosling).

4

Фильтры

Существует большое семейство программ для UNIX, вводящих некоторые данные, выполняющих простые преобразования и что-либо выводящих. Примером могут служить `grep` и `tail` для выбора данных из входного потока, `sort` для их сортировки, `wc` для подсчета и т. д. Такие программы называются *фильтрами*.

В этой главе мы рассмотрим наиболее часто используемые фильтры. Начнем с `grep` и попробуем применить более сложные шаблоны, чем в главе 1. Познакомимся также с двумя другими членами семейства `grep`: `egrep` и `fgrep`.

В последующих разделах кратко описаны несколько других полезных фильтров, в том числе `tr` для подстановки символов (транслитерации), `dd` для использования данных, полученных из других систем, и `uniq` для поиска повторяющихся текстовых строк. Команда `sort` также рассмотрена более подробно, нежели в главе 1.

Оставшаяся часть главы посвящена двум универсальным «преобразователям данных», или «программируемым фильтрам», названным так за то, что для описания правил преобразования данных в них используется простой язык программирования. Разные правила могут давать очень разные результаты.

Это программы `sed` (stream editor – потоковый редактор) и `awk`, названная в честь своих авторов. Обе они получены обобщением команды `grep`:

```
$ program шаблон-действие имена-файлов ...
```

Эта команда просматривает файлы из списка в поисках строк, соответствующих образцу, и, найдя таковые, выполняет над ними указанное действие. Для `grep` образцом служит регулярное выражение, как в ре-

дакторе ed, а действием по умолчанию – вывод строк, соответствующих образцу.

Программы sed и awk обобщают понятия образца и действия, в частности sed, будучи производной от ed, выполняет «программу», составленную из команд редактора над каждой строкой, полученной из входных файлов. Программа awk не так удобна для текстовых подстановок, как sed, зато она поддерживает арифметические операции, переменные, встроенные функции и язык программирования, весьма напоминающий Си. Данная глава не содержит исчерпывающего описания этих программ; оно есть в томе 2B справочного руководства по системе UNIX (см. *http://cm.bell-labs.com/7thEdMan/*).

4.1. Семейство программ grep

Программа grep была упомянута в главе 1 и с тех пор неоднократно использовалась в примерах.

```
$ grep шаблон имена-файлов ...
```

просматривает указанные файлы либо стандартный ввод и выводит все строки, содержащие образец. Программа незаменима при поиске переменных в программах, слов в документах или в выходных данных какой-либо программы.

```
$ grep -n variable *.[ch]      Найти variable в программах на Си
$ grep From $MAIL              Напечатать заголовки почтовых сообщений
$ grep From $MAIL | grep -v mary       Заголовки сообщений,
                                       не принадлежащих mary
$ grep -y mary $HOME/lib/phone-book    Найти телефонный номер mary
$ who | grep mary              Зарегистрировалась ли mary в системе?
$ ls | grep -v temp            Имена файлов, не содержащие temp
```

Параметр -n выводит номера строк, -v изменяет условие на противоположное, а -y устанавливает соответствие символов нижнего регистра в образце символам обоих регистров в файле (символы верхнего регистра по-прежнему соответствуют только верхнему регистру).

До сих пор во всех примерах программа grep выполняла поиск обычных строк, состоящих из букв и цифр. Но в действительности она может находить и более сложные образцы: grep может интерпретировать выражения на простом языке описания строк.

С технической точки зрения шаблоны являются ограниченным вариантом описателей строк, известных как *регулярные выражения*. Программа grep интерпретирует те же регулярные выражения, что и редактор ed; фактически она была создана из него (всего за один вечер) путем «хирургического вмешательства».

Регулярные выражения образуются присваиванием специальных значений определенным символам, таким, например, как *, которые ис-

пользуются оболочкой. Существует еще несколько метасимволов, имеющих, к сожалению, разночтения в применении. Рассмотрим их вкратце (список всех метасимволов, используемых в регулярных выражениях, приведен в табл. 4.1).

Таблица 4.1. Регулярные выражения grep и egrep (в порядке понижения приоритета)

Метасимволы	Значение
c	любой не специальный символ *c* совпадает сам с собой
c	отключает все специальные значения символа *c*
^	начало строки
$	конец строки
.	произвольный одиночный символ
[...]	любой из символов множества ...; допустимы диапазоны, например a-z
[^...]	любой символ, не входящий в ...; допустимы диапазоны
n	то же, что в *n*-м фрагменте \(...\) (только grep)
*r**	ноль или больше вхождений *r*
r+	одно или больше вхождений *r* (только egrep)
r?	ноль или одно вхождение *r* (только egrep)
r1r2	*r2* следует за *r1*
r1|*r2*	*r1* или *r2* (только egrep)
\(*r*\)	регулярное выражение *r* с тегами (только grep); может быть вложенным
(*r*)	регулярное выражение *r* (только egrep); может быть вложенным
	не существует регулярного выражения, соответствующего символу новой строки

Метасимволы ^ и $ «привязывают» шаблон к началу или концу строки соответственно. Например, выражение

```
$ grep From $MAIL
```

находит в сообщениях строки, содержащие образец From, а

```
$ grep '^From' $MAIL
```

выберет только те строки, которые *начинаются* с From, а значит, скорее всего, находятся в заголовке сообщения. Метасимволы регулярных выражений перекрываются метасимволами оболочки, поэтому параметр команды grep рекомендуется заключать в одинарные кавычки.

Программа grep, как и оболочка, поддерживает «классы символов», так что [a-z] соответствует любой букве нижнего регистра. Но есть и

отличия: если класс символов начинается со знака ^, то шаблону соответствуют все символы, *кроме* принадлежащих классу. Следовательно, выражению [^0-9] соответствуют все символы, не являющиеся цифрами. Кроме того, в оболочке символ \ перед символами] и – означает, что они входят в класс, тогда как программы grep и ed требуют, чтобы эти символы использовались там, где исключена двусмысленность. Например, последовательность []-] (это важно!) соответствует открывающей и закрывающей квадратным скобкам и знаку минус.

Знак . (точка) эквивалентен вопросительному знаку в оболочке, он соответствует любому символу. (Пожалуй, точка – это знак, имеющий больше всего различных значений в разных программах для UNIX.) Вот пара примеров:

```
$ ls -l | grep '^d'              Список подкаталогов
$ ls -l | grep '^.......rw'      Файлы, доступные остальным
                                   для чтения и записи
```

Знак ^ и следующие за ним семь точек соответствуют семи произвольным символам в начале строки; в случае применения к выходному потоку команда ls -l задает строку прав доступа.

Оператор *замыкания* (*closure*) * действует на предыдущий символ либо метасимвол (включая и класс символов) в выражении, такая конструкция соответствует произвольному числу следующих друг за другом символов, заданных в образце. Например, x* определяет последовательность символов x максимально возможной длины, [a-zA-Z]* определяет строку из букв, .* соответствует строке любых символов, а .*x – строке произвольных символов, заканчивающейся символом x, включая этот *последний* символ.

Отметим две важные особенности замыкания. Во-первых, оператор замыкания действует только на один символ, так что xy* соответствует нескольким символам y, следующим за символом x, а не строке вида xyxyxy. Во-вторых, «произвольное количество» включает ноль, поэтому, если необходим хотя бы один символ, надо указывать его отдельно. Например, правильное выражение для определения буквенной строки выглядит так: [a-zA-Z][a-zA-Z]* (буква и следующие за ней ноль или более букв). Шаблонный символ * для имени файла в оболочке – это то же, что и регулярное выражение .*.

В grep нет регулярного выражения, которое соответствовало бы символу новой строки, каждая строка обрабатывается отдельно.

Вместе с регулярными выражениями grep представляет собой простой язык программирования. Вспомним, например, что зашифрованный пароль находится во втором поле файла паролей. Для поиска пользователей, не имеющих пароля, введем команду:

```
$ grep '^[^:]*::' /etc/passwd
```

Это означает: начало строки, произвольное количество «недвоеточий», два двоеточия.

Программа `grep` – старейший член семейства, в котором позже появились `fgrep` и `egrep`. Сущность их работы одинакова, за исключением того, что `fgrep` выполняет поиск одновременно нескольких строк, а `egrep` интерпретирует истинные регулярные выражения так же, как `grep`, но с использованием оператора ИЛИ и скобок, позволяющих группировать выражения, как это описано ниже.

Обе команды позволяют при помощи параметра `-f` указать файл, содержащий шаблон. В файле могут находиться несколько шаблонов, каждый на отдельной строке, поиск по которым выполняется параллельно. Например, если вы часто делаете ошибки в одних и тех же словах, то можете сохранить их в файле, по одному в каждой строке, и с помощью `fgrep` проверять их наличие в своих документах:

```
$ fgrep -f common-errors document
```

Команда `egrep` обрабатывает те же регулярные выражения, что и `grep` (см. табл. 4.1), но с несколькими дополнениями. Скобки позволяют группировать выражения, так что `(xy)*` определяет пустую строку, `xy`, `xyxy`, `xyxyxy` и т. д. Оператор или записывается как вертикальная черта `|`; выражение `today|tomorrow` означает `today` либо `tomorrow` – так же, как и выражение `to(day|morrow)`. И наконец, в `egrep` имеются два дополнительных оператора замыкания, `+` и `?`. Шаблон `x+` определяет один или больше символов `x`, а `x?` – ноль или один символ, но не больше.

Программа `egrep` хорошо подходит для поиска слов с заданными свойствами в словаре. Мы работаем со словарем Webster's Second International, который хранится в системе в виде списка слов, по одному на строку, без словарных статей. Вы можете использовать словарь `/usr/dict/words`. Он не так велик и предназначен для проверки орфографии; проверьте, в каком формате хранятся в нем слова. Приведем шаблон для поиска слов, содержащих все гласные в алфавитном порядке:

```
$ cat alphvowels
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$
$ egrep -f alphvowels /usr/dict/web2 | 3
abstemious      abstemiously      abstentious
achelious       acheirous         acleistous
affectious      annelidous        arsenious
arterious       bacterious        caesious
facetious       facetiously       fracedinous
majestious
$
```

Шаблон в файле `alphvowels` не заключается в кавычки. Если шаблон `egrep` находится в кавычках, то оболочка не интерпретирует его сама, а лишь убирает кавычки, поэтому `egrep` никогда их не видит. Но так как файл оболочкой не обрабатывается, то и кавычки в нем *не* требуются.

В вышеприведенном примере можно было использовать команду grep, но egrep выполняет поиск по шаблонам с замыканиями намного быстрее, особенно в больших файлах.

Вот еще один пример – поиск слов, состоящих из шести и более букв, расположенных в алфавитном порядке:

```
$ cat monotonic
^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?$
$ egrep -f monotonic /usr/dict/web2 | grep '......' | 5
abdest     acknow     adipsy     agnosy     almost
befist     behint     beknow     bijoux     biopsy
chintz     dehors     dehort     deinos     dimpsy
egilops    ghosty
$
```

(Egilops – это болезнь пшеницы.)[1] Обратите внимание на использование grep для фильтрации вывода egrep.

Зачем нужны три программы grep? Дело в том, что fgrep, хоть и не умеет обрабатывать метасимволы, зато эффективно выполняет поиск нескольких тысяч слов одновременно (после инициализации время выполнения не зависит от количества слов), и вследствие этого применяется главным образом в таких задачах, как библиографический поиск. Обычно размер шаблона fgrep намного превосходит возможности алгоритмов, реализованных в grep и egrep. Сложнее объяснить разницу между grep и egrep. Программа grep появилась намного раньше остальных, она поддерживает синтаксис регулярных выражений, знакомый по редактору ed, обрабатывает регулярные выражения с тегами и имеет большой набор параметров. Программа egrep обрабатывает больше различных выражений (за исключением тегов) и выполняется намного быстрее (причем скорость не зависит от вида шаблона), но стандартная версия дольше стартует на сложных выражениях. Существует более новая версия, которая стартует быстрее, так что egrep и grep могут быть объединены в одну программу поиска по шаблону.

Упражнение 4.1. Изучите регулярные выражения с тегами (\(и \)) в приложении 1 или в ed(1) и используйте grep для поиска палиндромов – слов, пишущихся одинаково в обоих направлениях. Подсказка: задайте отдельные шаблоны для каждой длины слова. □

Упражнение 4.2. Схема работы grep заключается в чтении одной строки, проверке совпадений и повторении цикла. Как может отразиться на grep возможность задания в шаблоне символа новой строки? □

---

[1] Трудно понять, почему авторы сочли уместным пояснить значение именно этого слова. Расширим комментарий: abdest – (у мусульман) ритуал омовения рук перед молитвой; dehors – снаружи, за пределами; (to) dehort – отговаривать, разубеждать; dimpsy – тусклый, сумеречный. – *Примеч. ред.*

4.2. Другие фильтры

Цель этого раздела – познакомить читателя с богатым набором фильтров, поставляемых с системой, и показать на примерах, как они используются. Этот список ни в коем случае не претендует на полноту – в составе седьмой версии есть много других фильтров, а в каждой установленной системе со временем появляются собственные реализации. Все стандартные фильтры описаны в первом разделе руководства.

Начнем с программы `sort`, возможно, наиболее полезной из всех. Основные сведения о ней были приведены в главе 1: программа сортирует поступающие на вход строки по значениям кодов ASCII-символов. Кроме этого очевидного действия, выполняемого по умолчанию, есть множество других способов сортировки, для каждого из которых `sort` имеет соответствующий параметр. Например, параметр `-f` вызывает «свертку» нижнего и верхнего регистров, так что различия между ними игнорируются. Параметр `-d` (словарный порядок) игнорирует при сравнении любые символы, кроме букв, цифр и пробелов.

Чаще всего выполняется сравнение по символам, но иногда требуется сравнивать числа. Параметр `-n` обеспечивает сортировку по числовым значениям, а параметр `-r` изменяет результат операции сравнения на противоположный. Таким образом,

```
$ ls | sort -f       Сортирует файлы в алфавитном порядке
$ ls -s | sort -n    Сортирует файлы по возрастанию размера
$ ls -s | sort -nr   Сортирует файлы по убыванию размера
```

По умолчанию `sort` сравнивает строки целиком, но ей можно указать, по какому конкретному полю должна выполняться сортировка. Запись +*m* означает, что первые *m* полей должны быть проигнорированы при сравнении; `+0` – это сравнение с начала строки. Например,

```
$ ls -l | sort +3nr      Сортировка по размеру по убыванию
$ who | sort +4n         Сортировка по времени регистрации по убыванию
```

Среди других полезных параметров сортировки упомянем `-o`, который определяет выходной файл (это может быть один из входных файлов), и `-u`, позволяющий игнорировать строки, совпадающие по полю сортировки.

Можно задать несколько параметров сортировки, как в примере, взятом со страницы руководства `sort(1)`:

```
$ sort +0f +0 -u имена-файлов
```

Здесь `+0f` сортирует строки без учета регистра, однако идентичные строки не обязательно соседствуют. Следующий параметр `+0` выполняет сортировку совпадающих строк в обычном алфавитном порядке. Наконец, `-u` исключает дубликаты строк. Таким образом, получив набор слов, по одному на строке, команда выведет список неповторяю-

щихся слов. Предметный указатель оригинального издания этой книги был подготовлен аналогичной командой `sort` с использованием дополнительных возможностей, описанных в `sort(1)`.

Команда `uniq` послужила прототипом флага `-u` команды `sort`: она отбрасывает смежные дублирующиеся строки, оставляя только одну. Наличие для этой цели отдельной программы позволяет решать задачи, не связанные с сортировкой. Например, `uniq` может удалить повторяющиеся пустые строки, даже если входной файл не отсортирован.

Параметры позволяют выбрать способ обработки дублей: `uniq -d` выведет только те строки, которые были дублированы; `uniq -u` выводит уникальные строки (т. е. не повторяющиеся); а `uniq -c` подсчитывает количество вхождений каждой строки. Ниже будет приведен пример.

Команда `comm` применяется для сравнения файлов. Получив два *отсортированных* входных файла, `comm` выведет результат в три столбца: строки, которые встречаются только в первом файле, строки, встречающиеся только во втором и строки, встретившиеся в обоих файлах. Вывод любого из столбцов можно отменить параметром:

```
$ comm -12 f1 f2
```

выведет только строки, вошедшие в оба файла, а

```
$ comm -23 f1 f2
```

выведет строки, присутствующие только в первом файле и отсутствующие во втором. Это можно использовать для сравнения каталогов или списка слов со словарем.

Команда `tr` выполняет замену символов во входном файле. Чаще всего она применяется для замены регистра:

```
$ tr a-z A-Z      Заменить нижний регистр на верхний
$ tr A-Z a-z      Заменить верхний регистр на нижний
```

Команда `dd` заметно отличается от всех команд, рассмотренных до сих пор. Она предназначена главным образом для обработки данных на магнитных лентах из других систем – само имя напоминает о языке управления заданиями OS/360.[1] Команда может выполнять замену регистра (с синтаксисом, существенно отличающимся от `tr`); преобразование кода ASCII в EBCDIC и обратно; чтение и запись данных блоками фиксированного размера с заполнением отсутствующих полей пробелами, что используется в некоторых других операционных системах. На практике команда `dd` часто применяется для работы с «сырыми» неформатированными данными независимо от их происхождения, она также имеет возможность обработки двоичных данных.

[1] Язык управления заданиями (Job Control Language, JCL) для IBM System/360 содержал подробную спецификацию определения наборов данных (Dataset Definition, DD) для устройств ввода-вывода. – *Примеч. ред.*

Чтобы продемонстрировать возможности совместной работы фильтров, рассмотрим конвейер из команд, который выводит 10 слов, наиболее часто встречающихся во входном файле:

```
cat $* |
tr -sc A-Za-z '\012' |   Заменяет все небуквы на символы новой строки
sort |
uniq -c |
sort -n |
tail |
5
```

Команда `cat` занимается сбором файлов и передает их во входной поток `tr`. Команда `tr` взята из руководства: она заменяет все последовательности символов, не являющихся буквами, символом новой строки, помещая, таким образом, каждое слово на отдельную строку. Затем список сортируется, и команда `uniq -c` удаляет повторяющиеся слова, вставляя перед оставшимися словами счетчик вхождений, по которому выполняется сортировка командой `sort -n`. (Такое сочетание двух команд `sort`, окружающих команду `uniq`, встречается настолько часто, что может считаться идиомой.) В результате получен список слов, встречающихся в документе, отсортированный в порядке увеличения частоты использования.

Команда `tail` выводит 10 самых популярных слов, а `5` располагает их в пяти столбцах.

Кстати, обратите внимание, что команда, заканчивающаяся символом `|`, может быть продолжена на следующей строке.

Упражнение 4.3. Используйте команды этого раздела для программы проверки орфографии по словарю `/usr/dict/words`. В чем недостатки такого подхода и как их исправить? □

Упражнение 4.4. Напишите программу подсчета слов на своем любимом языке программирования и сравните ее размер, скорость выполнения и гибкость с конвейером команд. Сколько потребуется усилий для превращения ее в программу проверки орфографии? □

4.3. Потоковый редактор sed

Теперь рассмотрим программу `sed`. Это будет не очень сложно сделать, так как она является прямым потомком редактора `ed`, знакомство с ним облегает задачу.

Основная идея `sed` очень проста:

```
$ sed 'список команд ed' имена-файлов
```

читает построчно входной файл; выполняет поочередно команды из списка над каждой строкой и выводит результат на стандартное уст-

ройство вывода. Так, можно заменить все вхождения UNIX на UNIX(TM) в группе файлов командой

```
$ sed 's/UNIX/UNIX(TM)/g' имена-файлов … >output
```

Не ошибитесь в том, что здесь происходит: sed *не* изменяет содержимого входных файлов, а использует стандартный вывод, оставляя исходные файлы нетронутыми. Вы уже достаточно хорошо знаете оболочку, чтобы понимать, что не следует писать:

```
$ sed '…' file >file
```

Для замены содержимого файла следует использовать либо временный файл, либо другую программу. (Позже мы обсудим программу, реализующую идею перезаписи существующего файла; см. overwrite в главе 5.)

Вывод каждой строки в sed выполняется автоматически, поэтому нет необходимости добавлять p после команды подстановки, – если это сделать, то каждая измененная строка будет напечатана дважды. В то же время кавычки почти всегда необходимы, так как многие метасимволы sed имеют собственное значение в оболочке. Рассмотрим, например, команду du -a, печатающую список файлов. Обычно du выводит размер и имя файла:

```
$ du -a ch4.*
18          ch4.1
13          ch4.2
14          ch4.3
17          ch4.4
2           ch4.9
$
```

Удаление столбца с размерами можно осуществить с помощью sed, но команда редактирования требует наличия кавычек, чтобы не допустить интерпретации оболочкой символа * и знака табуляции:

```
$ du -a ch4.* | sed 's/.*→//'
ch4.1
ch4.2
ch4.3
ch4.4
ch4.9
$
```

В процессе подстановки удаляются все символы (.*) до последнего справа знака табуляции включительно (в шаблоне он показан знаком →)[1].

[1] В современных реализациях оболочки пользователя, например в bash, для помещения в командную строку «настоящего» символа табуляции необходимо непосредственно перед ним ввести *Ctrl-V*. Редактор командной строки оболочки csh обычно использует по умолчанию для введения символов, не требующих интерпретации, этот же управляющий символ. – *Примеч. науч. ред.*

Аналогичным способом можно получить имена пользователей и время регистрации из данных, выводимых командой who:

```
$ who
lr       tty1         Sep 29 07:14
ron      tty3         Sep 29 10:31
you      tty4         Sep 29 08:36
td       tty5         Sep 29 08:47
$ who | sed 's/ .* / /'
lr 07:14
ron 10:31
you 08:36
td 08:47
$
```

Команда s выполняет замену пробела и всего, что за ним следует, включая следующий пробел, единственным пробелом. Здесь тоже необходимы кавычки.

Практически такая же команда sed применяется в программе getname, возвращающей имя текущего пользователя:

```
$ cat getname
who am i | sed 's/ .*//'
$ getname
you
$
```

Еще одна команда sed применяется столь часто, что мы встроили ее в оболочку под именем ind. Команда ind добавляет во входной файл отступ слева на один шаг табуляции, это позволяет лучше расположить текст при печати на принтере.

Реализация ind чрезвычайно проста – в начало каждой строки добавляется знак табуляции:

```
sed 's/^/→/' $*          ind, версия 1
```

Эта версия вставляет табуляцию и в пустые строки, что совершенно излишне. Улучшенная версия основывается на способности sed выбирать строки, подлежащие изменению. Если в начало команды добавить шаблон, то будут обработаны только соответствующие ему строки:

```
sed '/./s/^/→/' $*      ind, версия 2
```

Шаблон /./ определяет строку, содержащую хотя бы один символ, отличный от символа новой строки; команда s выполняется только для непустых строк. Учтите, что sed выводит все строки независимо от того, вносились ли в них изменения, поэтому пустые строки по-прежнему находятся на своих местах.

Есть и другой способ реализации команды ind. Добавив в начало команды восклицательный знак !, можно обрабатывать только те строки, которые *не* соответствуют шаблону:

```
sed '/^$/!s/^/→/' $*      ind, версия 3
```

Шаблон /^$/ определяет пустые строки (конец строки следует сразу за ее началом), таким образом /^$/! означает «не выполнять команду на пустых строках».

Как говорилось выше, sed автоматически выводит строки независимо от того, были они обработаны или нет (если только они не были удалены). Кроме того, можно использовать большинство команд редактора ed. Поэтому не представляет труда написать программу, которая печатает первые три (например) прочитанные строки и завершает работу:

```
sed 3q
```

Хотя 3q и не является командой ed, она имеет смысл в sed: копировать строки три раза, затем выйти.

Может потребоваться дополнительная обработка данных, например выравнивание. Для этой цели можно было бы перенаправить вывод sed на вход ind, но, поскольку sed способен выполнять несколько команд одновременно, всю работу можно выполнить одним вызовом sed:

```
sed 's/^/→/
    3q'
```

Обратите внимание на то, как расположены кавычки и символ новой строки, каждая команда должна находиться на отдельной строке, при этом sed игнорирует начальные пробелы и знаки табуляции.

Может возникнуть желание написать программу – назовем ее head – для печати первых нескольких строк файлов. Но набрать на клавиатуре sed 3q (или 10q) настолько просто, что, скорее всего, такую программу писать нет смысла. Тем не менее, мы в своей системе применяем ind, так как соответствующая ему команда sed несколько сложнее. (В процессе написания мы заменили 30-строчную программу на C однострочной командой версии 2, рассмотренной выше.) Нет точного критерия целесообразности замены сложной командной строки одной командой; вот лучшее из правил, которые нам удалось выработать, – поместите новую команду в свой каталог /bin и посмотрите, как часто вы будете ее использовать.

Можно также поместить команды sed в файл и вызывать их оттуда строкой

```
$ sed -f cmdfile ...
```

Выбирать строки можно не только по их номерам, в частности команда

```
$ sed '/шаблон/q'
```

напечатает строки от начала файла до первой строки, соответствующей *шаблону*, включая и эту строку, а команда

```
$ sed '/шаблон/d'
```

удаляет все строки, соответствующие *шаблону*; удаление выполняется до того, как данные попадают на печать, поэтому удаленные строки просто исчезают.

Автоматический вывод результатов, вполне пригодный в большинстве случаев, иногда бывает нежелателен. Он может быть выключен параметром `-n`; в этом случае выводятся только строки, явно отправленные на печать командой `p`. Например,

```
$ sed -n '/шаблон/p'
```

делает то же самое, что и `grep`. А поскольку условие совпадения может быть инвертировано, то команда

```
$ sed -n '/шаблон/!p'
```

эквивалентна `grep -v`. (Так же как и `sed '/`*шаблон*`/d'`.)

Зачем использовать обе команды – `sed` и `grep`? В конце концов, `grep` – всего лишь простой частный случай `sed`. Частично это объясняется историческими причинами – команда `grep` появилась намного раньше, чем `sed`. Но `grep` живет и процветает благодаря своей лаконичности: с ее помощью проще выполнить ту работу, которую обе команды выполняют одинаково. (Команда `grep` имеет функции, отсутствующие в `sed`, например, определяемые параметром `-b`.) Хотя бывает, что программы умирают. Была когда-то программа `gres`, выполнявшая простые подстановки, исчезнувшая почти сразу после появления `sed`.

Символы новой строки могут быть добавлены с помощью `sed` аналогично тому, как это делалось в `ed`:

```
$ sed 's/$/\
> /'
```

Эта команда добавляет ко всем строкам символ новой строки, увеличивая вдвое их количество.

```
$ sed 's/[ →][ →]*/\
> /g'
```

А эта заменяет последовательности пробелов и символов табуляции одним символом разделителя строк, помещая каждое слово на отдельной строке. (Регулярное выражение `'[ →]'` соответствует пробелу или знаку табуляции, а `'[ →]*'` нулю или больше тех же символов, так что выражение в целом определяет один или несколько пробелов и/или знаков табуляции.)

Можно также использовать пары регулярных выражений или номеров строк для указания *диапазона* строк, над которыми будет выполняться любая из команд.

```
$ sed -n '20,30p'      Вывести строки с 20 по 30
$ sed '1,10d'          Удалить строки с 1 по 10 (= tail +11)
$ sed '1,/^$/d'        Удалить до первой пустой строки включительно
$ sed -n '/^$/,/^end/p'  Вывести все группы строк, начинающиеся
                         с пустой строки и заканчивающиеся строкой,
                         начинающейся словом end
$ sed '$d'             Удалить последнюю строку
```

Строки нумеруются последовательно от начала до конца входного потока данных, независимо от количества файлов.

Отсутствие относительной нумерации строк является фундаментальным ограничением программы `sed`, отсутствующим в редакторе `ed`. В частности, знаки + и – не обрабатываются в выражениях для номеров строк, поэтому невозможно вернуться назад во входном потоке:

```
$ sed '$-1d'              Недопустимо: ссылки назад запрещены
Unrecognized command: $-1d
$
```

После того как очередная строка прочитана, предыдущая исчезает навсегда: нет способа обратиться к строке, «соседней» с текущей, как требует эта команда. (Если быть честными до конца, нужно сказать, что такая возможность имеется, но она требует углубленного изучения команды `hold`.) Относительная адресация вперед также недоступна:

```
$ sed '/thing/+1d'        Недопустимо: ссылки вперед запрещены
```

Зато есть возможность выводить данные в несколько файлов. Например,

```
$ sed -n '/pat/w file1
>             /pat/!w file2' имена-файлов  …
$
```

выводит строки, соответствующие шаблону *pat* в `file1`, а остальные – в `file2`. Или, возвращаясь к первому примеру,

```
$ sed 's/UNIX/UNIX(TM)/gw u.out' имена-файлов ... >output
```

Такая команда, как и раньше, выведет все строки в файл `output`, но, кроме этого, создаст файл `u.out` для измененных строк.

Иногда требуется передать имена файлов, заданные параметрами в оболочке, в тело команды `sed`. Примером такого взаимодействия является программа `newer`, которая выводит имена файлов, созданных позже указанного файла.

```
$ cat newer
# newer f:  вывести имена файлов, созданных после f
ls -t | sed '/^'$1'$/q'
$
```

Кавычки предотвращают интерпретацию специальных символов оболочкой, а $1 заменяется именем файла. Другой способ передачи аргумента:

```
"/^$1\$/q"
```

Здесь вместо $1 будет подставлено значение, а \$ превратится в $.

Аналогично выглядит программа older, которая выводит имена файлов, созданных раньше указанного:

```
$ cat older
# older f:  вывести имена файлов, созданных раньше f
ls -tr | sed '/^'$1'$/q'
$
```

Единственное отличие заключается в параметре -r команды ls, располагающем список в обратном порядке.

Несмотря на то что программа sed имеет гораздо больше возможностей, чем здесь было показано (включая проверку условий, циклы и переходы, запоминание предыдущих строк), и, разумеется, выполняет большинство команд редактора ed, который описан в приложении 1, наиболее часто она применяется именно так, как здесь было описано – с одной-двумя командами редактирования, без использования длинных и сложных выражений. Основные возможности sed описаны в табл. 4.2, хотя там отсутствуют многострочные функции.

Таблица 4.2. Команды программы sed

Команда	Действие
a\	дописывать в выходной файл строки, заканчивающиеся знаком \
b *label*	переход к метке *label*
c\	заменить строки указанным текстом, в котором строки заканчиваются знаком \
d	удалить строку и прочитать следующую из входного потока
i\	вставить текст перед продолжением вывода
l	вывести строку с отображением непечатных символов
p	вывести строку
q	выйти
r *file*	читать файл *file*, копировать содержимое в выходной поток
s /*old*/*new*/*f*	заменить фрагмент *old* фрагментом *new*. При *f*=g заменить все, при *f*=p печатать, при *f*=w *file* писать в *file*

Таблица 4.2 (продолжение)

Команда	Действие
`t` *label*	проверка: перейти к метке *label*, если в текущей строке выполнена подстановка
`w` *file*	вывести строку в файл *file*
`y` /*str1*/*str2*/	заменить все символы строки *str1* соответствующими символами строки *str2* (диапазоны недопустимы)
`=`	вывести номер текущей строки
`!`*cmd*	выполнить команду *cmd*, только если строка не выбрана
`:` *label*	установить метку *label* для команд `b` и `t`
{	сгруппировать команды до следующего символа }

Программа `sed` удобна тем, что может обрабатывать файлы произвольной длины, работает быстро и имеет много общего с редактором `ed`, в том числе регулярные выражения и построчную обработку. С другой стороны, к ее недостаткам можно отнести ограниченную возможность запоминания (тяжело переносить текст из одной строки в другую), обработку данных только за один проход, невозможность перемещения назад, отсутствие ссылок вперед (`/.../+1`) и отсутствие средств для вычислений – это всего лишь текстовый редактор.

Упражнение 4.5. Измените программы `older` и `newer` так, чтобы они не выводили имя переданного им файла. Измените порядок вывода на обратный. □

Упражнение 4.6. Используйте `sed`, чтобы повысить надежность программы `bundle`. Подсказка: при использовании встроенных документов маркер конца распознается только при точном соответствии строке. □

4.4. Язык сканирования и обработки шаблонов awk

Некоторые ограничения `sed` устранены в программе `awk`. В ее основу положена та же идея, что и в `sed`, но реализация ближе к языку Си, чем к текстовому редактору. Формат вызова программы аналогичен `sed`:

```
$ awk 'program' имена-файлов …
```

но аргумент *program* имеет другое значение:

```
шаблон { действие }
шаблон { действие }
…
```

Программа `awk` читает входные файлы *имена-файлов* построчно. Каждая строка проверяется на соответствие с каждым *шаблоном*; при на-

личии соответствия выполняются *действия*. Как и sed, программа awk не изменяет входные файлы.

В качестве шаблонов могут выступать регулярные выражения, те же, что и в egrep, или более сложные условные выражения, напоминающие язык Си. Вот простой пример:

```
$ awk '/регулярное выражение/ { print }' имена-файлов ...
```

работает аналогично программе egrep: выводит все строки, определяемые *регулярным выражением*.

Необязательно одновременно указывать и шаблон, и действие. Если отсутствует действие, то выполняется действие по умолчанию – вывод выбранных шаблоном строк, то есть команда

```
$ awk '/регулярное выражение /' имена-файлов ...
```

выполняется аналогично предыдущей. Если же не задан шаблон, то действие выполняется над *каждой* входной строкой, и команда

```
$ awk '{ print }' имена-файлов …
```

делает то же, что и cat, только намного медленнее.

Прежде чем перейти к более интересным примерам, сделаем еще одно замечание. Команде awk, как и sed, можно передать файл с программой:

```
$ awk -f cmdfile имена-файлов ...
```

Поля

Программа awk автоматически делит каждую прочитанную строку на *поля* – строки символов (не пробелов), разделенные пробелами или знаками табуляции. Согласно этому определению строки, выводимые командой who, содержат пять полей:

```
$ who
you     tty2     Sep 29 11:53
jim     tty4     Sep 29 11:27
$
```

Этим полям awk присваивает имена $1, $2, ..., $NF, где NF – переменная, содержащая количество полей в строке. В приведенном примере NF=5 для обеих строк. (Обратите внимание на то, что NF – это количество полей, а $NF – последнее поле в строке. В отличие от оболочки, переменные awk не имеют префикса $, за исключением имен полей.) Например, чтобы не выводить размер файла в команде du -a, напишем

```
$ du -a | awk '{ print $2 }'
```

а чтобы вывести только имена пользователей и время их регистрации (по одному на строке):

```
$ who | awk '{ print $1, $5 }'
```

```
you 11:53
jim 11:27
$
```

а вот те же данные, отсортированные по времени регистрации:

```
$ who | awk '{ print $5, $1 }' | sort
11:27 jim
11:53 you
$
```

Ранее в этой главе уже были рассмотрены варианты команды sed, выполняющие те же действия. Несмотря на то что для подобных целей программа awk удобнее, чем sed, она работает медленнее как при запуске, так и в процессе выполнения, особенно на больших объемах данных.

По умолчанию awk считает, что поля разделены пробелами и знаками табуляции, но позволяет определить в качестве разделителя любой символ. Это можно сделать с помощью параметра -F (верхний регистр). Например, поля файла паролей /etc/passwd разделены двоеточиями:

```
$ sed 3q /etc/passwd
root:3D.fHR5KoB.3s:0:1:S.User:/:
ken:y.68wd1.ijayz:6:1:K.Thompson:/usr/ken:
dmr:z4u3dJWbg7wCk:7:1:D.M.Ritchie:/usr/dmr:
$
```

Для того чтобы напечатать имена пользователей, расположенные в первом поле, выполним команду

```
$ sed 3q /etc/passwd | awk -F '{ print $1 }'
root
ken
dmr
$
```

Существуют некоторые особенности использования пробелов и знаков табуляции. По умолчанию awk считает их разделителями и игнорирует, если они находятся в начале строки. Однако если разделителем назначен любой другой символ, кроме пробела, то начальные пробелы становятся частью поля. В частности, если разделителем назначен знак табуляции, то пробел разделителем уже не считается, и начальные пробелы попадут в первое поле.

Печать

Кроме подсчета полей на входе awk ведет и другую интересную статистику. Встроенная переменная NR хранит порядковый номер текущей входной строки. Для вывода номеров строк используйте команду

```
$ awk '{ print NR, $0 }'
```

Поле $0 представляет собой всю входную строку целиком. В операторе print значения, разделяемые запятыми, печатаются через разделитель выходных полей, по умолчанию это пробел.

Если возможностей форматирования оператора print недостаточно, то можно обратиться к printf. Так, например, можно напечатать номера строк в поле длиной четыре символа:

```
$ awk '{ printf "%4d %s\n", NR, $0 }'
```

Здесь спецификатор формата %4d задает отображение десятичного целого (NR) в четырехсимвольном поле, а спецификатор %s – отображение строки символов ($0), добавлен также символ новой строки \n, поскольку printf самостоятельно его не добавляет. Оператор printf программы awk аналогичен функции языка Си (см. printf(3)).

Можно переписать программу ind (рассмотренную ранее в этой главе) следующим образом:

```
$ awk '{ printf "\t%s\n", $0 }' $*
```

Здесь последовательно выводятся знак табуляции (\t) и входная строка.

Шаблоны

Предположим, что потребовалось найти в файле /etc/passwd пользователей, у которых нет пароля. Зашифрованный пароль хранится во втором поле, поэтому программа состоит из одного шаблона:

```
$ awk -F: '$2 == ""' /etc/passwd
```

Шаблон проверяет, является ли второе поле пустой строкой (с помощью оператора равенства ==). Есть и другие варианты написания такого шаблона:

```
$2 == ""              Второе поле пустое
$2 ~ /^$/             Второе поле совпадает с пустой строкой
$2 !~ /./             Второе поле не соответствует никакому символу
length($2) == 0       Длина второго поля равна нулю
```

Символ ~ означает соответствие регулярному выражению, а !~ – отсутствие соответствия. Собственно регулярное выражение заключено между символами косой черты.

Длину строки символов выдает length – встроенная функция awk. Символ ! в начале шаблона означает отрицание, например

```
!($2 == "")
```

Оператор ! подобен оператору языка Си, но не оператору sed (в sed ! следует за шаблоном).

Часто шаблоны применяются в awk для несложной проверки корректности данных. Обычно выполняется проверка несоответствия строк

некоторому критерию («отсутствие новостей – хорошая новость»). Например, следующий шаблон проверяет при помощи оператора вычисления остатка %, содержит ли входная строка четное число полей:

```
NF % 2 != 0          # печатать, если число полей нечетно
```

А в следующем примере, с функцией length, печатаются строки, длина которых превышает заданную:

```
length($0) > 72      # печатать, если строка слишком длинная
```

В программе awk действует то же соглашение о комментариях, что и в оболочке: комментарий начинается с символа #.

Чтобы сделать вывод более информативным, можно вывести на печать предупреждающее сообщение и фрагмент слишком длинной строки. Здесь использована еще одна встроенная функция – substr:

```
length($0) > 72 { print "Строка", NR, "слишком длинна: ", substr($0, 1, 60)}
```

Функция substr(*s*, *m*, *n*) возвращает подстроку строки *s* начиная с позиции *m* длиной *n* символов. (Строка начинается с позиции 1.) Если значение *n* не указано, то вывод продолжается до конца строки. Функцию substr можно также применять для извлечения полей, расположенных в фиксированной позиции, например часов и минут в выводе команды date:

```
$ date
Thu Sep 29 12:17:01 EDT 1983
$ date | awk '{ print substr($4, 1, 5) }'
12:17
$
```

Упражнение 4.7. Сколькими способами вы могли бы имитировать команду cat с помощью awk? Какой вариант самый короткий? □

Шаблоны BEGIN и END

В awk существуют два специальных шаблона, BEGIN и END. BEGIN позволяет определить действия, выполняемые до того, как будет прочитана первая строка ввода; этот шаблон можно использовать для инициализации переменных, печати заголовков или для переопределения разделителя полей присваиванием значения переменной FS:

```
$ awk 'BEGIN { FS = ":" }
>      $2 == "" ' /etc/passwd
$                Нет вывода: у всех пользователей есть пароль
```

Шаблон END определяет действия, выполняемые после того, как будет обработана последняя строка:

```
$ awk 'END { print NR }' ...
```

напечатает число входных строк.

Арифметические выражения и переменные

До сих пор рассматривались только примеры несложного преобразования строк. Настоящая же мощь awk заключается в способности выполнять вычисления на основе входных данных – подсчитывать объекты, вычислять суммы и средние значения и т. п. Часто awk применяется для суммирования столбцов чисел. Так, например, выполняется сложение всех чисел из первого столбца:

```
      { s = s + $1 }
END   { print s }
```

А поскольку в переменной NR хранится количество слагаемых, то выражение вида

```
END   { print s, s/NR }
```

выведет сумму и среднее значение.

Этот пример демонстрирует также работу с переменными в awk. Переменная s не является встроенной, она определена в момент первого использования. По умолчанию переменные инициализируются нулем, поэтому явная инициализация обычно не требуется.

В awk, как и в Си, допустима сокращенная запись арифметических выражений, поэтому в приведенном примере можно использовать и такую запись:

```
      { s += $1 }
END   { print s }
```

Выражение s += $1 эквивалентно s = s + $1, но более компактно.

Обобщая приведенный выше пример, получим:

```
      { nc += length($0) + 1  # счетчик символов, 1 для \n
        nw += NF              # счетчик слов
      }
END   { print NR, nw, nc }
```

Здесь подсчитывается количество строк, слов и символов на входе, аналогично тому, как это делается в программе wc, но без разделения результатов по файлам.

В качестве другого примера применения арифметических выражений приведем программу prpages, вычисляющую количество 66-строчных страниц, полученных в результате обработки набора файлов программой pr:

```
$ cat prpages
# prpages: вычисляет количество страниц, которое будет напечатано программой pr
wc $* |
awk '!/ total$/ { n += int(($1+55) / 56) }
     END        { print n }'
$
```

Команда pr выводит по 56 строк на страницу[1] (определено эмпирическим путем). В каждой строке, выводимой командой wc (если в ней не содержится слово total), число страниц округляется в большую сторону и приводится встроенной функцией int к целому типу отбрасыванием дробной части.

```
$ wc ch4.*
    753   3090  18129 ch4.1
    612   2421  13242 ch4.2
    637   2462  13455 ch4.3
    802   2986  16904 ch4.4
     50    213   1117 ch4.9
   2854  11172  62847 total
$ prpages ch4.*
53
$
```

Для проверки результата направим в awk вывод команды pr:

```
$ pr ch4.* | awk 'END { print NR/66 }'
53
$
```

Переменные awk могут хранить и символьные строки. То, как будет трактоваться переменная – как символьная строка или как число, зависит от контекста. Грубо говоря, в арифметических выражениях вида s+=$1 подразумевается числовое значение, а в строковых выражениях типа x="abc" – строковое значение. В неясных ситуациях, вроде x>y, операнды считаются строковыми, если не очевидно обратное. (Точные правила приведены в руководстве по программе awk.) Строковые переменные инициализируются пустой строкой. В следующих разделах будут приведены примеры работы со строками.

Программа awk содержит ряд встроенных переменных обоих типов, в частности NR и FS. В табл. 4.3 приведен их полный список, а в табл. 4.4 – перечень операторов.

Таблица 4.3. Встроенные переменные awk

Переменная	Значение
FILENAME	имя текущего входного файла
FS	разделитель полей (пробел и табуляция по умолчанию)
NF	количество полей во входной записи
NR	номер вводимой записи
OFMT	формат вывода для чисел (%g по умолчанию, см. printf(3))

[1] Команда pr добавляет на каждую страницу по пять строк на шапку и подвал, поэтому делим на 56, а не на 66. – *Примеч. перев.*

Переменная	Значение
OFS	разделитель полей выходных строк (пробел по умолчанию)
ORS	разделитель выходных строк (символ новой строки по умолчанию)
RS	разделитель входных записей (символ новой строки по умолчанию)

Таблица 4.4. Операторы awk (в порядке повышения приоритета)

Оператор	Действие
= += -= *= /= %=	присваивание; v *op* = *expr* эквивалентно v = v *op (expr*)
||	логическое ИЛИ: *expr1* || *expr2* истинно, если истинен любой из операндов
&&	логическое И: *expr1* || *expr2* истинно, если истинны оба операнда
!	инвертирование результата выражения
> >= < <= == != ~ !~	операторы сравнения; ~ – совпадение, !~ – несовпадение
ничего	конкатенация строк
+ -	плюс и минус
* / %	умножение, деление, вычисление остатка
++ --	инкремент и декремент (префиксная и постфиксная формы)

Упражнение 4.8. Способ тестирования prpages наводит на мысль об альтернативной реализации. Определите наиболее быструю путем эксперимента. □

Операторы управления

Очень распространенной ошибкой (знаем по собственному опыту) при редактировании больших документов является дублирование слов, очевидно, что почти всегда это происходит непреднамеренно. Для исправления таких ошибок в составе семейства программ Writer's Workbench существует программа double, отыскивающая пары идентичных смежных слов. Вот реализация double на основе awk:

```
$ cat double
awk '
FILENAME != prevfile {    # новый файл
    NR = 1                # переустановить счетчик строк
    prevfile = FILENAME
}
NF > 0 {
    if ($1 == lastword)
        printf "double %s, file %s, line %d\n", $1, FILENAME, NR
```

```
    for (i = 2; i <= NF; i++)
        if ($i == $(i-1))
            printf "double %s, file %s, line %d\n",$i,FILENAME,NR
    lastword = $NF
}' $*
$
```

Оператор ++ увеличивает значение операнда на единицу, а оператор -- соответственно уменьшает. Встроенная переменная FILENAME содержит имя текущего входного файла. Поскольку NR подсчитывает строки с начала входного потока, она переустанавливается в начале каждого файла с тем, чтобы облегчить поиск строки с ошибкой.

Условный оператор if имеет тот же вид, что и в языке Си.

```
if (условие)
        оператор1
else
        оператор2
```

Если *значение условия* истинно, то выполняется *оператор1*, если же его значение ложно и присутствует необязательная конструкция else, то выполняется *оператор2*.

Оператор цикла for аналогичен одноименному оператору Си, но отличается от используемого в оболочке:

```
for (выражение1; условие; выражение2)
      оператор
```

Оператор for идентичен следующему оператору while, допустимому в awk:

```
выражение1
     while (условие) {
            оператор
            выражение2
     }
```

Например,

```
for (i = 2; i <= NF; i++)
```

выполняет тело цикла, устанавливая значение переменной i равным 2, 3, ..., вплоть до количества полей NF.

Оператор break вызывает передачу управления за пределы ближайшего цикла while или for; оператор continue вызывает переход к следующей итерации (в точку *условие* в цикле while и в точку *выражение2* в цикле for). Оператор next инициирует ввод следующей строки и переход к первому шаблону программы для ее обработки. Оператор exit немедленно передает управление шаблону END.

Массивы

Как и большинство языков программирования, awk имеет средства работы с массивами. В этом простом примере программа заполняет массив строками входного файла, используя их номера в качестве индекса, затем печатает строки в обратном порядке:

```
$ cat backwards
# backwards: построчная печать файла в обратном порядке
awk '    { line[NR] = $0 }
END      { for (i = NR; i > 0; i--) print line[i] } ' $*
$
```

Обратите внимание, что массивы, как и переменные, не требуют объявления; размер массива ограничен лишь объемом оперативной памяти. Очевидно, что, загружая в массив достаточно большой файл, можно исчерпать доступную память. Для того чтобы распечатать конец большого файла, воспользуемся командой tail:

```
$ tail -5 /usr/dict/web2 | backwards
zymurgy
zymotically
zymotic
zymosthenic
zymosis
$
```

Преимущество команды tail заключается в том, что перемещение по файлу она организует средствами файловой системы, что позволяет избежать чтения промежуточных данных. Функция lseek рассмотрена в главе 7. (В системе, используемой авторами, команда tail имеет параметр -r, инвертирующий порядок вывода строк, что позволяет отказаться от применения backwards.)

Обычно при вводе каждая строка разделяется на поля. С помощью встроенной функции split аналогичная операция выполняется над любой строкой:

n = split(*s*, *arr*, *sep*)

Эта функция разбивает строку *s* на отдельные поля, сохраняемые в *n* элементах массива *arr*. В качестве разделителя полей выступает либо символ, заданный параметром *sep*, либо, при его отсутствии, – текущее значение переменной FS. Например, выражение split($0, a, ":") выделит поля, разделяемые двоеточиями, что можно использовать при обработке файла /etc/passwd, а split("9/29/83", date, "/") разделит строку с датой.

```
$ sed 1q /etc/passwd | awk '{split($0,a,":"); print a[1]}'
root
$ echo 9/29/83 | awk '{split($0,date,"/"); print date[3]}'
83
$
```

В табл. 4.5 перечислены встроенные функции awk.

Таблица 4.5. Встроенные функции awk

Функция	Возвращаемое значение
cos(*expr*)	косинус *expr*
exp(*expr*)	экспоненциальная функция
getline()	ввод очередной строки, возвращает 0, если достигнут конец файла, иначе 1
index(*s1*,*s2*)	положение строки *s2* в строке *s1*, возвращает 0, если подстрока не найдена
int(*expr*)	целая часть *expr*, дробная часть отбрасывается
length(*s*)	длина строки *s*
log(*expr*)	натуральный логарифм *expr*
sin(*expr*)	синус *expr*
split(*s*,*a*,*c*)	разбиение строки *s* на элементы *a*[1]...*a*[n] по символу *c*, возвращает *n*
sprintf(*fmt*, ...)	форматировать ... согласно спецификации *fmt*
substr(*s*,*m*,*n*)	подстрока строки *s* длиной *n*, начиная с позиции *m*

Ассоциативные массивы

При обработке данных часто возникает необходимость хранения наборов пар имя–значение. Например, при вводе таких данных

```
Susie   400
John    100
Mary    200
Mary    300
John    100
Susie   100
Mary    100
```

требуется подсчитать сумму для каждого имени:

```
John    200
Mary    600
Susie   500
```

Изящным решением для подобного рода задач являются ассоциативные массивы. Обычно в массивах индексы представлены целыми числами, но awk позволяет использовать в этом качестве любое значение. Поэтому

```
        { sum[$1] += $2 }
END     { for (name in sum) print name, sum[name] }
```

представляет собой законченную программу подсчета и печати сумм для пар имя–значение из предыдущего примера, не требующую предварительной сортировки. Имена выполняют функцию индексов массива sum, а в конце оператор цикла for специального вида выполняет проход по массиву, печатая значения его элементов. Синтаксис этого варианта оператора for имеет вид:

```
for (переменная in массив)
        оператор
```

Несмотря на внешнее сходство с оператором цикла оболочки, имеется существенное отличие. Выполняется проход по индексам *массива*, а не по его элементам, *переменной* поочередно присваиваются значения индексов. Индексы могут следовать в произвольном порядке, поэтому может потребоваться сортировка. В вышеприведенном примере вывод может быть перенаправлен в программу sort с тем, чтобы имена с наибольшими суммами печатались первыми.

```
$ awk '...' | sort +1nr
```

В реализации ассоциативной памяти применяются схемы хеширования, обеспечивающие примерно равное время доступа к элементам и независимость этого времени от размеров массива (по крайней мере, для средних размеров).

Ассоциативная память эффективна в таких задачах, как подсчет количества вхождений слов во входном файле:

```
$ cat wordfreq
awk '    { for (i = 1; i <= NF; i++) num[$i]++ }
END      { for (word in num) print word, num[word] }
' $*
$ wordfreq ch4.* | sort +1 -nr | sed 20q | 4
the 372          .CW 345          of 220           is 185
to 175           a 167            in 109           and 100
.P1 94           .P2 94           .PP 90           $ 87
awk 87           sed 83           that 76          for 75
The 63           are 61           line 55          print 52
$
```

В первом цикле for выполняется просмотр каждой входной строки с увеличением на единицу элементов массива, индексированных выбранным словом. (Не путайте переменную $i, используемую awk, представляющую i-е поле строки с переменными оболочки.) После того как файл прочитан, следующий цикл печатает в произвольном порядке найденные слова и их счетчики.

Упражнение 4.9. В выводе программы wordfreq встречаются команды форматирования, такие, например, как .CW, устанавливающая определенный шрифт. Как избавиться от таких «ненастоящих» слов? Как с помощью команды tr сделать wordfreq независимой от регистра? Срав-

ните реализацию и производительность wordfreq, конвейера команд из раздела 4.2 и такого решения:

```
sed 's/[ →][ →]*/\
/g' $* | sort | uniq -c | sort -nr
```

□

Строки

Несмотря на то что для таких небольших задач, как выборка отдельного поля, пригодны обе программы – и sed и awk, в тех случаях, когда требуется сколько-нибудь значительное программирование, применяется только awk. В качестве примера приведем программу, «сворачивающую» длинные строки до 80 символов. Каждая строка, превышающая этот размер, прерывается после 80-го символа, в качестве предупреждения добавляется символ \, затем обрабатывается остаток строки. Последний фрагмент «свернутой» строки выравнивается вправо – листинги программ, для которых мы обычно используем fold, выглядят при этом более наглядно. Вот как выглядят строки, свернутые до 20 символов вместо 80:

```
$ cat test
A short line.
A somewhat longer line.
This line is quite a bit longer than the last one.
$ fold test
A short line.
A somewhat longer li\
                  ne.
This line is quite a\
 bit longer than the\
           last one.
$
```

Довольно странно, что в седьмой версии отсутствует программа для добавления и удаления знаков табуляции, хотя в System V программа pr выполняет и то и другое. В нашей реализации fold преобразование табуляций в пробелы осуществляется при помощи sed, чтобы счетчик символов awk работал правильно. Дело в том, что начальные пробелы (типичные для текстов программ) обрабатываются правильно, но нарушается расположение столбцов, разделенных табуляциями в середине строки.

```
# fold:  сворачивает длинные строки
sed 's/\(→/    /g' $* |     # преобразовать табуляции в пробелы
awk '
BEGIN {
      N = 80          # сворачивать на 80-й позиции
      for (i = 1; i <= N; i++)     # заполнить строку пробелами
```

```
            blanks = blanks " "
}
{     if ((n = length($0)) <= N)
            print
      else {
            for (i = 1; n > N; n -= N) {
                  printf "%s\\\n", substr($0,i,N)
                  i += N;
            }
            printf "%s%s\n", substr(blanks,1,N-n), substr($0,i)
    }
} '
```

В awk нет оператора конкатенации строк; строки объединяются, если они расположены рядом. Строка blanks инициализирована пустым значением. Цикл в секции BEGIN заполняет строку пробелами, используя конкатенацию: в каждом проходе к строке пробелов добавляется еще один. Следующий цикл делит строку на части, пока остаток не станет достаточно коротким. Как и в Си, оператор присваивания может быть использован в выражении, поэтому конструкция

```
if ((n = length($0)) <= N) ...
```

присваивает значение длины входной строки переменной n до выполнения проверки. Не забывайте о скобках.

Упражнение 4.10. Измените программу fold так, чтобы она сворачивала строки по пробелам и табуляциям, не разбивая слова. Предусмотрите обработку длинных слов. □

Взаимодействие с оболочкой

Предположим, требуется написать программу field *n*, которая должна выводить *n*-е поле каждой входной строки так, чтобы, например, команда

```
$ who | field 1
```

выводила только регистрационное имя. Выбор поля в awk осуществляется элементарно, проблема в том, как передать ей номер поля *n*. Вот пример:

```
awk '{ print $'$1' }'
```

Переменная $1 доступна оболочке (она не взята в кавычки), благодаря чему awk получает номер поля. Другой способ заключается в использовании двойных кавычек:

```
awk "{ print \$$1 }"
```

В этом случае аргумент интерпретируется оболочкой, которая заменяет \$ на $, а $1 – на значение *n*. Авторы отдают предпочтение первому

способу, так как двойные кавычки в типичной программе awk потребуют слишком много знаков \.

Другой пример – программа addup *n*, суммирующая числа из *n*-го поля:

```
awk '{ s += $'$1' }
 END { print s }'
```

В следующем примере подсчитываются отдельные суммы для *n* столбцов и общий итог:

```
awk '
BEGIN { n = '$1' }
{     for (i = 1; i <= n; i++)
            sum[i] += $i
}
END {for (i = 1; i <= n; i++) {
            printf "%6g ", sum[i]
            total += sum[i]
      }
      printf "; total = %6g\n", total
} '
```

Чтобы избежать обилия кавычек, для присваивания переменной значения *n* использована конструкция BEGIN.

Основной недостаток всех этих примеров не в том, что приходится следить за тем, находятся ли переменные внутри кавычек или снаружи (хотя и это довольно утомительно), а в том, что программы способны читать только данные из стандартного ввода: нет никакого способа передать им одновременно параметр *n* и список файлов произвольной длины. Чтобы справиться с этой проблемой, нам понадобится программировать в оболочке, чем мы и займемся в следующей главе.

Календарь на основе awk

Последний пример иллюстрирует применение ассоциативных массивов и взаимодействие с оболочкой и дает представление о постепенном развитии программы за счет проводимых усовершенствований.

Задачей является создание программы, которая каждое утро будет отправлять письмо с напоминанием о предстоящих событиях. (Возможно, такая служба уже имеется в системе, см. calendar(1). Здесь представлен альтернативный подход.) В базовом варианте доставляются сообщения о событиях текущего дня; следующим шагом станет возможность предварительного уведомления о завтрашних делах. Планирование с учетом праздников и выходных дней оставим читателям в качестве упражнения.

Первым делом определим место для хранения календаря. Поступим просто – назовем файл calendar и поместим его в каталог /usr/you.

```
$ cat calendar
Sep 30  Мамин день рождения
Oct 1   Днем обедаем с Джо
Oct 1   Совещание в 16.00
$
```

Во-вторых, выберем способ просмотра календаря в поисках даты. Доступно множество вариантов, остановимся на программе awk, так как она наиболее удобна для арифметических операций с датами, хотя можно было бы воспользоваться и другими программами, такими как egrep и sed. Разумеется, записи, выбранные из календаря, будут доставляться по почте.

В-третьих, необходим надежный способ ежедневного автоматического сканирования календаря, желательно рано утром. Это можно выполнить с помощью команды at, которая уже упоминалась в главе 1.

Если определить формат файла calendar таким образом, чтобы каждая строка начиналась с названия месяца и дня (подобно тому, как их выводит date), то первая версия программы получится достаточно простой:

```
$ date
Thu Sep 29 15:23:12 EDT 1983
$ cat bin/calendar
# calendar:  версия 1 - только для сегодняшнего дня
awk <$HOME/calendar '
      BEGIN { split("`date`", date) }
      $1 == date[2] && $2 == date[3]
' | mail $NAME
$
```

Блок BEGIN помещает текущую дату, полученную от date, в массив, второй и третий элементы которого – это месяц и день. Предполагается, что переменная оболочки NAME содержит регистрационное имя пользователя.

Примечательная последовательность кавычек необходима для того, чтобы поместить строку даты в кавычки в теле программы awk. Более простым для понимания способом является вставка строки с датой в начало входного потока:

```
$ cat bin/calendar
# calendar:  версия 2 - только сегодня, без кавычек
(date; cat $HOME/calendar) |
awk '
  NR == 1    { mon = $2; day = $3 } # установить дату
  NR > 1 && $1 == mon && $2 == day  # вывести построчно
' | mail $NAME
$
```

Теперь изменим программу так, чтобы просматривать не только сегодняшние, но и завтрашние записи. В течение месяца достаточно будет

прибавлять единицу к текущей дате, но в последний день придется выбирать следующий месяц и устанавливать число в 1, не забывая при этом, что в разных месяцах разное количество дней.

Здесь-то и пригодятся ассоциативные массивы. В двух массивах – days и nextmon, – индексами в которых служат названия месяцев, будем хранить число дней в месяце и название следующего месяца. Тогда значение days["Jan"] равно 31, а значение nextmon["Jan"] – Feb. Вместо того чтобы писать последовательность операторов вида

```
days["Jan"] = 31; nextmon["Jan"] = "Feb"
days["Feb"] = 28; nextmon["Feb"] = "Mar"
...
```

воспользуемся функцией split для преобразования символьной строки в нужные структуры данных:

```
$ cat calendar
# calendar:  версия 3 - сегодня и завтра
awk <$HOME/calendar '
BEGIN {
    x = "Jan 31 Feb 28 Mar 31 Apr 30 May 31 Jun 30 " \
        "Jul 31 Aug 31 Sep 30 Oct 31 Nov 30 Dec 31 Jan 31"
    split(x, data)
    for (i = 1; i < 24; i += 2) {
        days[data[i]] = data[i+1]
        nextmon[data[i]] = data[i+2]
    }
    split("'"`date`"'", date)
    mon1 = date[2]; day1 = date[3]
    mon2 = mon1; day2 = day1 + 1
    if (day1 >= days[mon1]) {
        day2 = 1
        mon2 = nextmon[mon1]
    }
}
$1 == mon1 && $2 == day1 || $1 == mon2 && $2 == day2
' | mail $NAME
$
```

Обратите внимание, что январь (Jan) присутствует в списке дважды; такая избыточность упрощает обработку для декабря.

И наконец, научим программу calendar запускаться ежедневно. Для этого всего лишь требуется, чтобы кто-нибудь просыпался в пять утра и запускал программу. Можно делать это вручную, если не забывать (каждый день!) выполнять команду

```
$ at 5am
calendar
ctl-d
$
```

но это не очень надежный способ. Хитрость в том, чтобы заставить команду at не только запускать календарь, но и перезапускать саму себя.

```
$ cat early.morning
calendar
echo early.morning | at 5am
$
```

Второй строкой запускается команда at на следующий день, поэтому, будучи однажды запущена, такая последовательность команд бесконечно повторяет сама себя. Команда at сама устанавливает переменную PATH, текущий каталог и другие параметры, так что дополнительных действий не требуется.

Упражнение 4.11. Измените программу calendar так, чтобы она «знала» о выходных днях: в пятницу «завтра» включает в себя субботу, воскресенье и понедельник. Добавьте поддержку високосных годов. Нужна ли поддержка праздников? Как ее реализовать? □

Упражнение 4.12. Надо ли обрабатывать даты в середине строки, а не только в начале? И как насчет дат, записанных в других форматах, например 10/1/83? □

Упражнение 4.13. Почему программа calendar использует вызов getname, а не переменную $NAME? □

Упражнение 4.14. Напишите собственную версию команды rm, которая перемещает файлы во временный каталог вместо того, чтобы удалять их. Используйте at для очистки временного каталога в нерабочее время. □

Нерассмотренные возможности

Программа awk представляет собой язык (хоть и несколько неуклюжий), и невозможно полностью описать его в главе разумного размера. Вот неполный список того, о чем стоит почитать в руководстве:

- Перенаправление вывода функции print в файлы и конвейеры: за любым оператором print или printf может следовать знак > и имя файла (как строка в кавычках или переменная); вывод будет осуществляться в указанный файл. Как и в оболочке, >> вызывает добавление в конец вместо перезаписи. Для вывода в конвейер следует использовать | вместо >.
- Многострочные записи: если разделителю записей RS присвоено значение символа новой строки, то входные записи будут разделяться пустой строкой. Таким способом несколько входных строк могут трактоваться как одна запись.
- Конструкция «шаблон, шаблон» является селектором: как и в программах ed и sed группа строк может быть определена парой шаблонов. Селектор позволяет выбрать строки в диапазоне от вхождения

первого шаблона до следующего вхождения второго шаблона. Простой пример

```
NR == 10, NR == 20
```

показывает селектор, соответствующий строкам с 10 по 20 включительно.

4.5. Хорошие файлы и хорошие фильтры

Хотя последние несколько примеров использования `awk` представляют собой законченные программы, часто программы `awk` состоят из одной-двух строк и применяются в составе конвейеров. Это справедливо для большинства фильтров – иногда задача может быть решена при помощи одного фильтра, но чаще приходится разбивать ее на подзадачи, выполняемые отдельными фильтрами, и объединять их в конвейер. Такой подход часто называют основой UNIX-программирования. Хотя этот взгляд и представляется несколько однобоким, фильтры применяются в системе повсеместно, и понимание их работы стоит затраченных на это усилий.

Программы UNIX выводят данные в форматах, пригодных для ввода в другие программы. Файлы, предназначенные для фильтрации, состоят из строк текста, в них отсутствуют украшения в виде заголовков, трейлеров и пустых строк. Каждая строка хранит содержательные данные – имя файла, слово, описание выполняющегося процесса, так что программы типа `wc` и `grep` могут подсчитать объекты или найти их по имени. Если для каждого объекта имеется дополнительная информация, файл по-прежнему состоит из строк, но уже разделенных на поля пробелами или табуляциями, как в выводе команды `ls -l`. Получая данные, разделенные на поля, такие программы, как `awk`, могут выбирать, обрабатывать и перегруппировывать данные.

Конструкция фильтров подчиняется нескольким общим правилам. Все они выводят в стандартный вывод результат обработки файлов-аргументов или данных из стандартного вывода, если были запущены без аргументов. Аргументы всегда определяют *ввод* и никогда – вывод,[1] поэтому вывод команды всегда может быть направлен в конвейер. Необязательные аргументы (или аргументы, не являющиеся именами файлов, например шаблоны `grep`) предшествуют именам файлов. И наконец, сообщения об ошибках следует направлять в стандартный вывод ошибок, чтобы они не исчезли где-то в конвейере.

Эти соглашения слабо влияют на применение команд по отдельности, но если соблюдаются всеми программами, то обеспечивают простоту

[1] В одной из ранних версий UNIX файловая система была уничтожена служебной программой, нарушившей это правило: команда вполне безобидного вида заполнила «мусором» весь диск.

взаимодействия, проиллюстрированную множеством примеров в этой книге. Пожалуй, наиболее наглядно это продемонстрировано в программе подсчета слов в конце раздела 4.2. Если хоть одна из программ потребует имя входного или выходного файла или запросит интерактивного ввода параметров, или выведет заголовки или трейлеры, то конвейер не сработает. И, конечно же, если бы система UNIX не имела конвейеров, кому-нибудь пришлось бы написать такую программу. Но конвейеры существуют, работают и их достаточно легко использовать, если вы владеете нужными инструментами.

Упражнение 4.15. Команда `ps` выводит поясняющий заголовок, а `ls -l` выводит общее количество блоков в файлах. Прокомментируйте. □

История и библиография

Хороший обзор алгоритмов сравнения с шаблоном приведен в статье автора `egrep` Ала Ахо (Al Aho) «Pattern matching in strings» (Сравнение с образцом в строках), изданной «Proceedings of the Symposium on Formal Language Theory» в Санта-Барбаре в 1979 году.

Автором идеи и реализации программы `sed` является Ли Мак-Магон (Lee McMahon), взявший за основу редактор `ed`.

Программа `awk` далеко не так элегантно реализована Алом Ахо (Al Aho), Питером Вейнбергером (Peter Weinberger) и Брайаном Керниганом (Brian Kernighan). То обстоятельство, что язык назван по именам своих авторов, также не свидетельствует о богатом воображении последних. Ее структура рассмотрена разработчиками в статье «AWK – a pattern scanning and processing language» (AWK – язык поиска и обработки по шаблону), напечатанной в «Software – Practice and Experience» в июле 1978 года. Основными источниками идей awk стали язык `SNOBOL4`, `sed`, язык верификации, созданный Марком Рочкиндом (Marc Rochkind), программы `yacc` и `lex` и, разумеется, язык Си. В действительности схожесть `awk` и Си даже порождает трудности, так как язык похож на Си, но таковым не является. Часть конструкций отсутствует, а те, что есть, в чем-то отличаются.

В статье Дага Камера (Doug Comer) «The flat file system FFG: a database system consisting of primitives» (Плоская файловая система FFG: база данных на основе примитивов), появившейся в «Software – Practice and Experience» в ноябре 1982 года, обсуждается создание базы данных с использованием `awk` и оболочки.

5

Программирование в оболочке

Многие пользователи воспринимают оболочку как интерактивный командный процессор, но на самом деле она является языком программирования, в котором каждый оператор запускает команду. Этот исторически сформировавшийся язык во многом необычен, ведь ему приходится обеспечивать как интерактивность, так и программируемость. Разнообразие его применений привело к появлению огромного количества нюансов языка, хотя для эффективной работы не обязательно знать их все. В данной главе на примере разработки нескольких полезных программ поясняются основы программирования в оболочке. Это *не* учебник. Когда вы будете читать эту главу, под рукой всегда должна быть страница `sh(1)` справочного руководства по UNIX.

Особенности поведения оболочки, как и большинства других команд, часто легче понять опытным путем. Руководства могут быть непонятными, и нет ничего лучше удачного примера, чтобы прояснить ситуацию. Вот почему эта глава построена на примерах и представляет собой путеводитель по *использованию* оболочки для программирования, а не энциклопедию ее возможностей. Будет рассказано не только о том, что оболочка умеет делать, но и о том, как разрабатывать и писать программы для нее; акцент будет сделан на интерактивной проверке возникающих идей.

После того как вы напишете программу в оболочке или на каком-либо другом языке, может оказаться, что она так полезна, что другие пользователи системы также захотят с ней поработать. Однако пользователи программ гораздо требовательнее к ним, нежели их авторы. Поэтому главной задачей программирования в оболочке является создание «устойчивых» программ, которые могли бы обработать некорректные

входные данные и выдать полезную информацию в случае, если что-то пошло не так.

5.1. Переделываем команду cal

Программы оболочки часто применяются для улучшения или изменения пользовательского интерфейса программы. В качестве примера программы, которая может быть улучшена, рассмотрим `cal(1)`:

```
$ cal
usage: cal [month] year          Пока все хорошо
$ cal october 1983
Bad argument                     Уже не так хорошо
$ cal 10 1983
   October 1983
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$
```

Досадно, что месяц должен указываться в числовом виде. К тому же, оказывается, что `cal 10` выводит календарь для десятого года вместо того, чтобы делать это для текущего октября, и для того, чтобы получить календарь на один месяц, обязательно надо указывать год.

Обратите внимание на важный момент: какой бы интерфейс ни предоставляла команда `cal`, можно изменить его, не изменяя саму программу. Можно поместить в личный каталог `/bin` команду, которая преобразовывала бы удобный для вас синтаксис в такой, который требуется настоящей программе `cal`. Свою версию можно также назвать `cal` – меньше придется запоминать.

В первую очередь следует составить план того, что должна выполнять `cal`. Будем стремиться к тому, чтобы программа стала разумной. Она должна воспринимать название месяца, а не его номер. Получив два аргумента, она должна вести себя в точности так же, как старая версия, и, кроме того, преобразовывать названия месяцев в номера. Если задан только один аргумент, программа должна вывести календарь на месяц или год (по обстановке), если же аргументы не заданы, новая версия `cal` должна напечатать календарь на текущий месяц – очевидно, что это самое частое применение такой команды. Итак, задача заключается в том, чтобы понять, сколько задано аргументов, затем отобразить их согласно правилам стандартной `cal`.

Для таких решений удобно использовать оператор `case`:

```
case слово in
```

```
шаблон)      команды ;;
шаблон)      команды ;;
...
esac
```

Оператор `case` сравнивает *слово* с шаблонами, двигаясь сверху вниз, и выполняет команды, соответствующие первому (и только первому) подходящему шаблону. Шаблоны записываются по общим правилам, принятым в оболочке для имен файлов с небольшими дополнениями. Каждое действие завершается двойной точкой с запятой `;;`. (Последняя пара `;;` не обязательна, но оставим ее для удобства редактирования.)

Новая версия `cal` определяет количество аргументов, обрабатывает названия месяцев и вызывает настоящую программу `cal`. Переменная оболочки `$#` хранит количество аргументов, с которым был вызван командный файл. Другие специальные переменные оболочки перечислены в табл. 5.1.

Таблица 5.1. Встроенные переменные оболочки

Переменная	Смысл
`$#`	количество аргументов
`$*`	все аргументы
`$@`	аналогично `$*`, см. раздел 5.7
`$-`	параметры, переданные оболочке
`$?`	возвращает код завершения последней выполненной команды
`$$`	идентификатор процесса оболочки
`$!`	идентификатор процесса для последней команды, запущенной с `&`
`$HOME`	аргумент по умолчанию для команды `cd`
`$IFS`	список символов, которые разделяют слова в аргументах
`$MAIL`	файл, при изменении которого выводится сообщение «you have mail»
`$PATH`	список каталогов для поиска команд
`$PS1`	строка приглашения на ввод, по умолчанию `$`
`$PS2`	строка приглашения на продолжение командной строки, по умолчанию `>`

```
$ cat cal
# cal: улучшенный интерфейс к /usr/bin/cal

case $# in
0)    set `date`; m=$2; y=$6 ;;    # нет аргументов: сегодняшний день
1)    m=$1; set `date`; y=$6 ;;    # 1 аргумент: текущий год
*)    m=$1; y=$2 ;;                # 2 аргумента: месяц и год
esac
```

```
case $m in
jan*|Jan*)      m=1 ;;
feb*|Feb*)      m=2 ;;
mar*|Mar*)      m=3 ;;
apr*|Apr*)      m=4 ;;
may*|May*)      m=5 ;;
jun*|Jun*)      m=6 ;;
jul*|Jul*)      m=7 ;;
aug*|Aug*)      m=8 ;;
sep*|Sep*)      m=9 ;;
oct*|Oct*)      m=10 ;;
nov*|Nov*)      m=11 ;;
dec*|Dec*)      m=12 ;;
[1-9]|10|11|12) ;;            # числовое представление месяца
*)              y=$m; m="";; # только год
esac

/usr/bin/cal $m $y            # запустить настоящую программу
$
```

Первый оператор `case` проверяет, сколько задано аргументов (`$#`), и выбирает соответствующее действие. Последний шаблон первого оператора `case` (`*`) – это «ловушка», если количество аргументов не равно ни 0, ни 1, то выполняется последний оператор `case`. (Шаблоны просматриваются сверху вниз, поэтому такой собирающий все шаблон должен быть последним.) Он присваивает переменным `m` и `y` значения месяца и года. Получив два эти аргумента, новая версия `cal` действует так же, как и исходная.

В первом блоке оператора `case` присутствует такой оператор:

```
set `date`
```

По внешнему виду трудно определить, что делает этот оператор, но, если выполнить его, все станет очевидным:

```
$ date
Sat Oct  1 06:05:18 EDT 1983
$ set `date`
$ echo $1
Sat
$ echo $4
06:05:20
$
```

Команда `set` встроена в оболочку, и это команда, которая выполняет очень многие действия. При отсутствии аргументов она отображает значения переменных окружения, как было показано в главе 3. Одиночные аргументы устанавливают значения переменных `$1`, `$2` и т. д. Таким образом, `set `date`` устанавливает `$1` в день недели, `$2` в название месяца и т. д. Поэтому первый оператор `case` программы `cal` устанавливает текущий месяц и год в случае отсутствия аргументов; если

же задан один аргумент, то он воспринимается как месяц, а год берется из текущей даты.

Команда `set` также распознает различные параметры, самыми распространенными из них являются `-v` и `-x`; они включают режим отображения на терминале команд по мере их обработки оболочкой. Эти параметры необходимы для отладки сложных программ.

Остается преобразовать название месяца в текстовой форме в номер. Такую операцию выполняет второй оператор `case`, который практически не требует пояснений, все и так очевидно. Единственная тонкость заключается в использовании в шаблонах символа `|`, который, как и в `egrep`, обозначает альтернативу: `big|small` соответствует или `big`, или `small`. Конечно же, возможна и запись `[jJ]an*` и т. п. Программа допускает введение всех букв названия месяца в нижнем регистре (большинство команд воспринимают ввод в нижнем регистре) или же первая буква названия может быть заглавной (именно в таком формате выводит данные команда `date`). В табл. 5.2 объяснено, каким образом устанавливается соответствие каждому из шаблонов оболочки.

Таблица 5.2. Шаблоны оболочки

Символ	Соответствие
`*`	соответствует любой строке, в том числе и пустой
`?`	соответствует любому отдельному символу
`[ccc]`	соответствует любому из символов в *ccc* `[a-d0-3]` эквивалентно `[abcd0123]`
`"..."`	точно соответствует `...`; кавычки защищают специальные символы. Аналогично действует `'...'`
`\c`	буквально соответствует *c*
`a|b`	(только для операторов `case`) соответствует или `a`, или `b`
`/`	для имен файлов соответствует только явному знаку `/` в выражении; для оператора `case` обрабатывается подобно обычным символам
`.`	будучи первым символом имени файла, соответствует только явной `.` в выражении

Два последних рассматриваемых случая оператора `case` анализируют единственный аргумент, который мог бы быть годом (вспомните, что первый оператор `case` предполагал, что это месяц). Если это число, которое может представлять месяц, оно не обрабатывается. Иначе считается, что это год.

Наконец, последняя строка вызывает `/usr/bin/cal` (то есть настоящую `cal`) с преобразованными аргументами. Новая версия `cal` делает то, что ожидает от нее человек, впервые столкнувшийся с такой программой:

```
$ date
Sat Oct  1 06:09:55 EDT 1983
```

```
$ cal
   October 1983
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$ cal dec
   December 1983
 S  M Tu  W Th  F  S
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

$
```

А команда `cal 1984` выводит календарь на весь 1984 год.

Новая программа делает то же самое, что и исходная, но более простым и легким для восприятия способом. Поэтому мы решили назвать ее `cal`, а не `calendar` (команда с таким именем уже существует) и не `ncal` (менее мнемоническое название). Сохранение прежнего имени удобно для пользователей – не надо запоминать новые названия, можно продолжать действовать рефлекторно.

И, чтобы закончить с оператором `case`, обратим внимание на отличие правил установления соответствия шаблону оболочки от аналогичных правил в `ed` и его производных. Ведь наличие двух типов шаблонов означает, что необходимо изучить два набора правил и что их обработка осуществляется двумя фрагментами кода. Некоторые расхождения появились исключительно из-за неудачного выбора и так и не были исправлены, например нет никакого объяснения (за исключением совместимости с уже забытым прошлым) тому, что в `ed` соответствие любому символу задается посредством «`.`», а в оболочке – «`?`». Но есть и шаблоны, выполняющие разные задачи. Регулярные выражения редактора ищут символьную строку, которая может встретиться в любом месте строки, а для «привязки» поиска к началу и концу строки используются специальные символы `^` и `$`. Однако для имен файлов хотелось бы иметь такую привязку по умолчанию (поскольку это самый распространенный случай). Вводить

```
$ ls ^?*.c$          В таком виде не работает
```

вместо

```
$ ls *.c
```

было бы очень неудобно.

Упражнение 5.1. Если пользователи предпочитают усовершенствованную вами новую версию `cal`, то как сделать ее глобально доступной? Что надо сделать для того, чтобы поместить ее в `/usr/bin`? □

Упражнение 5.2. Стоит ли подправить `cal` таким образом, чтобы `cal 83` выводила календарь на 1983 год? Если да, то как тогда вывести календарь на 83 год? □

Упражнение 5.3. Измените `cal` так, чтобы можно было задавать сразу несколько месяцев, например

```
$ cal oct nov
```

или диапазон месяцев:

```
$ cal oct - dec
```

Если сейчас декабрь, и вводится команда `cal jan`, то календарь должен быть выведен для января текущего или следующего года? Когда же надо остановиться и прекратить улучшать `cal`? □

5.2. Какие команды мы выполняем, или команда which

Создание персональных версий программ, подобных `cal`, создает некоторые трудности. Наиболее явная из них формулируется следующим образом: если вы работаете вместе с Мэри и вводите `cal`, будучи зарегистрированным как `mary`, отработает стандартная, а не новая версия команды (если только в каталоге `/bin`, принадлежащем Мэри, нет ссылки на соответствующий файл). Это не очень удобно (вспомните, например, что сообщения об ошибках, выдаваемые исходной программой `cal`, не очень-то полезны), но это всего лишь одно из проявлений одной большой проблемы. Поскольку оболочка ищет команды в каталогах, указанных в переменной `PATH`, всегда есть шанс получить не ту версию команды, которая ожидается. Например, если ввести команду `echo`, то путем к файлу, который исполнится, может быть `./echo` или `/bin/echo`, или `/usr/bin/echo`, или что-то еще, в зависимости от компонентов переменной `PATH` и местоположения файлов. И если исполняемый файл с правильным именем, но делающий не то, чего ожидает пользователь, встретится на пути поиска раньше, то возникнет путаница. Наверное, самым ярким примером является команда `test` (о ней будет рассказано позже): это имя очевидным образом подходит для временных версий программы, которые и вызываются (гораздо чаще, чем хотелось бы) при обращении к настоящей `test`.[1] Полезной может оказаться программа, информирующая о том, какая версия программы будет выполняться.

[1] Дальше будет показано, как избежать подобных проблем в командных файлах, где обычно используется `test`.

Одна из возможных реализаций заключается в применении цикла для просмотра всех каталогов, перечисленных в PATH, в поиске исполняемых файлов с заданным именем. В главе 3 для перебора имен файлов и аргументов использовался оператор for. В данном случае нужен оператор такого вида:

```
for i in каждый компонент PATH
do
        если файл с заданным именем есть в каталоге i,
                вывести его полное путевое имя
done
```

Поскольку любая команда может быть запущена из-под обратных кавычек `...`, то очевидно следующее решение – применить команду sed к $PATH, заменяя двоеточия пробелами. Протестируем при помощи нашего старого друга echo:

```
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin       4 компонента
$ echo $PATH | sed 's/:/ /g'
 /usr/you/bin /bin /usr/bin       Выведено только 3!
$ echo `echo $PATH | sed 's/:/ /g'`
/usr/you/bin /bin /usr/bin        Также только 3
$
```

Отчетливо видна проблема: пустая строка в PATH является синонимом точки «.», и в результате преобразования двоеточий в пробелы информация о нулевых компонентах будет утрачена. Для того чтобы получить корректный список каталогов, следует преобразовать нулевую компоненту PATH в точку. Нулевой компонент может находиться как в середине строки, так и с любого конца, необходимо предусмотреть все эти варианты:

```
$ echo $PATH | sed 's/^:/.:/
>                   s/::/:.:/g
>                   s/:$/:./
>                   s/:/ /g'
. /usr/you/bin /bin /usr/bin
$
```

Можно было бы записать это как четыре отдельных команды sed, но так как sed осуществляет замены по порядку, достаточно одного вызова программы.

Как только стало известно, какие каталоги являются компонентами PATH, команда test(1), которая уже упоминалась, может определить, присутствует ли файл в каждом из каталогов. Надо сказать, что test является одной из самых неуклюжих программ UNIX. Например, test -r file проверяет, существует ли файл и доступен ли он для чтения, а test -w file проверяет, существует ли файл и разрешена ли в него запись. В седьмой версии нет команды test -x (она существовала в

System V и других версиях), которая бы вполне подошла для нашей задачи. Будем использовать `test -f`, она проверяет, существует ли файл и не является ли он каталогом (то есть, другими словами, является ли он обычным файлом). Сейчас в обращении находится несколько версий программы `test`, так что обратитесь к руководству для получения информации о вашей системе.

Любая команда возвращает *код завершения* – число, по значению которого оболочка определяет, что произошло. Код завершения – это целое число, принято соглашение о том, что `0` означает «истина» (команда выполнилась успешно), а ненулевая величина означает «ложь» (команда не выполнилась успешно). Обратите внимание на что, что в языке Си значения истины и лжи прямо противоположны только что описанным.

Поскольку «ложь» может задаваться различными ненулевыми величинами, то в этом коде завершения часто зашифровывается причина неудачи. Например, команда `grep` возвращает `0`, если найдено соответствие, `1` – если соответствие не найдено, и `2` – если в имени файла или шаблоне есть ошибка. Код завершения возвращается любой программой, несмотря на то, что его значение часто бывает не интересно пользователю. Команда `test` – это необычная команда, ее единственное предназначение заключается в возврате кода завершения. Она ничего не выводит и не изменяет файлы.

Оболочка хранит код завершения последней программы в переменной `$?`:

```
$ cmp /usr/you/.profile /usr/you/.profile
$                                   Нет вывода; они одинаковые
$ echo $?
0                Ноль означает успешное выполнение: файлы идентичны
$ cmp /usr/you/.profile /usr/mary/.profile
/usr/you/.profile /usr/mary/.profile differ: char 6, line 3
$ echo $?
1              Ненулевая величина означает, что файлы были различными
$
```

У некоторых команд, например `cmp` и `grep`, есть параметр `-s`, и если он задан, то команда возвращает соответствующий код завершения, но ничего не выводит.

Оператор оболочки `if` выполняет команды в зависимости от кода завершения команды:

```
if команда
then
        команды, если условие истинно
else
        команды, если условие ложно
fi
```

Положение разделителей строк имеет важное значение: `fi`, `then` и `else` распознаются, только если находятся в начале строки или следуют за точкой с запятой. Часть `else` может отсутствовать.

Оператор `if` всегда запускает команду, являющуюся условием, тогда как оператор `case` осуществляет сравнение с шаблоном непосредственно в оболочке. В некоторых версиях UNIX (в том числе в System V) `test` является встроенной функцией оболочки, поэтому `if` и `test` будут выполнены так же быстро, как и `case`. Если же `test` не является встроенной, то операторы `case` более производительны, чем операторы `if`, и именно их следует применять для любого сопоставления с шаблоном:

```
case "$1" in
hello)  команда
esac
```

выполнится быстрее, чем

```
if test "$1" = hello     Медленнее, если test не встроена в оболочку
then
        команда
fi
```

Именно по этой причине операторы `case` иногда применяются в оболочке для тестирования того, что в большинстве языков программирования тестировали бы с помощью оператора `if`. С другой стороны, в операторе `case` нет простого способа, позволяющего определить, есть ли права на чтение файла, это лучше сделать, используя `test` и `if`.

Теперь все готово для написания первой версии команды `which`, которая будет сообщать, какой файл соответствует команде:

```
$ cat which
# which cmd: какая из имеющихся в $PATH команд выполняется, версия 1

case $# in
0)  echo 'Usage: which command' 1>&2; exit 2
esac
for i in `echo $PATH | sed 's/^:/.:/
                            s/::/:.:/g
                            s/:$/:./
                            s/:/ /g'`
do
    if test -f $i/$1    # если можно, используйте test -x
    then
        echo $i/$1
        exit 0          # найдено
    fi
done
exit 1                  # не найдено
$
```

Давайте опробуем ее:

```
$ cx which                          Делаем команду исполняемой
$ which which
./which
$ which ed
/bin/ed
$ mv which /usr/you/bin
$ which which
/usr/you/bin/which
$
```

Оператор case использован просто для контроля ошибок. Обратите внимание на перенаправление 1>&2 в команде echo – сообщения об ошибке не пропадут в канале. Встроенная в оболочку функция exit может быть использована для возврата кода завершения. Если команда не работает, то возвращается код ошибки – exit 2, если файл не найден – exit 1, если же найден – exit 0. Если оператор exit явно не присутствует, то код завершения для командного файла – это статус последней выполненной команды.

Что произойдет, если в текущем каталоге есть программа под названием test? (Действуем в предположении, что test не является встроенной функцией оболочки.)

```
$ echo 'echo hello' >test          Создадим поддельную test
$ cx test                          Сделаем ее исполняемой
$ which which                      Теперь попробуем which
hello                              Неудача!
./which
$
```

Требуется уделить больше внимания контролю за ошибками. Можно запустить which (если в текущем каталоге не было команды test!), найти полное путевое имя test и указать его явно. Но это плохой способ: в разных системах программа test может находиться в разных каталогах, а команда which также зависит от sed и echo, так что придется указывать и их путевые имена. Существует более простое решение: переопределить PATH в командном файле так, чтобы выполнялись только команды из каталогов /bin и /usr/bin. Старое значение PATH, несомненно, должно быть сохранено (только для команды which, чтобы определить последовательность каталогов для просмотра).

```
$ cat which
# which cmd: какая команда выполняется, окончательная версия

opath=$PATH
PATH=/bin:/usr/bin

case $# in
0)  echo 'Usage: which command' 1>&2; exit 2
esac
```

```
for i in `echo $opath | sed 's/^:/.:/
                             s/::/:.:/g
                             s/:$/:./
                             s/:/ /g'`
do
    if test -f $i/$1    # это только /bin/test
    then                # или /usr/bin/test
        echo $i/$1
        exit 0          # найдено
    fi
done
exit 1          # не найдено
$
```

Команда `which` теперь работает даже в случае наличия на пути поиска фальшивой `test` (или `sed`, или `echo`).

```
$ ls -l test
-rwxrwxrwx 1 you        11 Oct  1 06:55 test      Все еще здесь
$ which which
/usr/you/bin/which
$ which test
./test
$ rm test
$ which test
/bin/test
$
```

В оболочке есть два оператора для комбинирования команд: `||` и `&&`, они более компактны, и часто работать с ними удобнее, чем с оператором `if`. Так, `||` может заменить некоторые операторы `if`:

```
test -f имя-файла || echo file имя-файла does not exist
```

эквивалентно

```
if test ! -f имя-файла              Символ ! инвертирует условие
then
        echo file имя-файла does not exist
fi
```

Несмотря на его внешний вид оператор `||` не имеет ничего общего с каналами; это условный оператор, логическое ИЛИ. Выполняется команда, находящаяся слева от `||`. Если код завершения равен нулю (выполнено успешно), то команда справа от `||` игнорируется. Если же левая часть возвращает ненулевую величину (неуспешно), то выполняется правая часть, и значение всего выражения равно коду завершения правой части. Другими словами, `||` – это оператор логического ИЛИ, который не выполняет команду правой части, если левая выполнилась успешно. Аналогично `&&` соответствует логическому И; он выполняет команду своей правой части, только если успешно выполнилась левая.

Упражнение 5.4. Почему перед выходом команда `which` не устанавливает `PATH` в `opath`? □

Упражнение 5.5. Если в оболочке завершение `case` реализуется посредством `esac`, а завершение `if` – с помощью `fi`, то почему для завершения `do` используется `done`? □

Упражнение 5.6. Добавьте параметр `-a` для `which`, чтобы она печатала все файлы в `PATH`, а не выходила после первого найденного. Подсказка: `match='exit0'`. □

Упражнение 5.7. Измените `which` так, чтобы она знала о встроенных в оболочку функциях типа `exit`. □

Упражнение 5.8. Измените `which` так, чтобы она проверяла права на выполнение файлов. Пусть она печатает сообщение об ошибке в случае, если файл не найден. □

5.3. Циклы while и until: организация поиска

В главе 3 для выполнения некоторого количества повторяющихся программ применялся цикл `for`. Обычно цикл `for` просматривает список имен файлов (например, `for i in *.c`) или все аргументы программы оболочки (`for i in $*`). Но циклы оболочки могут применяться не только для решения таких задач, – посмотрите на цикл `for` в программе `which`.

Существует три вида циклов: `for`, `while` и `until`. Безусловно, самым распространенным из них является `for`. Он выполняет набор команд – тело цикла – один раз для каждого элемента из множества слов (чаще всего это имена файлов). Циклы `while` и `until` выполняют команды тела цикла на основе анализа кода завершения команды. В операторе `while` тело цикла выполняется до тех пор, пока команда-условие не вернет ненулевой код; для оператора `until` – пока команда-условие не вернет нулевой код. Циклы `while` и `until` идентичны, если не считать различий в интерпретации кода завершения команды.

Базовая форма каждого из описанных циклов выглядит следующим образом:

```
for i in список слов
do
        тело цикла, $i устанавливается в следующий элемент списка
done

for i   (Подразумевается список всех аргументов командного
         файла, т. е. $*)
do
        тело цикла, $i устанавливается в следующий аргумент
done

while команда
```

```
do
        тело цикла выполняется, пока команда возвращает true
done

until   команда
do
        тело цикла выполняется, пока команда возвращает false
done
```

Вторая форма `for`, в которой под пустой строкой подразумевается `$*`, представляет собой удобную краткую запись для постоянной работы.

В качестве команды-условия, управляющей циклом `while` или `until`, может выступать любая команда. Приведем тривиальный пример – цикл `while`, следящий за входом пользователя (скажем, Мэри) в систему:

```
while sleep 60
do
        who | grep mary
done
```

Команда `sleep`, создающая паузу длиной 60 секунд, обычно выполняется всегда (до тех пор, пока не будет прервана), следовательно, возвращает «успех», так что цикл раз в минуту будет проверять, зарегистрировалась ли Мэри.

Недостаток этой версии заключается в том, что если Мэри уже зарегистрирована, вы сможете узнать об этом только через 60 секунд. И если Мэри продолжает работать в системе, раз в минуту будут поступать сообщения об этом. Можно вывернуть цикл «наизнанку» и переписать его при помощи оператора `until`, чтобы получать информацию один раз, без задержки – в случае, если Мэри сейчас в системе:

```
until who | grep mary
do
        sleep 60
done
```

Это более интересное условие. Если Мэри в системе, то `who | grep mary` выводит ее данные в списке `who` и возвращает «истину», потому что `grep` возвращает код, указывающий, найдено ли что-нибудь, а кодом завершения конвейера является код завершения его последнего элемента.

Теперь завершим эту команду, дадим ей название и проинсталлируем ее:

```
$ cat watchfor
# watchfor: следит за входом пользователя в систему

PATH=/bin:/usr/bin
```

```
case $# in
0)  echo 'Usage: watchfor person' 1>&2; exit 1
esac

until who | egrep "$1"
do
    sleep 60
done
$ cx watchfor
$ watchfor you
you     tty0    Oct   1 08:01     Работает
$ mv watchfor  /usr/you/bin       Инсталлируем ее
$
```

Команда `grep` заменена на `egrep`, поэтому можно ввести

```
$ watchfor 'joe|mary'
```

чтобы отслеживать сразу нескольких пользователей.

Более сложный пример: будем отслеживать вход в систему и выход из нее *всех* пользователей и выводить отчет о «движении» пользователей – получится нечто вроде расширенной команды `who`. Базовая структура очень проста: раз в минуту запускать `who`, сравнивать ее выходные данные с ее же данными минуту назад и печатать информацию о любых отличиях. Выходные данные `who` будут храниться в файле, расположенном в каталоге `/tmp`. Чтобы отличать наши файлы от файлов, принадлежащих другим процессам, в имена файлов включена переменная оболочки `$$` (идентификатор процесса команды); это общепринятое соглашение. В имени временного файла зашифровано название соответствующей команды – это удобно для системного администратора. Команды (в том числе и эта версия `watchwho`) часто оставляют ненужные файлы в `/tmp`, поэтому важно иметь возможность определить, какая именно команда так поступает.

```
$ cat watchwho
# watchwho: следит за тем, кто входит в систему и выходит из нее

PATH=/bin:/usr/bin
new=/tmp/wwho1.$$
old=/tmp/wwho2.$$
>$old       # создать пустой файл

while :     # вечный цикл
do
    who >$new
    diff $old $new
    mv $new $old
    sleep 60
done | awk '/>/ { $1 = "in:     "; print }
            /</ { $1 = "out:    "; print }'
```

Команда : встроена в оболочку, единственное, что она умеет делать – это оценивать свои аргументы и возвращать «истину». Вместо нее можно было использовать команду true, которая просто возвращает код завершения «истина». (Существует и команда false.) Но : более эффективна, чем true, так как она не выполняет команду из файловой системы.

В выводе diff использованы < и >, чтобы различать данные из двух файлов; программа awk обрабатывает эту информацию и сообщает об изменениях в более доступном для понимания формате. Обратите внимание на то, что весь цикл while соединяется с awk, образуя конвейер (а можно было бы запускать awk отдельно, раз в минуту). Для такой обработки нельзя использовать команду sed, поскольку ее вывод всегда следует за строкой ввода: всегда есть строка ввода, которая обрабатывается, но не выводится, так возникает ненужная задержка.

Файл old создается пустым, поэтому первым выводом команды watchwho является список пользователей, находящихся в системе в настоящий момент. Если заменить команду, которая первоначально создает old, на who >$old, то watchwho будет печатать только отличия; выбор здесь – дело вкуса.

Рассмотрим другую программу, реализация которой требует применения циклов: периодический просмотр почтового ящика. Если есть изменения, программа выводит сообщение «Вам пришла почта». Такая программа совсем не лишняя, она представляет собой хорошую альтернативу встроенному в оболочку механизму, использующему переменную MAIL. Представленная реализация использует вместо файлов переменные оболочки (просто для того, чтобы показать, что такое тоже возможно).

```
$ cat checkmail
# checkmail: следит за увеличением размера почтового ящика

PATH=/bin:/usr/bin
MAIL=/usr/spool/mail/`getname`  # зависит от системы

t=${1-60}

x="`ls -l $MAIL`"
while :
do
    y="`ls -l $MAIL`"
    echo $x $y
    x="$y"
    sleep $t
done | awk '$4 < $12 { print "Вам пришла почта" }'
$
```

Снова мы прибегли к команде awk, на этот раз для того, чтобы обеспечить вывод сообщения только в случае увеличения почтового ящика, а не просто при его изменении. В противном случае сообщение выводи-

лось бы и после удаления сообщения. (Этот изъян есть у версии, встроенной в оболочку.)

Промежуток времени обычно равен 60 секундам, но можно задать другое значение, указав параметр в командной строке:

```
$ checkmail 30
```

Переменная оболочки `t` устанавливается в заданное значение (если оно введено) или в `60`, если значение не задано, посредством строки

```
t=${1-60}
```

В ней используется еще одно свойство оболочки.

Обозначение `${var}` эквивалентно `$var`, оно удобно для того, чтобы избежать неприятностей с переменными внутри строк, содержащих буквы или цифры:

```
$ var=hello
$ varx=goodbye
$ echo $var
hello
$ echo $varx
goodbye
$ echo ${var}x
hellox
$
```

Некоторые символы внутри фигурных скобок задают специальную обработку переменной. Если переменная не определена, а после ее имени стоит знак вопроса, то выводится строка, следующая за `?`, и оболочка завершает выполнение (если только это не интерактивная сессия). Если сообщение не задано, то печатается стандартное:

```
$ echo ${var?}
hello                                  Все хорошо; var определена
$ echo ${junk?}
junk: parameter not set      Стандартное сообщение (по умолчанию)
$ echo ${junk?error!}
junk: error!                           Выведено заданное сообщение
$
```

Обратите внимание, что сообщение, генерируемое оболочкой, всегда содержит имя переменной, которая не определена.

Еще одна возможная форма данного обозначения выглядит так: `${var-thing}`, это выражение имеет значение `$var`, если `var` определена, и `thing`, если не определена. Выражение `${var=thing}` вычисляется аналогично, но оно еще присваивает `$var` значение `thing`:

```
$ echo ${junk-'Hi there'}
Hi there
```

```
$ echo ${junk?}
junk: parameter not set          junk не изменен
$ echo ${junk='Hi there'}
Hi there
$ echo ${junk?}
Hi there                         junk установлен в Hi there
$
```

Правила вычисления переменных приведены в табл. 5.3.

Таблица 5.3. Вычисление переменных оболочки

Переменная	Значение
`$var`	значение `var`; если `var` не определена, то ничего
`${var}`	аналогично; удобно использовать, если за именем переменной следует буквенно-цифровое выражение
`${var-thing}`	значение `var`, если она определена; в противном случае `thing`; `$var` не изменяется
`${var=thing}`	значение `var`, если она определена; в противном случае `thing`, `$var` устанавливается в `thing`
`${var?message}`	значение `var`, если она определена. Иначе вывести `message` и выйти из оболочки. Если сообщение пусто, вывести: `var: parameter not set`
`${var+thing}`	`thing`, если `var` определена, иначе ничего

Вернемся к примеру, с которого началось обсуждение таких выражений:

```
t=${1-60}
```

`t` устанавливается в `$1`, а если аргумент не задан, то в `60`.

Упражнение 5.9. Посмотрите на реализацию команд `true` и `false` в каталоге `/bin` или `/usr/bin`. (Как их найти?) □

Упражнение 5.10. Измените команду `watchfor` так, чтобы при задании нескольких аргументов они воспринимались как разные пользователи (чтобы не приходилось вводить `'joe|mary'`). □

Упражнение 5.11. Напишите версию `watchwho`, которая использовала бы `comm`, а не `awk`, для сравнения старых и новых данных. Какую из версий вы предпочитаете? □

Упражнение 5.12. Напишите версию `watchwho`, которая хранила бы выходные данные `who` не в файлах, а в переменных оболочки. Какая версия вам больше нравится? Какая работает быстрее? Следует ли командам `watchwho` и `checkmail` выполнять `&` автоматически? □

Упражнение 5.13. В чем отличие команды оболочки `:` (которая ничего не делает) от символа комментария `#`? Действительно ли нужны оба? □

5.4. Команда trap: перехват прерываний

Если нажать клавишу *Del* или повесить телефонную трубку во время выполнения команды `watchwho`, в каталоге `/tmp` останутся один или два временных файла. Перед выходом `watchwho` должна удалять свои временные файлы. Необходимо иметь возможность обнаруживать подобные ситуации и способ восстановления нормального хода событий.

При нажатии клавиши *Del сигнал прерывания* посылается всем процессам, работающим на терминале в настоящий момент. Аналогично, когда повешена телефонная трубка, отправляется *сигнал отбоя*. Существуют и другие сигналы. До тех пор пока программа не обрабатывает сигналы явно, сигнал будет завершать ее выполнение. Оболочка защищает программы, запущенные с использованием `&` от прерываний, но не от отсоединения.

В главе 7 будет подробно рассказано о сигналах, а для того чтобы обрабатывать их в оболочке, надо знать лишь немногое. Встроенная в оболочку команда `trap` определяет последовательность команд, которая должна быть выполнена, когда поступит сигнал:

```
trap последовательность-команд список-номеров-сигналов
```

Последовательность-команд является единым аргументом, так что почти всегда приходится заключать ее в кавычки. Номера сигналов – это целые числа, идентифицирующие сигналы. Например, `2` – это сигнал, генерируемый при нажатии клавиши *Del*, а `1` – сигнал, генерируемый, когда повешена телефонная трубка. Номера сигналов, наиболее часто используемых при программировании в оболочке, перечислены в табл. 5.4.

Таблица 5.4. Номера сигналов оболочки

Номер	Значение
0	выход из оболочки (по любой причине, в том числе в конце файла)
1	отбой (повешена телефонная трубка)
2	прерывание (клавиша *Del*[a])
3	выход (*ctl-*; требует от программы создания дампа памяти)
9	`kill` (не может быть ни перехвачен, ни проигнорирован)
15	завершение – сигнал, по умолчанию генерируемый `kill(1)`

[a] На клавиатурах старых терминалов была специальная клавиша *Del*, выдававшая сигнал прерывания. Для того чтобы послать текущему процессу SIGINT, используйте комбинацию клавиш *Ctrl+C*. – *Примеч. науч. ред.*

Поэтому для того, чтобы очистить временные файлы в `watchwho`, непосредственно перед циклом следует поместить команду `trap` для перехвата сигналов отбоя, прерывания и завершения:

```
...
```

```
trap 'rm -f $new $old; exit 1' 1 2 15
while :
...
```

Последовательность команд, образующая первый аргумент trap, подобна обращению к подпрограмме, которое происходит сразу же после получения сигнала. Когда она заканчивается, программа, которая выполнялась, будет возобновлена с того места, в котором она находилась до получения сигнала. Таким образом, последовательность команд trap должна явно запустить exit, иначе программа оболочки будет продолжать выполняться после прерывания. Еще одно замечание – последовательность команд будет прочитана дважды: в момент установки trap и в момент обращения к ней. Поэтому лучше поместить последовательность команд в одинарные кавычки, тогда переменные будут вычисляться только при выполнении функций trap. В примере, рассматриваемом сейчас, это не имеет особого значения, но позже будет приведен еще один пример, для которого это важно. Параметр -f указывает команде rm, что она не должна задавать вопросов.

Бывает, что trap оказывается удобной при интерактивной работе, чаще всего это происходит тогда, когда надо предотвратить уничтожение программы сигналом отбоя, сгенерированным прерванным телефонным соединением:

```
$ (trap '' 1; долго-выполняющаяся-команда) &
2134
$
```

Пустая последовательность команд означает требование «игнорировать прерывания» в данном процессе и в порожденных им процессах. Скобки в записи команды являются причиной того, что trap и команда выполняются вместе в фоновом режиме в подоболочке; если бы скобок не было, то trap была бы применена и к родительской оболочке и к долго-выполняющейся-команде.

Команда nohup(1) – это коротенькая программа оболочки, предоставляющая такую услугу. Вот ее полная версия в седьмой версии:

```
$ cat `which nohup`
trap "" 1 15
if test -t 2>&1
then
    echo "Sending output to 'nohup.out'"
    exec nice -5 $* >>nohup.out 2>&1
else
    exec nice -5 $* 2>&1
fi
$
```

Команда test -t проверяет, является ли стандартный вывод терминалом, для того чтобы знать, надо ли сохранять выходные данные. Про-

грамма запускается в фоновом режиме с nice, получая таким образом более низкий приоритет, чем интерактивные программы. (Обратите внимание на то, что nohup не задает PATH. Должна ли она это делать?)

Команда exec включена только для повышения производительности; без нее команды выполнялись бы точно так же, как и с ней. Но exec встроена в оболочку, она заменяет процесс выполняющейся в данное время оболочки указанной программой, экономя таким образом один процесс – оболочку, которая обычно ожидала бы завершения программы. Можно было использовать команду exec и во многих других местах, например в конце улучшенной программы cal, когда происходит обращение к /usr/bin/cal.

Кстати, сигнал 9 – это тот сигнал, который не может быть перехвачен или проигнорирован; он уничтожает всегда. Из оболочки он посылается следующим образом:

```
$ kill -9 идентификатор-процесса ...
```

По умолчанию kill -9 не используется, потому что у процесса, уничтожаемого таким сигналом, нет возможности «привести дела в порядок», прежде чем умереть.

Упражнение 5.14. Версия программы nohup, представленная выше, объединяет стандартный вывод ошибок со стандартным выходным потоком. Хорошо ли это? Если нет, то как аккуратно отделить их друг от друга?

Упражнение 5.15. Найдите встроенную в оболочку команду times и добавьте в свой .profile строку так, чтобы при выходе из системы оболочка сообщала, сколько было использовано процессорного времени.

Упражнение 5.16. Напишите программу, которая искала бы следующий доступный идентификатор пользователя в /etc/passwd. Если вам это интересно (и есть соответствующие права), переделайте таким образом команду, добавляющую нового пользователя в систему. Какие нужны права? Как следует обрабатывать прерывания?

5.5. Замена файла: команда overwrite

У команды sort есть параметр -o для перезаписи файла:

```
$ sort file1 -o file2
```

Эта запись эквивалентна следующей:

```
$ sort file1 >file2
```

Если file1 и file2 – это один и тот же файл, то перенаправление > очистит входной файл, не выполняя сортировку. А параметр -o работает корректно, т. к. входные данные сортируются и сохраняются во временном файле до того, как создается выходной файл.

Многие другие команды также могли бы использовать параметр `-o`. Например, `sed` могла бы редактировать файл прямо на месте:

```
$ sed 's/UNIX/UNIX(TM)/g' ch2 -o ch2      Так не работает!
```

Было бы непрактично изменять все подобные команды так, чтобы можно было добавить этот параметр. Более того, это вообще плохая идея: лучше централизовать функции, как делает оболочка при помощи оператора `>`. Давайте напишем программу `overwrite`, которая будет выполнять эту работу. Пример, приведенный выше, будет выглядеть так:

```
$ sed 's/UNIX/UNIX(TM)/g' ch2 | overwrite ch2
```

Основная мысль весьма незатейлива – просто сохранять входные данные, пока не будет достигнут конец файла, затем скопировать их в указанный файл:

```
# overwrite: копировать стандартный ввод в вывод после EOF
# версия 1. С ОШИБКОЙ

PATH=/bin:/usr/bin

case $# in
1)  ;;
*)  echo 'Usage: overwrite file' 1>&2; exit 2
esac

new=/tmp/overwr.$$
trap 'rm -f $new; exit 1' 1 2 15

cat >$new       # собрать входные данные
cp $new $1      # перезаписать входной файл
rm -f $new
```

Команда `cp` использована вместо `mv` для того, чтобы права и владелец выходного файла (если он уже существовал) не изменились.

В этой заманчиво простой версии сделана роковая ошибка: если во время выполнения `cp` пользователь нажмет клавишу *Del*, то исходный входной файл будет испорчен. Необходимо предотвратить остановку перезаписи входного файла, вызываемую прерыванием:

```
# overwrite: копировать стандартный ввод в вывод после EOF
# версия 2. И здесь есть ОШИБКА

PATH=/bin:/usr/bin

case $# in
1)  ;;
*)  echo 'Usage: overwrite file' 1>&2; exit 2
esac

new=/tmp/overwr1.$$
old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15

cat >$new           # получить входные данные
```

```
cp $1 $old          # сохранить исходный файл

trap `` 1 2 15      # до завершения игнорировать сигналы
cp $new $1          # перезаписать входной файл

rm -f $new $old
```

Если клавиша *Del* нажата до начала обработки исходного файла, то временные файлы удаляются, и с файлом ничего не происходит. После создания резервной копии сигналы игнорируются, поэтому последняя команда `cp` не будет прервана и файл будет перезаписан полностью.

Все еще остается небольшая проблема. Рассмотрим

```
$ sed 's/UNIX/UNIX(TM)g' precious | overwrite precious
command garbled: s/UNIX/UNIX(TM)g
$ ls -l precious
-rw-rw-rw- 1 you           0 Oct  1 09:02 precious    #$%@*!
$
```

Если в программе, снабжающей `overwrite` входными данными, есть ошибка, то ее вывод будет пустым, и надежная `overwrite`, как ей и положено, уничтожит данные во входном файле.

Можно предложить несколько решений. Например, `overwrite` может запрашивать подтверждение на замену файла, но если сделать эту команду интерактивной, многие ее преимущества будут утрачены. Можно сделать так, чтобы `overwrite` проверяла свои входные данные на «непустоту» (посредством `test -z`), но это некрасиво и неправильно, к тому же какая-то часть выходных данных уже может быть сгенерирована к моменту обнаружения ошибки.

Лучшим решением является запуск программы, генерирующей данные, под контролем `overwrite` – так, чтобы можно было проверять код завершения. Традиции и наша интуиция выступают против такого подхода. В конвейере `overwrite` обычно находится в конце. Но она должна быть первой, т. к. иначе не будет работать правильно. Команда `overwrite` ничего не направляет в стандартный вывод, так что универсальность сохраняется. А ее синтаксис не такой уж необычный: `time`, `nice`, `nohup` – все эти команды воспринимают другие команды в качестве аргументов.

Вот безопасная версия программы:

```
# overwrite: копировать стандартный ввод в вывод после EOF
# окончательная версия

opath=$PATH
PATH=/bin:/usr/bin

case $# in
0|1)    echo 'Usage: overwrite file cmd [args]' 1>&2; exit 2
esac

file=$1; shift
```

```
new=/tmp/overwr1.$$; old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15   # очистить файлы

if PATH=$opath "$@" >$new          # получить входные данные
then
    cp $file $old       # сохранить исходный файл
    trap '' 1 2 15      # до завершения игнорировать сигналы
    cp $new $file
else
    echo "overwrite: $1 failed, $file unchanged" 1>&2
    exit 1
fi
rm -f $new $old
```

Встроенная в оболочку команда `shift` сдвигает весь список аргументов на одну позицию влево: `$2` превращается в `$1`, `$3` в `$2` и т. д. Выражение «`$@`» подразумевает все аргументы (после выполнения `shift`), как и `$*`, но не интерпретированные; подробнее об этом будет рассказано в разделе 5.7.

Обратите внимание, что `PATH` восстанавливается для запуска команд пользователя; если бы этого не происходило, то команды, не содержащиеся в `/bin` или `/usr/bin`, были бы недоступны для `overwrite`.

Теперь `overwrite` работает (хоть и выглядит несколько громоздко):

```
$ cat notice
UNIX is a Trademark of Bell Laboratories
$ overwrite notice sed 's/UNIXUNIX(TM)/g' notice
command garbled: s/UNIXUNIX(TM)/g
overwrite: sed failed, notice unchanged
$ cat notice
UNIX is a Trademark of Bell Laboratories          Без изменений
$ overwrite notice sed 's/UNIX/UNIX(TM)/g' notice
$ cat notice
UNIX(TM) is a Trademark of Bell Laboratories
$
```

Команда `sed` часто применяется для того, чтобы заменить все вхождения одного слова другим. Имея в наличии `overwrite`, легко создать командный файл, автоматизирующий выполнение данной задачи:

```
$ cat replace
# replace: заменить str1 в файлах на str2

PATH=/bin:/usr/bin

case $# in
0|1|2)  echo 'Usage: replace str1 str2 files' 1>&2; exit 1
esac

left="$1"; right="$2"; shift; shift

for i
do
       overwrite $i sed "s@$left@$right@g" $i
```

```
done
$ cat footnote
UNIX is not an acronym
$ replace UNIX Unix footnote
$ cat footnote
Unix is not an acronym
$
```

(Не забывайте о том, что если список для оператора `for` пуст, то подразумевается значение по умолчанию, `$*`.) Вместо / для разграничения команды замены использован символ `@`, так как его конфликт со строкой ввода менее вероятен.

Команда `replace` устанавливает `PATH` в `/bin:/usr/bin`, исключая `$HOME/bin`. Это означает, что `overwrite` должна находиться в `/usr/bin`, чтобы `replace` работала. Такое допущение сделано для того, чтобы упростить задачу; если не удается инсталлировать `overwrite` в `/usr/bin`, надо внести `$HOME/bin` в `PATH` внутри команды `replace` или же явно указывать путевое имя `overwrite`. С этого момента будет подразумеваться, что создаваемые программы находятся в каталоге `/usr/bin`; они предназначены для этого.

Упражнение 5.17. Почему `overwrite` не использует сигнал `0` в `trap`, чтобы удалять файлы при выходе? Подсказка: попробуйте ввести `DEL` во время выполнения следующей программы:

```
trap "echo exiting; exit 1" 0 2
sleep 10
```

□

Упражнение 5.18. Добавьте параметр `-v` для команды `replace`, чтобы выводить все изменившиеся строки в `/dev/tty`. Хорошая подсказка: `s/$left/$right/g$vflag`. □

Упражнение 5.19. Настройте `replace` так, чтобы она работала вне зависимости от символов строки замены. □

Упражнение 5.20. Может ли команда `replace` быть использована для замены переменной `i` на `index` везде в программе? Что можно изменить для того, чтобы удалось решить эту задачу? □

Упражнение 5.21. Подходит ли `replace` и достаточно ли она мощна для того, чтобы быть внесенной в каталог `/usr/bin`? Не лучше ли просто вводить соответствующие `sed` команды в случае необходимости? Почему? □

Упражнение 5.22 (сложное).

```
$ overwrite file 'who | sort'
```

не работает. Объясните, почему не работает, и исправьте. Подсказка: см. `eval` в `sh(1)`. Как предложенное решение повлияет на интерпретацию метасимволов в команде? □

5.6. Команда zap: уничтожение процесса по имени

Команда kill уничтожает только процессы, для которых указан идентификатор. Поэтому когда требуется уничтожить конкретный фоновый процесс, обычно приходится сначала запустить ps, чтобы определить идентификатор процесса, а потом ввести его без ошибок как аргумент для kill. Но глупо иметь одну программу, выводящую число, которое сразу же приходится вручную вводить в другую программу. Почему бы не написать программу, скажем, zap, чтобы автоматизировать эту работу?

Одна из причин заключается в том, что уничтожающий процесс опасен, и необходимо соблюдать осторожность, чтобы не уничтожить не тот процесс. В качестве защитной меры будем выполнять zap интерактивно и выбирать жертвы при помощи команды pick.

Краткое напоминание о pick: она выводит все свои аргументы по очереди и ждет ответа пользователя; если ответ y, то аргумент выводится (подробнее об этой команде будет рассказано в следующем разделе). Команда zap использует pick для того, чтобы проверить, являются ли процессы, выбранные по имени, теми, которые пользователь хочет уничтожить:

```
$ cat zap
# zap pattern: уничтожить процессы, соответствующие шаблону
# В этой версии ОШИБКА

PATH=/bin:/usr/bin

case $# in
0)  echo 'Usage: zap pattern' 1>&2; exit 1
esac

kill `pick \`ps -ag | grep "$*"\` | awk '{print $1}'`
$
```

Обратите внимание на вложенные кавычки, защищенные символами обратной косой черты. Программа awk выбирает указанный pick идентификатор процесса из выходных данных команды ps:

```
$ sleep 1000 &
22126
$ ps -ag
   PID TTY TIME CMD
...
 22126 0   0:00 sleep 1000
...
$ zap sleep
22126?
0? q                          Что происходит?
$
```

Проблема заключается в том, что вывод ps разбит на слова, и команда pick воспринимает их как отдельные аргументы вместо того, чтобы обрабатывать всю строку целиком. Оболочка обычно разделяет строку на аргументы, ориентируясь на пробелы, как в

```
for i in 1 2 3 4 5
```

В этой программе необходимо так отрегулировать разбиение строк на аргументы (осуществляемое оболочкой), чтобы соседние «слова» разделяли только символы новой строки.

Переменная оболочки IFS (internal field separator – внутренний разделитель полей) – это строка символов, которые разделяют слова на списки аргументов, такие как обратный апостроф или операторы for. Обычно IFS содержит пробел, знак табуляции и символ новой строки, но его можно и изменить, например, оставив только символ новой строки:

```
$ echo 'echo $#' >nargs
$ cx nargs
$ who
you      tty0    Oct  1 05:59
pjw      tty2    Oct  1 11:26
$ nargs `who`
10                          Десять полей, разделенных пробелами и
                            разделителями строк
$ IFS='
'
                            Только разделитель строк
$ nargs `who`
2                           Две строки, два поля
$
```

Если IFS включает в себя только разделитель строк, то команда zap работает правильно:

```
$ cat zap
# zap pat: уничтожить все процессы, соответствующие pat
# окончательная версия

PATH=/bin:/usr/bin
IFS='
'           # только разделитель строк
case $1 in
"") echo 'Usage: zap [-2] pattern' 1>&2; exit 1 ;;
-*) SIG=$1; shift
esac

echo '   PID TTY   TIME CMD'
kill $SIG `pick \`ps -ag | egrep "$*"\` | awk '{print $1}'`
$ ps -ag
   PID TTY TIME CMD
...
```

```
22126 0   0:00 sleep 1000
...
$ zap sleep
   PID TTY TIME CMD
 22126 0   0:00 sleep 1000? y
 23104 0   0:02 egrep sleep? n
$
```

В эту версию внесено еще два полезных изменения: добавлен необязательный аргумент для указания сигнала (обратите внимание на то, что если аргумент отсутствует, то SIG не будет определен и, следовательно, будет считаться пустой строкой), а вместо grep использована egrep – для того, чтобы можно было задавать более сложные шаблоны, как, например, 'sleep|date'. Команда echo печатает заголовки колонок для вывода ps.

Может показаться странным, что команда получила имя zap вместо того, чтобы называться просто kill. Основная причина заключается в том, что (в отличие от примера с командой cal) это не новая версия команды kill, ведь zap обязана быть интерактивной. К тому же нам хотелось сохранить название kill для настоящей команды. Еще одно замечание: zap чрезвычайно медленная команда – сказывается наличие всех дополнительных программ, при этом больше всего затрат приходится на команду ps (которая, так или иначе, но должна выполняться). В следующей главе будет предложена более производительная реализация.

Упражнение 5.23. Измените zap таким образом, чтобы выводить заголовок ps из конвейера так, чтобы он был не чувствителен к изменениям формата вывода ps. Насколько это усложнит программу?

5.7. Команда pick: пробелы и аргументы

Практически все, что потребуется для написания команды pick в оболочке, уже было описано ранее. Единственное, с чем вы еще не знакомы, – это механизм чтения пользовательского ввода. Встроенная в оболочку функция read считывает только одну строку текста с устройства стандартного ввода и присваивает полученный текст (без символа новой строки) указанной переменной:

```
$ read greeting
hello, world                    Ввести новое значение для greeting
$ echo $greeting
hello, world
$
```

Функция read чаще всего используется в .profile для настройки окружения при входе в систему, главным образом для задания переменных оболочки, таких как TERM.

Читать данные `read` может только со стандартного ввода; ее даже нельзя перенаправить. Ни одна из встроенных в оболочку команд не может быть перенаправлена (в отличие от операторов управления типа `for`) при помощи `>` или `<`:

```
$ read greeting </etc/passwd
goodbye                  В любом случае надо ввести значение
illegal io               Оболочка сообщает об ошибке
$ echo $greeting
goodbye                  greeting имеет введенное значение,
                         а не значение из файла
$
```

Можно было бы назвать такое положение вещей ошибкой в оболочке, но такова реальность. К счастью, обычно можно решить проблему, перенаправив цикл, в котором находится `read`. Это будет ключевым моментом реализации команды `pick`:

```
# pick:  выбор аргументов

PATH=/bin:/usr/bin

for i               # для каждого аргумента
do
    echo -n "$i? " >/dev/tty
    read response
    case $response in
    y*) echo $i ;;
    q*) break
    esac
done </dev/tty
```

Команда `echo -n` уничтожает заключительный символ новой строки, поэтому ответ может быть введен на той же строке, где находится приглашение на ввод. И, конечно же, приглашения на ввод выводятся в `/dev/tty`, поскольку устройство стандартного вывода – это почти наверняка не терминал.

Оператор `break` позаимствован из Си: он заканчивает ближайший из вложенных циклов. В данном случае он обеспечивает выход из цикла `for` при вводе `q`. Выбор завершается при вводе `q` – это простой, доступный способ, совместимый с другими программами.

Интересно поэкспериментировать с пробелами в аргументах команды `pick`:

```
$ pick '1 2' 3
1 2?
3?
$
```

Если вы хотите посмотреть, как `pick` читает свои аргументы, запустите ее и нажимайте *Return* после каждого приглашения на ввод. В таком

виде все работает хорошо: `for i` правильно обрабатывает аргументы. Цикл можно было бы записать и по-другому:

```
$ grep for pick                    Посмотрим, что делает эта версия
for i in $*
$ pick '1 2' 3
1?
2?
3?
$
```

Эта форма не работает, так как операнды цикла просматриваются повторно, и первый аргумент из-за наличия пробелов превращается в два. Попробуем заключить `$*` в кавычки:

```
$ grep for pick                    Попробуем другую версию
for i in "$*"
$ pick '1 2' 3
1 2 3?
$
```

Тоже не работает, потому что `"$*"` – это одно слово, образованное из всех аргументов, собранных вместе и разделенных пробелами.

Естественно, решение существует, но оно почти что из области черной магии: строка `"$@"` интерпретируется оболочкой особым образом и преобразуется в точности в аргументы командного файла:

```
$ grep for pick                    Испытаем третью версию
for i in "$@"
$ pick '1 2' 3
1 2?
3?
$
```

Если строка `$@` не заключена в кавычки, то она идентична `$*`; особенности проявляются только при наличии двойных кавычек. В программе `overwrite` `$@` применялась для защиты аргументов пользовательской команды.

Подводим итоги:

- `$*` и `$@` преобразуются в аргументы и повторно просматриваются; из-за пробелов, имеющихся в аргументе, он превращается в несколько аргументов.
- `"$*"` представляет собой единое слово, составленное из всех аргументов командного файла, соединенных вместе, с пробелами.
- `for i` ничем не отличается от `for i in "$@"`; пробелы в аргументах не обрабатываются особым образом, результатом является список слов, совпадающих с исходными аргументами.

Если аргументы для команды pick не заданы, то, вероятно, она должна читать свой стандартный ввод, так что можно было написать

```
$ pick <mailinglist
```

вместо

```
$ pick `cat mailinglist`
```

Но не будем исследовать эту версию pick: в ней возникают некоторые проблемы, и она гораздо сложнее, чем та же самая программа, написанная на Си, которая будет представлена в следующей главе.

Первые два из предложенных ниже упражнений сложные, но они могут кое-чему научить продвинутых программистов оболочки.

Упражнение 5.24. Попробуйте написать команду pick, которая считывала бы аргументы с устройства стандартного ввода, не указанные в командной строке. Она должна правильно обрабатывать пробелы. Работает ли ответ q? Если нет, попробуйте выполнить следующее упражнение. □

Упражнение 5.25. Несмотря на то что встроенные функции оболочки (как read и set) не могут быть перенаправлены, сама оболочка может быть временно перенаправлена. Прочитайте раздел sh(1), в котором описывается exec, и подумайте, как read могла бы читать с /dev/tty без вызова подоболочки. (Может быть, стоит сначала прочитать главу 7.) □

Упражнение 5.26 (гораздо более легкое). Используйте read в своем .profile для инициализации TERM и всего от нее зависящего, как, например, шагов табуляции. □

5.8. Команда news: служебные сообщения

В главе 1 рассказывалось о том, что в вашей системе может присутствовать команда news, передающая сообщения, представляющие интерес для всех пользователей системы. Называться такая команда может и по-другому, у каждой реализации могут быть и свои особенности, но, в том или ином виде, она есть в большинстве систем. Команда news, которая будет создаваться сейчас, не призвана заменить вашу локальную команду, просто на ее примере будет показано, насколько просто написать такую команду в оболочке. Потом интересно будет сравнить версию, разработанную нами, с локальной версией.

Основной идеей программ такого рода является хранение персональных новостей в специальном каталоге (по одной в файле) /usr/news. Тогда news (это и есть рассматриваемая программа news) сравнивает время изменения файлов в /usr/news со временем изменения файла в домашнем каталоге (.news_time), который служит временной меткой. При отладке можно использовать «.» как каталог и для файлов новос-

тей и для .news_time; когда же программа будет готова к работе, его можно будет заменить на /usr/news.

```
$ cat news
# news: выводить файлы новостей, версия 1

HOME=.      # только для отладки
cd .        # символ-заполнитель для /usr/news
for i in `ls -t * $HOME/.news_time`
do
    case $i in
    */.news_time)   break ;;
    *)      echo news: $i
    esac
done
touch $HOME/.news_time
$ touch .news_time
$ touch x
$ touch y
$ news
news: y
news: x
$
```

Команда touch изменяет время модифицирования файла, являющегося ее аргументом, на текущее время, не изменяя сам файл. Для отладки выведены только имена файлов, а не их содержимое. Цикл завершается, когда доходит до .news_time, поэтому перечисляются только более новые файлы. Обратите внимание, что * в операторах case может соответствовать /, что невозможно в шаблонах имен файлов.

Что произойдет, если .news_time не существует?

```
$ rm .news_time
$ news
$
```

Такое неожиданное молчание – ошибка. Это случилось потому, что ls не нашла файл, и она сообщает о возникшей проблеме на свое устройство стандартного вывода, прежде чем вывести какую бы то ни было информацию о существующих файлах. Ошибка несомненна – диагностика должна выводиться на устройство стандартного вывода ошибок, – но ее можно обойти, распознав имеющуюся проблему в цикле и перенаправив вывод стандартных ошибок на стандартный вывод, чтобы все версии работали одинаково. (В последних версиях системы эта ошибка была исправлена, мы оставили ее для того, чтобы показать, как можно справиться с небольшими неполадками.)

```
$ cat news
# news: выводить файлы новостей, версия 2

HOME=.      # только для отладки
cd .        # символ-заполнитель для /usr/news
```

```
IFS='
'           # только символ разделителя строк
for i in  `ls -t * $HOME/.news_time 2>&1`
do
    case $i in
    *' not found')  ;;
    */.news_time)   break ;;
    *)      echo news: $i ;;
    esac
done
touch $HOME/.news_time
$ rm .news_time
$ news
news: news
news: y
news: x
$
```

Переменная IFS содержит только разделитель строк, поэтому сообщение

```
./.news_time not found
```

не разбирается на три слова.

После этого news должна выводить файлы новостей, а не показывать их имена. Полезно знать, кто отправил сообщение и когда, поэтому будем использовать команду set и ls -l для того, чтобы печатать перед сообщением заголовок:

```
$ ls -l news
-rwxrwxrwx 1 you         208 Oct  1 12:05 news
$ set `ls -l news`
-rwxrwxrwx: bad option(s)                    Что-то не так!
$
```

Вот пример того, как проявляется единообразие программ и данных в оболочке. Команда set выражает недовольство, потому что ее аргумент (-rwxrwxrwx) начинается со знака минус, напоминая тем самым параметр. Простой (хотя и некрасивый) способ разрешения ситуации состоит в том, чтобы предварить аргумент обычным символом:

```
$ set X`ls -l news`
$ echo "news: ($3) $5 $6 $7"
news: (you) Oct 1 12:05
$
```

Формат вывода удобен: наряду с именем файла указывается автор и дата сообщения.

Готова окончательная версия команды news:

```
# news: выводить файлы новостей, окончательная версия

PATH=/bin:/usr/bin
```

```
IFS='
'           # только разделитель строк
cd /usr/news

for i in `ls -t * $HOME/.news_time 2>&1`
do
    IFS=' '
    case $i in
    *' not found')  ;;
    */.news_time)   break ;;
    *)  set X`ls -l $i`
        echo "
$i: ($3) $5 $6 $7
"
        cat $i
    esac
done
touch $HOME/.news_time
```

Дополнительные символы новой строки в заголовке разделяют сообщения по мере печати. Первоначально значение `IFS` – это символ новой строки, поэтому сообщение `not found` (если оно выводится) первой команды `ls` обрабатывается как единый аргумент. При втором присваивании значением `IFS` становится пробел, поэтому вывод второй `ls` разбивается на множество аргументов.

Упражнение 5.27. Добавьте параметр `-n` (notify – уведомить) для команды `news`, чтобы извещать о новых сообщениях, но не выводить их и не обращаться к `.news_time`. Такую команду можно добавить в `.profile`. □

Упражнение 5.28. Сравните принцип работы и реализацию команды `news`, предложенные в этой книге, с соответствующей командой вашей системы. □

5.9. Отслеживание изменений файла: get и put

В этом разделе, завершающем длинную главу, будет обсуждаться большой и более сложный, чем рассмотренные ранее, пример, иллюстрирующий взаимодействие оболочки с программами `awk` и `sed`.

Программа развивается по мере исправления ошибок и добавления новых возможностей. Бывает удобно отслеживать эти версии, особенно если с ними работают на других машинах. Пользователи вполне могут спросить: «Что изменилось с тех пор, как мы получили эту версию?» или: «Как исправлена такая-то ошибка?» К тому же при наличии резервной копии применение новых идей будет более безопасным: если что-то не заработает, можно безболезненно вернуться к исходной программе.

В качестве одного из решений можно предложить хранить копии всех версий, но это тяжело организовать, к тому же требуется много диско-

вого пространства. Вместо этого извлечем выгоду из такого предположения: у следующих друг за другом версий должно быть очень много общего, которое можно сохранить только единожды. Команда `diff -e`

```
$ diff -e old new
```

генерирует список команд `ed`, которые преобразуют `old` в `new`. Таким образом, можно хранить все версии файла в одном (отдельном) файле, сохраняя одну полную версию и набор команд редактора, преобразующих ее в любую другую версию.

Два очевидных решения: хранить самую последнюю версию и набор команд, «переводящих время назад», или же хранить самую первую версию и набор команд, «переводящих часы вперед». Хотя второй вариант легче запрограммировать, первый работает быстрее, если версий много – ведь обычно интерес вызывают более поздние версии.

Итак, остановимся на первом варианте. В едином файле, назовем его *файлом предыстории* (*history file*), находится текущая версия и наборы редактирующих команд, которые превращают каждую версию в предыдущую (то есть в ближайшую более старую). Каждый набор команд редактора начинается со строки, выглядящей следующим образом:

```
@@@ пользователь дата сводка
```

Сводка – это одна строка, описывающая изменения, предоставленная *пользователем*.

Есть две команды для хранения версий: `get` извлекает версию из файла предыстории, а `put` вносит новую версию в этот файл, предварительно запросив однострочное сообщение об изменениях.

Прежде чем представить программу, приведем пример, показывающий, как работают `get` и `put` и как хранятся версии:

```
$ echo a line of text >junk
$ put junk
Summary: make a new file              Введите описание
get: no file junk.H                   Истории не существует...
put: creating junk.H                  ... поэтому put создает ее
$ cat junk.H
a line of text
@@@ you Sat Oct  1 13:31:03 EDT 1983 make a new file
$ echo another line >>junk
$ put junk
Summary: one line added
$ cat junk.H
a line of text
another line
@@@ you Sat Oct  1 13:32:28 EDT 1983 one line added
2d
@@@ you Sat Oct  1 13:31:03 EDT 1983 make a new file
$
```

«Набор редактирующих команд» состоит из одной-единственной строки `2d`, которая удаляет строку 2 в файле, превращая новую версию в исходную.

```
$ rm junk
$ get junk                                  Самая новая версия
$ cat junk
a line of text
another line
$ get -1 junk
$ cat junk                  Предыдущая перед самой новой версией
a line of text
$ get junk                                  Снова самая новая версия
$ replace another 'a different' junk        Изменить ее
$ put junk
Summary: second line changed
$ cat junk.H
a line of text
a different line
@@@ you Sat Oct  1 13:34:07 EDT 1983 second line changed
2c
another line
.
@@@ you Sat Oct  1 13:32:28 EDT 1983 one line added
2d
@@@ you Sat Oct  1 13:31:03 EDT 1983 make a new file
$
```

Редактирующие команды, извлекающие нужную версию, прогоняются по файлу сверху вниз: первый набор преобразует новейшую версию во вторую по свежести, следующий набор превращает вторую в третью, и т. д. Поэтому на самом деле преобразование нового файла в старый при выполнении команд `ed` происходит постепенно, по одной версии за раз.

Трудности могут возникнуть, если в файле есть строки, начинающиеся с трех символов `@`, а раздел BUGS команды `diff(1)` предупреждает о строках, которые содержат только точку. Именно потому, что обозначение `@@@` вряд ли может встретиться в тексте, оно было выбрано для маркировки команд редактирования.

Несмотря на то что было бы поучительно показать, как развивались команды `get` и `put`, они достаточно длинные и описание их различных форм заняло бы слишком много времени. Поэтому представлены будут только их окончательные версии. Команда `put` проще:

```
# put:  ввести файл в историю

PATH=/bin:/usr/bin

case $# in
    1)  HIST=$1.H ;;
    *)  echo 'Usage: put file' 1>&2; exit 1 ;;
```

```
esac
if test ! -r $1
then
    echo "put: can't open $1" 1>&2
    exit 1
fi
trap 'rm -f /tmp/put.[ab]$$; exit 1' 1 2 15
echo -n 'Summary: '
read Summary

if get -o /tmp/put.a$$ $1       # предыдущая версия
then                            # объединить части вместе
    cp $1 /tmp/put.b$$          # текущая версия
    echo "@@@ `getname` `date` $Summary" >>/tmp/put.b$$
    diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$   # последние отличия
    sed -n '/^@@@/,$p' <$HIST >>/tmp/put.b$$ # старые отличия
    overwrite $HIST cat /tmp/put.b$$         # поместить обратно
else                            # создать новый
    echo "put: creating $HIST"
    cp $1 $HIST
    echo "@@@ `getname` `date` $Summary" >>$HIST
fi
rm -f /tmp/put.[ab]$$
```

После считывания однострочной сводки изменений `put` вызывает `get` для того, чтобы извлечь предыдущую версию файла из файла предыстории. Параметр `-o` команды `get` указывает альтернативный файл для вывода. Если `get` не находит файла предыстории, она возвращает код ошибки, и `put` создает новый файл предыстории. Если же файл существует, то оператор `then` создает новую историю во временном файле из (по порядку) самой новой версии, строки `@@@`, команд редактора, преобразующих новейшую версию в ближайшую предыдущую, старых команд редактора и строк `@@@`. В заключение временный файл копируется в файл предыстории посредством команды `overwrite`.

Команда `get` сложнее, чем `put`, в основном потому, что у нее есть параметры.

```
# get: извлечь файл из истории

PATH=/bin:/usr/bin

VERSION=0
while test "$1" != ""
do
    case "$1" in
        -i) INPUT=$2; shift ;;
        -o) OUTPUT=$2; shift ;;
        -[0-9]) VERSION=$1 ;;
        -*) echo "get: Unknown argument $i" 1>&2; exit 1 ;;
        *)  case "$OUTPUT" in
            "") OUTPUT=$1 ;;
```

```
            *)  INPUT=$1.H ;;
            esac
    esac
    shift
done
OUTPUT=${OUTPUT?"Usage: get [-o outfile] [-i file.H] file"}
INPUT=${INPUT-$OUTPUT.H}
test -r $INPUT || { echo "get: no file $INPUT" 1>&2; exit 1; }
trap 'rm -f /tmp/get.[ab]$$; exit 1' 1 2 15
# разбить на текущую версию и команды редактирования
sed <$INPUT -n '1,/^@@@/w /tmp/get.a'$$'
               /^@@@/,$w /tmp/get.b'$$
# выполнить редактирование
awk </tmp/get.b$$ '
    /^@@@/  { count++ }
    !/^@@@/ && count > 0 && count <= - '$VERSION'
    END { print "$d"; print "w", "'$OUTPUT'" }
' | ed - /tmp/get.a$$
rm -f /tmp/get.[ab]$$
```

Параметры достаточно стандартные: `-i` и `-o` указывают альтернативный ввод и вывод; `-[0-9]` выбирают конкретную версию, при этом `0` — самая новая (значение по умолчанию), `-1` — версия на одну старше самой новой, и т. д. Аргументы обрабатываются в цикле `while`, содержащем команды `test` и `shift`. Выбран цикл `while`, а не `for`, потому что некоторые параметры (`-i`, `-o`) поглощают следующий аргумент, вот почему необходимо применение `shift`, а циклы `for` и команды `shift` не в состоянии взаимодействовать должным образом, если `shift` находится внутри `for`. Параметр `ed` «-» отключает подсчет символов, который обычно сопровождает чтение или запись файла.

Строка

```
test -r $INPUT || {echo "get: no file $INPUT" 1>&2; exit 1;}
```

эквивалентна

```
if test ! -r $INPUT
then
        echo "get: no file $INPUT" 1>&2
        exit 1
fi
```

(то есть форме, которая была использована в `put`), но она короче и понятнее для программистов, которые знакомы с оператором `||`. Команды, находящиеся внутри фигурных скобок, выполняются в текущей оболочке, а не в подоболочке; в данном случае это необходимо, чтобы выход по `exit` осуществлялся из `get`, а не из подоболочки. Символы `{` и `}` подобны `do` и `done` — они имеют особый смысл, только если перед ними стоит точка с запятой, разделитель строк или другой символ конца команды.

Наконец мы добрались до кода, который и выполняет собственно всю работу. Сначала sed разбивает файл предыстории на две части: самую свежую версию и набор команд редактирования. Затем программа awk обрабатывает команды редактирования. Строки @@@ подсчитываются (но не выводятся), и до тех пор, пока их количество не превышает номер искомой версии, команды редактирования пропускаются (вспомните, что awk по умолчанию выводит строку ввода). Две команды добавляются после команд исторического файла: $d удаляет отдельную строку @@@, которую sed оставила для текущей версии, а команда w записывает файл в место его окончательного расположения. Необходимо использовать overwrite, потому что команда get изменяет только версию файла, а не файл истории.

Упражнение 5.29. Напишите команду version, которая делала бы следующее:

```
$ version -5 file
```

выводила бы сводку изменений, дату изменения и пользователя, который произвел изменение выбранной версии в файле предыстории, а

```
$ version sep 20 file
```

сообщала бы, какая версия была текущей 20 сентября. Это было бы удобно в:

```
$ get `version sep 20 file`
```

(Для удобства version может показывать имя файла предыстории.) □

Упражнение 5.30. Измените get и put так, чтобы они обрабатывали файл предыстории в отдельном каталоге вместо того, чтобы загромождать рабочий каталог файлами вида .H. □

Упражнение 5.31. Не все версии файла стоит хранить после того, как получена хорошая версия. Как удалить версию из середины файла истории? □

5.10. Оглянемся назад

Когда надо написать новую программу, то первое, о чем начинаешь думать – это как написать ее на своем любимом языке программирования. В нашем случае этим языком обычно является оболочка.

Несмотря на то что ее синтаксис не совсем привычен, оболочка – это превосходный язык программирования, язык высокого уровня: операторами являются целые программы. Так как оболочка интерактивна, то программы можно разрабатывать в интерактивном режиме и совершенствовать их постепенно, пока не заработают. Затем, если программы предназначены не только для себя, можно «навести на них глянец» и доработать для сообщества пользователей. В тех очень редких

случаях, когда программа оболочки оказывается малопроизводительной, можно переписать ее на Си, имея под рукой проверенный работающий прототип. (Этот способ будет несколько раз применен в следующей главе.)

Такой обобщающий подход характерен для среды программирования UNIX – здесь принято использовать то, что уже сделано другими вместо того, чтобы опять начинать с нуля, начать с чего-то маленького и усовершенствовать его; применять для обкатки новых идей различные инструментальные средства.

В данной главе было представлено множество примеров, которые легко выполнить, используя существующие программы и оболочку. Иногда достаточно просто перегруппировать аргументы, как в случае с `cal`. Иногда оболочка предоставляет цикл для просмотра множества имен файлов или последовательности команд, например в `watchfor` и `checkmail`. Даже сложные примеры все равно требуют меньше работы, чем при программировании на Си; например, 20-строчная версия программы `news` заменяет версию из 350 (!) строк на Си.

Но недостаточно просто наличия программируемого командного языка. Недостаточно и большого набора команд. Главное в том, чтобы все компоненты *работали вместе*. Все они придерживаются общих соглашений, касающихся представления информации и обмена ею. Каждая программа разработана так, что все сконцентрировано на выполнении одного задания, причем на хорошем выполнении. Оболочка связывает программы вместе, делает это быстро и эффективно, какую бы идею вы ни реализовывали. Именно благодаря такому взаимодействию программная среда UNIX очень продуктивна.

История и библиография

Идея команд `get` и `put` взята из системы SCCS (Source Code Control System – система управления исходными текстами), созданной Марком Рочкиндом (Marc Rochkind) и описанной им в статье «The source code control system» (Система управления исходными текстами), появившейся в журнале «IEEE Trans. on Software Engineering» в 1975 году. SCCS является гораздо более мощной и гибкой, чем простые программы, описанные в этой главе; она предназначена для хранения больших программ в производственной среде. Однако основа SCCS – это все та же программа `diff`.

6

Программирование с использованием стандартного ввода-вывода

До сих пор для создания новых инструментов применялись уже существующие, но надо сказать, практически все, что можно сделать при помощи оболочки, `sed` и `awk`, уже сделано. В этой главе мы напишем несколько программ на языке программирования Си. В дальнейшем при обсуждении и разработке программного дизайна также будет доминировать основополагающая философия создания программ для совместной работы – мы хотим создавать инструменты, с помощью которых другие смогут создавать собственные инструменты. Для каждого примера будет рассмотрена стратегия реализации, представляющаяся авторам наиболее разумной: программа начинается с абсолютного минимума, выполняющего какую-нибудь полезную функцию, затем по мере необходимости к нему добавляются различные свойства и возможности.

Есть веские основания писать программы с нуля. Может случиться так, что поставленную задачу просто невозможно решить с помощью существующих программ. Так часто бывает с программами, которые должны обрабатывать нетекстовые файлы. Надо сказать, что большая часть программ, представленных ранее, успешно работает только с текстовой информацией. Если же решение, использующее только средства оболочки и другие универсальные средства, существует, оно может быть недостаточно надежным и производительным. В этом случае версия, написанная в оболочке, хорошо подходит для уточнения спецификации и пользовательского интерфейса. (И если она работает достаточно хорошо, нет смысла переделывать программу.) Характерным примером является команда `zap` из предыдущей главы: создание первой версии в оболочке заняло всего несколько минут, окончатель-

ная версия предоставляет вполне пригодный пользовательский интерфейс, но работает программа слишком медленно.

Выбран именно язык Си, потому что это стандартный язык систем UNIX – ядро и все пользовательские программы написаны на Си, и, если честно, остальные языки далеко не так хорошо поддерживаются. Предполагается, что вы знаете Си, по крайней мере настолько, чтобы понять написанное. Если это не так, прочтите книгу «The C Programming Language» Б. В. Кернигана (B. W. Kernighan) и Д. М. Ритчи (D. M. Ritchie), изданную в Prentice-Hall в 1978 году.[1]

Будет использоваться библиотека стандартного ввода-вывода – набор функций, обеспечивающих эффективный и переносимый ввод-вывод и системные сервисы для программ на Си. Библиотека стандартного ввода-вывода доступна на большинстве не-UNIX-систем, поддерживающих Си, так что программы, которые ограничивают взаимодействие с системой ее возможностями, легко переносимы.

Примеры, представленные в этой главе, обладают общим свойством: все они представляют собой небольшие сервисные программы, которые используются каждый день, но при этом они не вошли в седьмую версию. Если в вашей системе есть подобные программы, полезно сравнить предложенную реализацию с уже имеющейся.

Если же вы впервые увидите программы в этой книге, то можете, как и мы, обнаружить их полезность. В любом случае эти программы помогут понять следующее: не существует совершенной системы, но, применив совсем небольшое усилие, можно улучшить положение дел и устранить дефекты.

6.1. Стандартный ввод и вывод: vis

Многие программы читают только с одного входа и пишут на один выход; для таких программ вполне достаточно устройств стандартного ввода-вывода, и этого почти всегда хватает для начала.

Рассмотрим программу `vis`, которая копирует стандартный ввод на стандартный вывод, при этом все непечатаемые символы выводятся в виде *nnn*, где *nnn* – восьмеричное значение символа. Эта программа незаменима для обнаружения странных или нежелательных символов, попавших в файлы. Например, каждый символ возврата на одну позицию `vis` выведет как `\010` (это восьмеричное значение символа):

```
$ cat x
a b c
$ vis <x
```

[1] Керниган Б., Ритчи Д. «Язык программирования Си», 3-е издание, Невский Диалект, 2002.

```
abc\010\010\010___
$
```

Для того чтобы сканировать несколько файлов при помощи этой рудиментарной версии `vis`, объедините файлы командой `cat`:

```
$ cat file1 file2 ... | vis
...
$ cat file1 file2 ... | vis | grep '\\'
...
```

Таким образом, можно не беспокоиться о доступе к файлам из программы.

Кстати, может показаться, что эту задачу решит и `sed`, ведь команда `l` выводит непечатаемые символы в понятном виде:

```
$ sed -n l x
abc←←←___
$
```

Вероятно, вывод `sed` выглядит даже понятнее, чем вывод `vis`. Но дело в том, что `sed` предназначена только для обработки текстовых файлов:

```
$ sed -n l /usr/you/bin
$                    Абсолютно ничего!
```

(Так было на PDP-11; что касается системы на VAX, там выполнение `sed` прерывалось, видимо из-за того, что входные данные интерпретировались как слишком длинная текстовая строка). Так что `sed` использовать нельзя, поэтому направим свои усилия на создание новой программы.

Самыми простыми подпрограммами ввода и вывода являются `getchar` и `putchar`. Каждый вызов `getchar` извлекает следующий символ из стандартного ввода, которым может быть файл, конвейер или терминал (по умолчанию) – программа этого не знает. Аналогичным образом `putchar(c)` помещает символ `c` в стандартный вывод, по умолчанию это также терминал.

Функция `printf(3)` осуществляет преобразование формата вывода. Порядок вызова `printf` и `putchar` произволен; выходные данные будут появляться в порядке обращения к подпрограммам. Есть и соответствующая функция `scanf(3)` для преобразования формата входных данных; она считывает ввод и разбивает его на строки, числа и т. д., как указано. Порядок вызова `scanf` и `getchar` также произволен.

Приведем первую версию `vis`:

```
/* vis: сделать видимыми странные символы (версия 1) */

#include <stdio.h>
#include <ctype.h>

main()
{
```

```
    int c;

    while ((c = getchar()) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else
            printf("\\%03o", c);
    exit(0);
}
```

Функция `getchar` возвращает следующий байт ввода или значение `EOF`, если достигнут конец файла (или ошибку). Вспомните, что `EOF` – это *не* байт файла (см. главу 2). Значение `EOF` гарантированно отличается от любого значения, которое может содержать отдельный байт, поэтому его можно отличить от настоящих данных; переменная `c` объявлена как `int`, а не `char`, так что она сможет содержать значение `EOF`. Строка

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

должна присутствовать в начале каждого исходного файла. Она заставляет компилятор Си читать *заголовочный файл* (`/usr/include/stdio.h`) стандартных функций и символов, который содержит описание `EOF`. Далее в тексте будет встречаться краткая запись имени файла – `<stdio.h>`.

Файл `<ctype.h>` – это еще один заголовочный файл, который описывает машинонезависимые тесты, определяющие свойства символов. В первой версии `vis` использованы `isascii` и `isprint`, – они определяют, принадлежит ли введенный символ набору ASCII (значение меньше `0200`) и является ли он печатаемым; остальные тесты перечислены в табл. 6.1. Обратите внимание на то, что символ новой строки, знак табуляции и пробел не являются «печатаемыми» по определениям `<ctype.h>`.

Таблица 6.1. Макросы проверки символов <ctype.h>

Имя	Что проверяет
`isalpha(c)`	буква: `a-z A-Z`
`isupper(c)`	верхний регистр: `A-Z`
`islower(c)`	нижний регистр: `a-z`
`isdigit(c)`	цифра: `0-9`
`isxdigit(c)`	шестнадцатеричная цифра: `0-9 a-f A-F`
`isalnum(c)`	буква или цифра
`isspace(c)`	пробел, табуляция, символ новой строки, вертикальная табуляция, перевод страницы, возврат каретки
`ispunct(c)`	не алфавитно-цифровой, не управляющий и не пробельный символ

Имя	Что проверяет
isprint(c)	печатаемый символ: любой графический
iscntrl(c)	управляющие символы: 0 <= c < 040 || c == 0177
isascii(c)	ASCII-символ: 0 <= c <= 0177

Вызов exit в конце vis не обязателен для корректного исполнения программы, но благодаря ему любой пользователь, запустивший vis, увидит код нормального завершения (обычно ноль) по завершении программы. Есть и другой способ получить код завершения – выход из main посредством return 0; тогда кодом завершения программы является величина, возвращаемая main. Если return или exit не присутствуют в программе явно, то код завершения предсказать невозможно.

Чтобы откомпилировать программу, написанную на Си, поместите исходный текст в файл с именем, оканчивающимся на .c, например vis.c, скомпилируйте ее при помощи cc – компилятор поместит результат в файл a.out (a означает ассемблер), теперь запустите его:

```
$ cc vis.c
$ a.out
hello worldctl-g
hello world\007
ctl-d
$
```

Можно переименовать a.out, когда он заработает, а можно указать параметр -o команды cc:

```
$ cc -o vis vis.c          Вывод в vis, а не в a.out
```

Упражнение 6.1. Было решено, что знаки табуляции трогать не надо (то есть не представлять их как \011 или >, или же \t), так как главной задачей программы vis является поиск по-настоящему аномальных символов. Можно предложить и другую модель – однозначно идентифицировать каждый символ ввода: знаки табуляции, пробелы в конце строки, неграфические символы и т. д. Измените vis так, чтобы для таких символов, как табуляция, обратная косая черта, возврат на одну позицию, разрыв страницы и т. д., выводилось их условное Си-представление: \t, \\, \b, \f и т. д., а пробелы в конце строк помечались. Можно ли сделать это однозначно? Сравните то, что получится, с командой sed,

```
$ sed -n l
```

□

Упражнение 6.2. Измените vis так, чтобы она свертывала длинные строки до некоторой разумной длины. Как это повлияет на однозначность, упомянутую в предыдущем упражнении? □

6.2. Аргументы программы: vis, версия 2

Когда программа, написанная на Си, выполняется, аргументы командной строки становятся доступны функции `main` как количество `argc` и массив указателей на символьные строки, содержащие аргументы, `argv`. Принято соглашение о том, что `argv[0]` – это имя самой команды, так что `argc` всегда больше 0; «полезными» же являются аргументы `argv[1] ...argv[argc-1]`. Вам уже известно, что перенаправления ввода и вывода < и > осуществляются оболочкой, а не отдельными программами, поэтому перенаправление не влияет на количество аргументов, которые видит программа.

Чтобы пояснить, как происходит обработка аргументов, давайте изменим программу `vis`, добавив необязательный аргумент: `vis -s` удаляет все непечатаемые символы вместо того, чтобы показывать их условное видимое представление. Этот параметр используется для очистки файлов из других систем, например, применяющих в конце строки CRLF (carriage return and line feed – возврат каретки и перевод строки) вместо символа новой строки.

```
/* vis: сделать видимыми странные символы (версия 2) */

#include <stdio.h>
#include <ctype.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    int c, strip = 0;

    if (argc > 1 && strcmp(argv[1], "-s") == 0)
        strip = 1;
    while ((c = getchar()) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else if (!strip)
            printf("\\%03o", c);
    exit(0);
}
```

Упомянутый ранее `argv` – это указатель на массив, элементами которого являются указатели на массивы символов; каждый массив завершается ASCII-символом `NUL` (`'\0'`), поэтому можно рассматривать такой массив как строку. Эта версия `vis` начинается с проверки наличия аргументов, если же аргумент задан, то он сравнивается с `-s` (неправильные аргументы игнорируются). Функция `strcmp(3)` сравнивает две строки и в случае совпадения возвращает ноль.

В табл. 6.2 приведен перечень функций обработки строк и просто полезных функций, одной из которых является `strcmp`. Обычно лучше

предпочесть эти функции, а не писать собственные, потому что они стандартны, отлажены и во многих случаях работают быстрее, чем то, что можно написать самостоятельно, так как они были оптимизированы для конкретных машин (некоторые из них написаны на Ассемблере).

Упражнение 6.3. Измените аргумент `-s` так, чтобы `vis -s`*n* печатала только строки из *n* и более последовательных печатаемых символов, опуская непечатаемые символы и короткие последовательности печатаемых. Таким образом можно выделить текстовые части нетекстовых файлов (например, исполняемых программ). Некоторые версии системы содержат программу `strings`, которая реализует такое решение. Что удобнее – специальная программа или аргумент для `vis`? □

Упражнение 6.4. Одной из сильных сторон системы UNIX является доступность исходного текста Си – код иллюстрирует красивые решения многих задач программирования. Выскажите свое мнение относительно альтернативы: удобочитаемый код Си или возможная в некоторых случаях оптимизация за счет переписывания программы на Ассемблере. □

Таблица 6.2. Стандартные функции обработки строк

Функция	Действие
`strcat(s,t)`	присоединяет строку `t` в конец строки `s`; возвращает `s`
`strncat(s,t,n)`	присоединяет не более чем `n` символов `t` в конец `s`
`strcpy(s,t)`	копирует `t` в `s`; возвращает `s`
`strncpy(s,t,n)`	копирует ровно `n` символов; дополняет символом `null` при необходимости
`strcmp(s,t)`	сравнивает `s` и `t`, возвращает `<0`, `0`, `>0` для `<`, `==`, `>`
`strncmp(s,t,n)`	сравнивает не более `n` символов
`strlen(s))`	возвращает длину строки `s`
`strchr(s,c)`	возвращает указатель к первому `c` в `s`, `NULL`, если его нет
`strrchr(s,c)`	возвращает указатель к последнему `c` в `s`, `NULL`, если его нет. В старых системах это были функции `index` и `rindex`
`atoi(s)`	возвращает целочисленное представление `s`
`atof(s)`	возвращает значение `s` с плавающей точкой; требует объявления `double atof()`
`malloc(n)`	возвращает указатель на `n` байт памяти, `NULL`, если не может
`calloc(n,m)`	возвращает указатель на `n×m` байт памяти, заполняет байтом `0`, возвращает `NULL` при неудаче, `malloc` и `calloc` возвращают `char *`[a]
`free(p)`	освобождает память, распределенную `malloc` или `calloc`

[a] В современных реализациях Си `malloc` и `calloc` возвращают `void*`. – *Примеч. науч. ред.*

6.3. Доступ к файлам: vis, версия 3

Две первые версии программы vis читали только стандартный ввод и записывали выходные данные только в стандартный вывод, при этом и ввод и вывод были унаследованы от оболочки. Пришло время сделать следующий шаг – изменить vis так, чтобы она могла обратиться к файлу по его имени, тогда

```
$ vis file1 file2 ...
```

просматривала бы указанные файлы, а не устройство стандартного ввода. Если же имена файлов не указаны, то vis должна поступать, как раньше, то есть читать свой стандартный ввод.

Вопрос в том, как прочитать файлы, то есть как связать имена файлов с операторами ввода, которые и осуществляют считывание данных.

Есть несколько простых правил. Прежде чем файл можно будет прочитать или записать в него данные, его надо *открыть* при помощи стандартной библиотечной функции fopen. Функция fopen получает имя файла (например, temp или /etc/passwd), выполняет некоторые служебные действия и согласования с ядром, после чего возвращает внутреннее имя, которое и будет использоваться при дальнейших операциях с файлом.

Это внутреннее имя на самом деле является указателем (*указатель файла*) на структуру, содержащую информацию о файле, такую как местоположение буфера, позиция текущего символа в буфере, читается файл или записывается и т. д. Одно из описаний, полученных благодаря включению <stdio.h>, относится к структуре под названием FILE. Объявление указателя файла выглядит следующим образом:

```
FILE *fp;
```

Такая запись означает, что fp представляет собой указатель на FILE. Функция fopen возвращает указатель на FILE; таково объявление типа для fopen в <stdio.h>.

Вызов fopen в программе выглядит так:

```
char *name, *mode;

fp = fopen(name, mode);
```

Первый аргумент функции – это имя файла, символьная строка. Второй аргумент (тоже символьная строка) указывает, как именно предполагается использовать файл; разрешены такие режимы, как чтение ("r"), запись ("w") и добавление ("a").

Если файл, открываемый для записи или добавления, не существует, то он создается (если возможно). Если для записи открывается существующий файл, то его старое содержимое стирается. Попытка прочитать несуществующий файл приводит к ошибке – так же, как и попытка

прочитать или записать файл, для которого у пользователя нет прав доступа. В случае ошибки функция fopen возвращает недействительное значение указателя NULL (обычно определяется как (char *)0 в <stdio.h>).

Файл открыт, теперь надо как-то прочитать или записать его. Существует несколько способов сделать это, простейшим является использование getc и putc. Функция getc извлекает из файла следующий символ:

```
c = getc(fp)
```

в переменную c помещается следующий символ из файла, на который указывает fp; когда достигнут конец файла, возвращается EOF. Функция putc устроена аналогично:

```
putc(c, fp)
```

помещает символ c в файл fp и возвращает c. В случае ошибки getc и putc возвращают EOF.

Когда запускается программа, три файла уже открыты и на них существуют указатели. Это файлы стандартного ввода, стандартного вывода и стандартного вывода ошибок; соответствующие указатели называются stdin, stdout и stderr. Эти указатели файлов объявлены в <stdio.h>; они могут использоваться везде, где могут существовать объекты типа FILE *. Это константы, а не переменные, поэтому присваивать им значения нельзя.

Функция getchar() эквивалентна getc(stdin), а putchar(c) — putc(c,stdout).

В действительности все четыре «функции», упомянутые выше, определены в <stdio.h> как макросы, они выполняются быстрее, потому что не вызываются для каждого символа отдельно (как функция). Некоторые другие определения из <stdio.h> приведены в табл. 6.3.

Таблица 6.3. Некоторые описания <stdio.h>

Определение	Смысл
stdin	стандартный ввод
stdout	стандартный вывод
stderr	стандартный вывод ошибок
EOF	конец файла; обычно -1
NULL	недействительный указатель; обычно 0
FILE	используется для объявления указателей файлов
BUFSIZ	обычный размер буфера ввода-вывода (часто 512 или 1024)
getc(fp)	возвращает один символ из потока fp
getchar()	getc(stdin)
putc(c,fp)	помещает символ c в поток fp

Таблица 6.3 (продолжение)

Определение	Смысл
`putchar(c)`	`putc(c, stdout)`
`feof(fp)`	не ноль, если в потоке `fp` достигнут конец файла
`ferror(fp)`	не ноль, если в потоке `fp` обнаружена ошибка
`fileno(fp)`	файловый дескриптор потока `fp`, см. главу 7

Сделаем еще несколько предварительных замечаний и приступим к третьей версии `vis`. Если в командной строке есть аргументы, то они обрабатываются по порядку. Если аргументы не заданы, обрабатывается стандартный ввод.

```
/* vis: сделать видимыми непечатаемые символы (версия 3) */

#include <stdio.h>
#include <ctype.h>
int strip = 0;      /* 1 => удалить специальные символы */
char *progname;     /*program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    FILE *fp;
    progname = argv[0];

    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
        case 's':   /* -s: удалить странные символы */
            strip = 1;
            break;
        default:
            fprintf(stderr, "%s: unknown arg %s\n", argv[0], argv[1]);
            exit(1);
        }
        argc--;
        argv++;
    }
    if (argc == 1)
        vis(stdin);
    else
        for (i = 1; i < argc; i++)
            if ((fp=fopen(argv[i], "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n", progname, argv[i]);
                exit(1);
            } else {
                vis(fp);
                fclose(fp);
```

```
            }
        exit(0);
    }
```

В данной программе подразумевается выполнение соглашения о том, что первыми обрабатываются необязательные аргументы.

После того как обработаны все необязательные параметры, `argc` и `argv` корректируются так, что оставшаяся часть программы не зависит от наличия таких аргументов. Несмотря на то что `vis` распознает только одиночный параметр, программа написана как цикл для того, чтобы показать, как можно организовать обработку аргументов. В главе 1 упоминалось, что UNIX-программы обрабатывают необязательные аргументы. Еще одной причиной для этого, помимо склонности к анархии, служит то, что разбор любой комбинации параметров выполняется исключительно просто. В некоторых системах есть функция `getopt(3)`, с ее помощью пытались как-то упорядочить ситуацию; стоит изучить ее, прежде чем браться за написание своей собственной.

Функция `vis` выводит один файл:

```
vis(fp) /* сделать символы видимыми в FILE *fp */
    FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else if (!strip)
            printf("\\%03o", c);
}
```

Функция `fprintf` практически идентична `printf`, отличие лишь в том, что в аргументе – файловом указателе – указывается файл для записи.

Функция `fclose` обрывает связь между указателем файла и внешним именем, установленную `fopen`, и указатель освобождается для нового файла. Поскольку количество файлов, которые система может открывать одновременно (около 20), ограничено, лучше освобождать файлы, которые больше не понадобятся. Обычно вывод, осуществляемый какой-то из стандартных библиотечных функций типа `printf`, `putc` и т. д., буферизуется таким образом, что запись выполняется большими порциями, что увеличивает производительность. (Исключением является вывод на терминал, обычно осуществляемый по мере поступления или, по крайней мере, по вводу символа новой строки (построчно).) Вызов `fclose` для файла вывода закрывает любой буферизованный вывод. Когда программа вызывает `exit` или возвращается из `main`, происходит автоматическое обращение к `fclose` для каждого открытого файла.

Так же как `stdin` и `stdout`, программе присваивается `stderr`. Выходные данные, записанные на `stderr`, появятся на терминале пользователя,

даже если стандартный вывод будет перенаправлен. Программа vis выводит диагностические сообщения на stderr, а не на stdout, поэтому если по какой-то причине невозможно получить доступ к одному из файлов, сообщение об этом появится на терминале, а не исчезнет в конвейере или файле вывода. (Стандартный вывод ошибок был изобретен в некоторой степени именно потому, что ошибки начали исчезать в конвейерах.)

Условие выхода из программы выбрано, можно сказать, случайным образом. Это происходит, если vis не может открыть файл со входными данными; такой выбор выглядит разумным для программы, чаще всего использующейся интерактивно и получающей один файл ввода. Однако если хотите, можете выбрать и другую конструкцию.

Упражнение 6.5. Напишите программу printable, которая выводила бы имя каждого файла-аргумента, содержащего только печатаемые символы; если же в файле встречается какой-то непечатаемый символ, его имя не выводится. Такая программа полезна в следующих случаях:

```
$ pr `printable *` | lpr
```

Добавьте параметр -v, чтобы изменить смысл проверки на обратный, как в grep. Как программе следует поступать, если имена файлов не указаны? Какой код должна возвращать printable? □

6.4. Поэкранный вывод: команда p

До сих пор для исследования файлов применялась команда cat. Но если файл достаточно длинный, а соединение с системой высокоскоростное, то cat выводит данные слишком быстро – так, что даже хорошая реакция (быстрое нажатие *ctl*-s и *ctl*-q) не помогает прочитать его.

Нужна программа, которая печатала бы файл небольшими заданными порциями. Подобной стандартной программы не существует, вероятно, потому, что в годы создания системы UNIX коммуникационные каналы были медленными, а терминалами служили печатающие устройства. Поэтому напишем теперь программу (назовем ее p), которая будет выводить файл порциями, помещающимися на экране терминала, при этом, прежде чем перейти к демонстрации следующего экрана, программа ждет подтверждения пользователя на выполнение этого действия. («p» – это хорошее короткое название для программы, которая используется постоянно.) Как и другие программы, p считывает входные данные из файлов, указанных в аргументах, или с устройства стандартного ввода:

```
$ p vis.c
...
$ grep '#define' *.[ch] | p
...
$
```

Эту программу лучше всего писать на Си, потому что на Си проще – стандартные средства не подходят при смешивании ввода из файла или канала с терминальным вводом.

Основная (безо всяких излишеств) конструкция обеспечивает вывод небольшими фрагментами. Подходящим размером порции представляются 22 строки: это чуть меньше, чем 24 строки (помещающиеся на экран большинства видеотерминалов), и это треть стандартной страницы (66 строк). Предложим простой способ пригласить пользователя к действиям – не печатать последний символ новой строки каждой 22-строчной порции. Тогда курсор будет останавливаться в правом конце последней строки, а не на левой границе следующей. Когда пользователь нажмет клавмшу *RETURN*, появится недостающий символ новой строки, и следующая строка будет выведена на правильном месте. Если пользователь нажмет *ctl*-`d` или `q` в конце экрана, произойдет выход из `p`.

Длинные строки не будут обрабатываться специальным образом. Если задано несколько файлов, будем просто переходить с одного на другой, без каких бы то ни было комментариев. То есть поведение

```
$ p имена-файлов...
```

будет аналогично

```
$ cat имена-файлов... | p
```

Имена файлов можно добавить при помощи цикла `for`:

```
$ for i in имена-файлов...
> do
>       echo $i:
>       cat $i
> done | p
```

Несомненно, есть масса функций, которые можно было бы добавить в программу. Лучше начать с «голой» версии, а потом развивать ее так, как велит разум. Если пойти таким путем, то добавляться будут свойства, которые действительно нужны пользователям, а не те, что нам кажутся нужными.

Структура `p` в основном аналогична команде `vis`: функция `main` циклически проходит по файлам, обращаясь к функции `print`, которая выполняется для каждого файла.

```
/* p: печатать входные данные порциями (версия 1) */

#include <stdio.h>
#define PAGESIZE    22
char    *progname;  /* имя программы для сообщения об ошибке */

main(argc, argv)
    int argc;
    char *argv[];
{
```

```
    int i;
    FILE *fp, *efopen();

    progname = argv[0];
    if (argc == 1)
        print(stdin, PAGESIZE);
    else
        for (i = 1; i < argc; i++) {
            fp = efopen(argv[i], "r");
            print(fp, PAGESIZE);
            fclose(fp);
        }
    exit(0);
}
```

Функция efopen инкапсулирует очень распространенное действие: пытается открыть файл, а если это невозможно, выводит сообщение об ошибке и завершается. Для того чтобы в сообщении об ошибке идентифицировалась программа, нарушившая нормальную работу (или та, чья работа была нарушена), efopen ссылается на внешнюю строковую переменную progname, содержащую имя программы, которая устанавливается в main.

```
FILE *efopen(file, mode)    /* открыть файл с помощью fopen file, завершиться
если это невозможно */
    char *file, *mode;
{
    FILE *fp, *fopen();
    extern char *progname;

    if ((fp = fopen(file, mode)) != NULL)
        return fp;
    fprintf(stderr, "%s: can't open file %s mode %s\n",
        progname, file, mode);
    exit(1);
}
```

Прежде чем остановиться на таком варианте efopen, авторы опробовали несколько других. В одном из них функция возвращала после вывода сообщения нулевой указатель, свидетельствующий о неудаче. Так у кода, вызывающего функцию, появлялась возможность выбора: продолжать или выйти. В другом варианте у efopen был третий аргумент, определяющий, нужно ли возвращаться в случае неудачной попытки открыть файл. Однако практически во всех рассматриваемых примерах нет никакого смысла продолжать работу, если файл не удалось открыть, поэтому лучше использовать версию efopen, представленную выше.

Основная деятельность программы p осуществляется функцией print:

```
print(fp, pagesize) /* печатать fp порциями размером с pagesize */
    FILE *fp;
```

```
    int pagesize;
{
    static int lines = 0;   /* количество строк к этому моменту */
    char buf[BUFSIZ];

    while (fgets(buf, sizeof buf, fp) != NULL)
        if (++lines < pagesize)
            fputs(buf, stdout);
        else {
            buf[strlen(buf)-1] = '\0';
            fputs(buf, stdout);
            fflush(stdout);
            ttyin();
            lines = 0;
        }
}
```

Размер буфера ввода задается константой `BUFSIZ`, определенной в файле `<stdio.h>`. Функция `fgets(buf,size,fp)` переносит следующую строку ввода (вплоть до символа новой строки, включая его) из `fp` в `buf`, добавляя завершающий символ `\0`; максимальное количество копируемых символов равно `size-1`. В конце файла возвращается `NULL`. (Функция `fgets` могла бы быть построена удобнее: она возвращает не количество символов, а `buf`; к тому же не выдается предупреждения в случае слишком длинной строки ввода. Правда символы не теряются, но приходится просматривать `buf`, чтобы понять, что же происходит на самом деле.)

Функция `strlen` возвращает длину строки; она нужна для того, чтобы убрать замыкающий символ новой строки из последней строки ввода. Функция `fputs(buf,fp)` записывает строку `buf` в файл `fp`. Вызов `fflush` в конце страницы закрывает буферизованный вывод.

Решение задачи чтения ответа пользователя, поступающего после вывода каждой страницы, отводится процедуре `ttyin`. Она не может читать стандартный ввод, так как `p` должна работать корректно, даже если входные данные поступают из файла или канала. Для обработки данной ситуации программа открывает файл `/dev/tty`, соответствующий терминалу пользователя, куда бы ни перенаправлялся стандартный ввод. Процедура `ttyin` написана так, чтобы возвращать первый символ ответа, но здесь это ее свойство не использовано.

```
ttyin()     /* обработка ответа из /dev/tty (версия 1) */
{
char buf[BUFSIZ];
    FILE *efopen();
    static FILE *tty = NULL;

    if (tty == NULL)
        tty = efopen("/dev/tty", "r");
    if (fgets(buf, BUFSIZ, tty) == NULL || buf[0] == 'q')
```

```
        exit(0);
    else    /* ordinary line */
        return buf[0];
}
```

Указатель на файл `tty` объявляется как `static`, поэтому он сохраняет значение от одного вызова `ttyin` к следующему; файл `/dev/tty` открывается только при первом вызове.

Очевидно, что есть множество свойств, которыми можно было бы снабдить `p`, причем без особого труда, но первая версия этой программы, написанная авторами, делала только то, что здесь описано: она выводила 22 строки и ждала ответа пользователя. Прошло много времени, прежде чем добавилось что-то еще, и до сегодняшнего дня очень мало кто обращается к ее дополнительным возможностям.

Одно из легко реализуемых дополнений – это введение переменной `pagesize`, задающей количество строк на странице, она может устанавливаться в командной строке:

```
$ p -n ...
```

печать производится порциями по *n* строк. Для этого надо просто добавить несколько строчек обычного кода в начало функции `main`:

```
/* p: печатать входные данные порциями (версия 2) */
    ...
    int i, pagesize = PAGESIZE;

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    ...
```

Функция `atoi` преобразует символьную строку в целое число (см. `atoi(3)`).

Еще одним дополнением `p` может быть возможность временного выхода в конце каждой страницы для выполнения какой-то другой команды. По аналогии с `ed` и многими другими программами, если пользователь вводит строку, которая начинается с восклицательного знака, то оставшаяся часть строки воспринимается как команда и передается на исполнение оболочке. Такое изменение также тривиально реализуемо, так как существует функция `system(3)`, которая делает именно то, что нужно (но следует проявить осторожность, далее будет пояснено, почему). Вот и новая версия `ttyin`:

```
ttyin() /* обработка ответа из /dev/tty (версия 2) */
{
    char buf[BUFSIZ];
```

```
    FILE *efopen();
    static FILE *tty = NULL;

    if (tty == NULL)
        tty = efopen("/dev/tty", "r");
    for (;;) {
        if (fgets(buf,BUFSIZ,tty) == NULL || buf[0] == 'q')
            exit(0);
        else if (buf[0] == '!') {
            system(buf+1);  /* здесь ОШИБКА */
            printf("!\n");
        }
        else    /* ordinary line */
            return buf[0];
    }
}
```

К сожалению, в этой версии есть с трудом различимая, но фатальная ошибка. Команда, запущенная `system`, наследует стандартный ввод от `p`, так что если `p` считывала данные из канала или файла, то команда может вмешаться и помешать ее вводу:

```
$ cat /etc/passwd | p -1
root:3D.fHR5KoB.3s:0:1:S.User:/:!ed          Вызывает ed из p
?                                            ed читает /etc/passwd...
!                                            ...запутывается и выходит
```

Для того чтобы решить эту проблему, необходимо знать, как в UNIX происходит управление процессами, об этом будет рассказано в разделе 7.4. А пока помните, что использование стандартной функции `system` из библиотеки может приводить к ошибкам, но знайте, что `ttyin` работает корректно, если она скомпилирована с той версией `system`, которая представлена в главе 7.

Итак, написаны две программы, `vis` и `p`, которые можно рассматривать как несколько приукрашенные варианты `cat`. Может быть, следовало бы сделать их частями `cat`, доступными при указании соответствующего необязательного аргумента `-v` или `-p`? Подобный вопрос – писать ли новую программу или добавлять новые возможности к уже существующий, возникает каждый раз, когда нужно что-то сделать. Окончательного ответа мы не знаем, но есть несколько правил, которые могут помочь определиться.

Основной принцип заключается в том, что программа должна выполнять только одну базовую функцию, если она начнет делать слишком много всего, она станет громоздкой, медленной, ее сложнее будет сопровождать и использовать. На самом деле дополнительные возможности часто остаются неиспользованными, потому что люди не в состоянии запомнить параметры.

Поэтому лучше не комбинировать `cat` и `vis`, ведь `cat` просто копирует свои входные данные, не изменяя их, в то время как `vis` их преобразу-

ет. При их слиянии получилась бы программа, выполняющая два разных действия. Ситуация с `cat` и `p` практически настолько же очевидна: `cat` предназначена для быстрого, производительного копирования, а `p` – для просмотра. К тому же `p` преобразует вывод: каждый 22-й символ новой строки удаляется. Похоже, что лучшим решением будет сохранение трех отдельных программ.

Упражнение 6.6. Будет ли `p` разумно работать в случае, если `pagesize` является неположительным числом? □

Упражнение 6.7. Что еще можно сделать с `p`? Оцените и реализуйте (если сочтете целесообразным) возможность выводить заново части, прочитанные ранее. (Авторам нравится это свойство.) Добавьте возможность выводить меньше, чем экран входных данных после каждой паузы. Добавьте возможность просмотра вперед и назад в поиске строки, для которой указан номер или содержимое. □

Упражнение 6.8. Используйте средства обработки файлов, присущие встроенной в оболочку функции `exec` (см. `sh(1)`), для того чтобы исправить обращение `ttyin` к `system`. □

Упражнение 6.9. Если не указать `p`, откуда брать ввод, программа будет спокойно ждать ввода с терминала. Стоит ли выявить эту возможную ошибку? Если да, то как? Подсказка: `isatty(3)`. □

6.5. Пример: pick

Версия `pick`, предложенная в главе 5, была явным расширением возможностей оболочки. Программа, которая сейчас будет написана на Си, будет несколько отличаться от версии главы 5. Если аргументы для `pick` указаны, они обрабатываются так же, как и раньше, а вот если указан только один аргумент «-», то команда обрабатывает стандартный ввод.

Почему бы не читать стандартный ввод в случае, если не указан ни один аргумент? Рассмотрим вторую версию команды `zap` из раздела 5.6:

```
kill $SIG `pick \`ps -ag | egrep "$*"\` | awk '{print $1}'`
```

Что произойдет, если для шаблона `egrep` не найдено ни одного соответствия? В этом случае нет аргументов для `pick`, и она начинает считывать свой стандартный ввод; а команда `zap` таинственным образом выходит из строя. Требование наличия явного аргумента является простым способом устранения неоднозначности подобных ситуаций, обозначение же «-» подсказано `cat` и другими программами.

```
/* pick: возможность выбора каждого аргумента */

#include <stdio.h>
char    *progname;  /* имя программы для сообщения об ошибке */

main(argc, argv)
    int argc;
```

```
    char *argv[];
{
    int i;
    char buf[BUFSIZ];

    progname = argv[0];
    if (argc == 2 && strcmp(argv[1],"-") == 0)  /* pick - */
        while (fgets(buf, sizeof buf, stdin) != NULL) {
            buf[strlen(buf)-1] = '\0';  /* удалить символ новой строки */
            pick(buf);
        }
    else
        for (i = 1; i < argc; i++)
            pick(argv[i]);
    exit(0);
}

pick(s) /* возможность выбора s */
    char *s;
{
    fprintf(stderr, "%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}
```

В программе `pick` сосредоточены возможности интерактивного выбора аргументов. Это не просто полезная услуга, таким образом уменьшается потребность в «интерактивных» параметрах для других команд.

Упражнение 6.10. Теперь, когда в нашем распоряжении есть `pick`, нужна ли `rm -i`? □

6.6. Об ошибках и отладке

Каждому, кто хоть раз пытался написать программу, знакомо понятие «баг» (ошибка). Единственное, что можно сделать, чтобы написать программу без ошибок, – это выбрать простую, без излишеств конструкцию, аккуратно ее реализовать и сохранять ее простой по ходу любых изменений.

Некоторые средства UNIX могут помочь в обнаружении ошибок, но надо сказать, что ни одно из них нельзя назвать первоклассным. Однако чтобы показать это на примере, нужна ошибка, а все программы в этой книге совершенны. Так что создадим типичную ошибку. Рассмотрим функцию `pick`, представленную выше. Вот еще одна ее версия, на этот раз с ошибкой (чур, не подсматривать в оригинал!).

```
pick(s) /* возможность выбора s */
    char *s;
{
    fprintf(stderr, "%s? ", s);
    if (ttyin() == 'y')
```

```
        printf("%s\n", s);
}
```

Что произойдет, если скомпилировать и запустить программу?

```
$ cc pick.c -o pick
$ pick *.c                          Пробуем
Memory fault - core dumped          Катастрофа!
$
```

«Memory fault» означает, что программа пытается сослаться на область памяти, доступ в которую ей не разрешен. Обычно так бывает, когда указатель ссылается «в никуда». Еще одно диагностическое сообщение похожего содержания – это «bus error», оно часто возникает при сканировании незаконченной строки.

«Core dumped» означает, что ядро сохранило состояние исполняющейся программы в файле `core` в текущем каталоге. Можно заставить программу записать дамп оперативной памяти, нажав *ctl-* (если это интерактивная программа) или введя команду `kill -3` (если программа выполняется в фоновом режиме).

«Вскрытие» производят две программы – `adb` и `sdb`. Как и большинство отладчиков, они окутаны тайной, сложны и незаменимы. Программа `adb` входит в состав седьмой версии; а `sdb` доступна в более поздних версиях системы. Одна из них наверняка есть в каждой системе.[1]

В этой книге есть возможность представить только минимальное описание каждого из отладчиков: вывести трассировку стека, то есть функцию, которая выполнялась в момент аварийного завершения программы, функцию, которая вызвала ее, и т. д. Функция, чье имя стоит первым в трассировке, – это та функция, на которой программа прервалась.

Чтобы получить трассировку стека с помощью `adb`, используем команду `$C`:

```
$ adb pick core             Вызов adb
$C                          Запрос трассировки стека
~_strout(0175722,011,0,011200)
        adjust:     0
        fillch:     060542
__doprnt(0177345,0176176,011200)
~fprintf(011200,0177345)
        iop:        011200
        fmt:        0177345
        args:       0
~pick(0177345)
        s:          0177345
```

[1] Пользователи свободно распространяемых ОС, таких как FreeBSD или Linux, могут обратиться к отладчику `gdb`. – *Примеч. науч. ред.*

```
~main(035,0177234)
        argc:       035
        argv:       0177234
        i:          01
        buf:        0
ctl-d                       Выход
$
```

Это означает, что `main` вызвала `pick`, которая вызвала `fprintf`, она, в свою очередь, вызвала `_doprnt`, которая вызвала `_strout`. Так как `_doprnt` не упоминается нигде в `pick.c`, проблема должна быть где-то в `fprintf` или выше. (Строки после каждой подпрограммы представляют значения локальных переменных. Команда `$c` подавляет вывод такой информации, а в некоторых версиях `adb` это происходит и при использовании самой команды `$C`.)

Прежде чем окончательно выявить ошибку, попробуем сделать то же самое с помощью `sdb`:

```
$ sdb pick core
Warning: `a.out' not compiled with -g
lseek: address 0xa64      Функция, на которой программа оборвалась
*t                        Запрос трассировки стека
lseek()
fprintf(6154,2147479154)
pick(2147479154)
main(30,2147478988,2147479112)
*q                         Выход
$
```

Информация имеет разный формат, но суть одна и та же: `fprintf`. (Трассировка выглядит по-другому, потому что программа была запущена на другой машине – VAX-11/750, на которой иначе реализована стандартная библиотека ввода-вывода.) И действительно, если посмотреть на вызов `fprintf` в испорченной версии `pick`, то видно, что он неверен:

```
fprintf("%s? ", s);
```

Отсутствует `stderr`, поэтому форматирующая строка `"%s? "` используется как указатель `FILE`, что, естественно, вызывает хаос.

Эта ошибка была выбрана потому, что она является весьма распространенной, скорее по недосмотру, а не из-за неправильного проектирования.

Можно выявить ошибки вызова функции с неправильными аргументами, используя верификатор Си `lint(1)`. Программа `lint` обследует программы, написанные на Си, на предмет возможных ошибок, проблем с переносимостью и сомнительных конструкций. Если запустить `lint` для всего файла `pick.c`, ошибка будет идентифицирована:

```
$ lint pick.c
...
```

```
fprintf, arg. 1 used inconsistently "llib-lc"(69) :: "pick.c"(28)
...
$
```

В переводе это означает, что описание первого аргумента `fprintf` в стандартной библиотеке отличается от способа его использования в 28-й строке программы. Это полезное указание на нечто неправильное.

Правда надо сказать, что `lint` применяют с переменным успехом. Верификатор точно указывает, что же в программе неверно, но при этом выдает множество не относящихся к делу сообщений (в вышеприведенном примере они опущены), так что необходимо обладать некоторой квалификацией, чтобы понять, на что стоит обратить внимание, а что можно проигнорировать. Усилия вознаграждаются, так как `lint` может находить такие ошибки, которые человек просто не в состоянии заметить. Стоит запустить `lint` после долгого редактирования и убедиться, что вам понятны все выдаваемые предупреждения.

6.7. Пример: zap

Программа `zap`, выборочно уничтожающая процессы, также была представлена в главе 5 в виде командного файла. Основным недостатком той версии является скорость: программа создает так много процессов, что выполнение становится очень медленным, что нежелательно, особенно для программы, убивающей ошибочные процессы. Переписанная на Си, `zap` будет работать быстрее. Не будем делать всю работу: по-прежнему используем `ps` для поиска информации о процессе. Это *гораздо* легче, чем получать информацию из ядра, к тому же такое решение переносимо. Программа `zap` открывает канал, в котором со стороны ввода находится `ps`, и считывает данные оттуда, а не из файла. Функция `popen(3)` аналогична `fopen`, только первым аргументом является не имя файла, а команда. Существует и функция `pclose`, которая не используется в данном примере.

```
/* zap: интерактивный убийца процессов */

#include <stdio.h>
#include <signal.h>
char    *progname;  /* имя программы для сообщения об ошибке */
char    *ps = "ps -ag"; /* зависит от системы */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fin, *popen();
    char buf[BUFSIZ];
    int pid;

    progname = argv[0];
```

```
    if ((fin = popen(ps, "r")) == NULL) {
        fprintf(stderr, "%s: can't run %s\n", progname, ps);
        exit(1);
    }
    fgets(buf, sizeof buf, fin);    /* получить заголовочную строку */
    fprintf(stderr, "%s", buf);
    while (fgets(buf, sizeof buf, fin) != NULL)
        if (argc == 1 || strindex(buf, argv[1]) >= 0) {
            buf[strlen(buf)-1] = '\0'; /* suppress \n */
            fprintf(stderr, "%s? ", buf);
            if (ttyin() == 'y') {
                sscanf(buf, "%d", &pid);
                kill(pid, SIGKILL);
            }
        }
    exit(0);
}
```

Программа была написана с использованием `ps -ag` (параметр зависит от конкретной системы),[1] но до тех пор пока вы не станете привилегированным пользователем, вы сможете уничтожать только свои собственные процессы.

При первом вызове `fgets` получает от `ps` заголовочную строку (попробуйте понять, что произойдет, если попытаться уничтожить «процесс», соответствующий этой заголовочной строке – это полезное упражнение).

Функция `sscanf` принадлежит к семейству `scanf(3)`, занимающемуся преобразованием формата ввода. Она конвертирует не файл, а строку. Системный вызов `kill` посылает указанный сигнал процессу; при этом сигнал `SIGKILL`, определенный в `<signal.h>`, не может быть перехвачен или проигнорирован. Может быть, вы помните (об этом говорилось в главе 5), что численное значение отправленного сигнала равно 9, но лучше использовать символьные константы, а не разбрасывать по программе магические числа.

Если аргументы не заданы, команда `zap` предоставляет для выбора каждую строку вывода `ps`. Если же аргумент указан, то `zap` предлагает только соответствующие ему строки вывода `ps`. Функция `strindex(s1,s2)` проверяет, соответствует ли аргумент какой-либо части строки вывода `ps`, используя `strncmp` (см. табл. 6.2). Функция `strindex`

[1] Параметр `-a` требует отображения процессов всех пользователей, а не только тех, которые принадлежат пользователю, запустившему `ps`. В System V `ps -a` не выводит процессы, являющиеся лидерами группы процессов, для включения их в выводимый список необходим параметр `-g`. В ОС ветви BSD действие параметра `-g` не определено. Например, во FreeBSD v. 4.4 `ps` не имеет параметра `-g`, но его указание при запуске `ps` сообщений об ошибке не вызывает. – *Примеч. науч. ред.*

возвращает позицию в s1, в которой встретилась s2, или -1, если соответствие не обнаружено.

```
strindex(s, t)  /* возвращает индекс t в s и -1, если не найдено */
    char *s, *t;
{
    int i, n;

    n = strlen(t);
    for (i = 0; s[i] != '\0'; i++)
        if (strncmp(s+i, t, n) == 0)
            return i;
    return -1;
}
```

Список наиболее часто применяемых функций из стандартной библиотеки ввода-вывода приведен в табл. 6.4.

Таблица 6.4. Полезные стандартные функции ввода-вывода

Функция	Действие
fp=fopen(s,mode)	открыть файл s; режимы r, w, a для чтения, записи и добавления (в случае ошибки возвращает NULL)
c=getc(fp)	извлечь символ; getchar() — это getc(stdin)
putc(c,fp)	вставить символ; putchar() — это putc(c, stdout)
ungetc(c,fp)	вернуть символ обратно в файл ввода fp, за один раз не более одного символа
scanf(fmt,a1,...)	читать символы из stdin в a1, ... в соответствии с fmt. Каждый a_i должен быть указателем. Возвращает EOF или количество преобразованных полей
fscanf(fp,...)	читать из файла fp
sscanf(s,...)	читать из строки s
printf(fmt,a1,...)	форматировать a1, ... в соответствии с fmt, выводить в stdout
fprintf(fp,...)	выводить ... в файл fp
sprintf(s,...)	выводить ... в строку s
fgets(s,n,fp)	прочитать не более чем n символов из fp в s. В конце файла возвращает NULL
fputs(s,fp)	выводить строку s в файл fp
fflush(fp)	сбросить буферизованный вывод для файла fp
fclose(fp)	закрыть файл fp
fp=popen(s,mode)	открыть канал для команды s. См. fopen
pclose(fp)	закрыть канал fp
system(s)	запустить команду s и ждать завершения

Упражнение 6.11. Измените `zap` так, чтобы можно было задавать любое количество аргументов. Предложенная версия `zap` предлагает для выбора строку, соответствующую самой себе. Правильно ли это? Если нет, измените программу. Подсказка: `getpid(2)`. □

Упражнение 6.12. Напишите функцию `fgrep(1)`, основываясь на `strindex`. Сравните время исполнения при сложном поиске, например для 10 слов в документе. Почему `fgrep` работает быстрее? □

6.8. Интерактивная программа сравнения файлов: idiff

Часто бывает так, что в наличии имеются две версии файла, в чем-то различных, при этом каждая версия содержит часть искомого файла. Обычно так случается, когда два пользователя вносят изменения независимо друг от друга. Программа `diff` может рассказать, чем отличаются файлы, но не поможет, если надо выбрать что-то из первого файла, а что-то – из второго.

В этом разделе будет написана программа `idiff` («интерактивная `diff`»), которая выводит каждую порцию выходных данных `diff` и предлагает пользователю возможность выбрать (по своему усмотрению) первый или второй варианты или же редактировать их. Команда `idiff` выводит выбранные части в правильном порядке в файл `idiff.out`. То есть если рассматриваются два таких файла:

file1:
```
This is
a test
of
your
skill
and comprehension.
```

file2:
```
This is
not a test
of
our
ability.
```

`diff` выводит:

```
$ diff file1 file2
2c2
< a test
---
> not a test
4,6c4,5
< your
< skill
< and comprehension.
---
> our
> ability.
$
```

Диалог с idiff может выглядеть следующим образом:

```
$ idiff file1 file2
2c2                    Первое отличие
< a test
---
> not a test
? >                    Пользователь выбирает вторую (>) версию
4,6c4,5                Второе отличие
< your
< skill
< and comprehension.
---
> our
> ability.
? <                    Пользователь выбирает первую (<) версию
idiff output in file idiff.out
$ cat idiff.out        Вывод помещен в этот файл
This is
not a test
of
your
skill
and comprehension.
$
```

Если вместо < или > выбран ответ e, то idiff вызывает редактор ed, при этом две группы строк уже считаны в редактор. Если бы вторым ответом было e, то буфер редактора выглядел бы следующим образом:

```
your
skill
and comprehension.
---
our
ability.
```

То, что ed записывает обратно в файл, и попадает в конечный вывод.

Наконец, любая команда может быть запущена во время выполнения idiff посредством выхода в оболочку – !*cmd*.

Самая сложная с технической точки зрения задача выполняется функцией diff, которая уже была сделана до нас. Так что собственно idiff остается только разбирать выходные данные diff и открывать, закрывать, читать и записывать соответствующие файлы в нужное время. В функции main программы idiff открываются файлы и запускается процесс diff:

```
/* idiff:  интерактивная diff */

#include <stdio.h>
#include <ctype.h>
char    *progname;
```

```
#define HUGE    10000   /* большое количество строк */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fin, *fout, *f1, *f2, *efopen();
    char buf[BUFSIZ], *mktemp();
    char *diffout = "idiff.XXXXXX";

    progname = argv[0];
    if (argc != 3) {
        fprintf(stderr, "Usage: idiff file1 file2\n");
        exit(1);
    }
    f1 = efopen(argv[1], "r");
    f2 = efopen(argv[2], "r");
    fout = efopen("idiff.out", "w");
    mktemp(diffout);
    sprintf(buf,"diff %s %s >%s",argv[1],argv[2],diffout);
    system(buf);
    fin = efopen(diffout, "r");
    idiff(f1, f2, fin, fout);
    unlink(diffout);
    printf("%s output in file idiff.out\n", progname);
    exit(0);
}
```

Функция `mktemp(3)` создает файл, имя которого гарантированно отличается от имени любого существующего файла. Аргумент функции `mktemp` перезаписывается: шесть `X` заменяются идентификатором процесса `idiff` и буквой.[1] Системный вызов `unlink(2)` удаляет названный файл из файловой системы.

Перебором изменений, отмеченных `diff`, занимается функция `idiff`. Основная идея достаточно проста: вывести часть выходных данных `diff`, перескочить через ненужные данные в одном файле, затем скопировать выбранную версию из другого. Существует множество утомительных подробностей, делающих программу не такой маленькой, как хотелось бы, но если разбить ее на части, несложно понять, как она работает.

[1] `diffout` представляет собой указатель на строку символов, которая является константой. Это обстоятельство вызовет аварийное завершение программы, как только функция `mktemp` предпримет попытку записи в область памяти, распределенную под константы. Однако это не ошибка авторов, просто в те времена, когда эта программа была написана, еще можно было так обращаться с указателями на строковые константы. Чтобы избежать фатальной ошибки из-за попытки записи в неправильный сегмент памяти, необходимо при использовании `gcc` указать в командной строке компилятора опцию `-fwritable-strings`. – *Примеч. науч. ред.*

```
idiff(f1, f2, fin, fout)    /* обработка различий */
    FILE *f1, *f2, *fin, *fout;
{
    char *tempfile = "idiff.XXXXXX";
    char buf[BUFSIZ], buf2[BUFSIZ], *mktemp();
    FILE *ft, *efopen();
    int cmd, n, from1, to1, from2, to2, nf1, nf2;

    mktemp(tempfile);
    nf1 = nf2 = 0;
    while (fgets(buf, sizeof buf, fin) != NULL) {
        parse(buf, &from1, &to1, &cmd, &from2, &to2);
        n = to1-from1 + to2-from2 + 1; /* число строк из diff */
        if (cmd == 'c')
            n += 2;
        else if (cmd == 'a')
            from1++;
        else if (cmd == 'd')
            from2++;
        printf("%s", buf);
        while (n-- > 0) {
            fgets(buf, sizeof buf, fin);
            printf("%s", buf);
        }
        do {
            printf("? ");
            fflush(stdout);
            fgets(buf, sizeof buf, stdin);
            switch (buf[0]) {
            case '>':
                nskip(f1, to1-nf1);
                ncopy(f2, to2-nf2, fout);
                break;
            case '<':
                nskip(f2, to2-nf2);
                ncopy(f1, to1-nf1, fout);
                break;
            case 'e':
                ncopy(f1, from1-1-nf1, fout);
                nskip(f2, from2-1-nf2);
                ft = efopen(tempfile, "w");
                ncopy(f1, to1+1-from1, ft);
                fprintf(ft, "---\n");
                ncopy(f2, to2+1-from2, ft);
                fclose(ft);
                sprintf(buf2, "ed %s", tempfile);
                system(buf2);
                ft = efopen(tempfile, "r");
                ncopy(ft, HUGE, fout);
                fclose(ft);
                break;
```

```
            case '!':
                system(buf+1);
                printf("!\n");
                break;
            default:
                printf("< or > or e or !\n");
                break;
            }
        } while (buf[0]!='<' && buf[0]!='>' && buf[0]!='e');
        nf1 = to1;
        nf2 = to2;
    }
    ncopy(f1, HUGE, fout);  /* может выйти из строя на слишком длинном файле */
    unlink(tempfile);
}
```

Функция `parse` **осуществляет банальную, но достаточно сложную операцию по разбору строк, выводимых** `diff`**, выделяя четыре номера строк и команду (**`a`**,** `c` **или** `d`**). Эта функция сложна, потому что** `diff` **может выводить по одному номеру строки с каждой стороны от буквы команды, а может и по два.**

```
parse(s, pfrom1, pto1, pcmd, pfrom2, pto2)
    char *s;
    int *pcmd, *pfrom1, *pto1, *pfrom2, *pto2;
{
#define a2i(p) while (isdigit(*s)) p = 10*(p) + *s++ - '0'

    *pfrom1 = *pto1 = *pfrom2 = *pto2 = 0;
    a2i(*pfrom1);
    if (*s == ',') {
        s++;
        a2i(*pto1);
    } else
        *pto1 = *pfrom1;
    *pcmd = *s++;
    a2i(*pfrom2);
    if (*s == ',') {
        s++;
        a2i(*pto2);
    } else
        *pto2 = *pfrom2;
}
```

Макрос `a2i` **занимается специализированным преобразованием из ASCII в целое число в четырех местах, где это необходимо.**

Функции `nskip` **и** `ncopy` **пропускают указанное количество строк или копируют их из файла:**

```
nskip(fin, n)   /* пропустить n строк в файле fin */
    FILE *fin;
{
```

```
    char buf[BUFSIZ];

    while (n-- > 0)
        fgets(buf, sizeof buf, fin);
}

ncopy(fin, n, fout) /* копировать n строк из fin в fout */
    FILE *fin, *fout;
{
    char buf[BUFSIZ];

    while (n-- > 0) {
        if (fgets(buf, sizeof buf, fin) == NULL)
            return;
        fputs(buf, fout);
    }
}
```

В том виде, как она написана сейчас, `idiff`, завершаясь при прерывании, оставляет после себя несколько файлов в `/tmp`, что не очень-то красиво. В следующей главе будет рассказано о том, как перехватывать прерывания, чтобы избавиться от временных файлов, подобных имеющимся в данной ситуации.

Сделаем важное замечание относительно двух программ, `zap` и `idiff` — и за одну, и за другую самую сложную работу выполняет кто-то другой. Эти программы просто предоставляют удобный интерфейс для другой программы, которая вычисляет нужную информацию. Использовать чью-то работу вместо того, чтобы делать ее самому, — это недорогой способ повышения производительности.

Упражнение 6.13. Добавьте команду `q` в `idiff` следующим образом: ответ `q<` будет автоматически выбирать все оставшиеся ответы `<`, а `q>` будет автоматически выбирать все оставшиеся ответы `>`. □

Упражнение 6.14. Измените `idiff` так, чтобы любые аргументы `diff` передавались в `diff`; возможными кандидатами являются `-b` и `-h`. Измените `idiff` так, чтобы можно было указывать другой редактор:

```
$ idiff -e другой-редактор file1 file2
```

Как эти два изменения будут взаимодействовать друг с другом? □

Упражнение 6.15. Измените `idiff` так, чтобы вместо временного файла для вывода `diff` использовались `popen` и `pclose`. Как это повлияет на скорость и сложность программы? □

Упражнение 6.16. Команда `diff` обладает следующим свойством: если один из ее аргументов является каталогом, она просматривает этот каталог в поиске файла с тем же именем, которое указано вторым аргументом. Но если попробовать этот способ с `idiff`, она непонятным образом выходит из строя. Объясните, что происходит, и исправьте ошибку. □

6.9. Доступ к окружению

Из программы, написанной на Си, легко получить доступ к переменным окружения оболочки, это обстоятельство позволяет адаптировать программы к окружению, не требуя многого от их пользователей. Предположим, что используется терминал, экран которого больше, чем обычные 24 строки. Что делать, если надо выполнить программу `p` и при этом хочется воспользоваться возможностями терминала? Указывать размер экрана при каждом запуске `p` слишком утомительно:

```
$ p -36 ...
```

Всегда можно поместить командный файл в свой каталог `/bin`:

```
$ cat /usr/you/bin/p
exec /usr/bin/p -36 $*
$
```

Есть и третье решение – изменить `p` так, чтобы она использовала переменную окружения, которая определяет свойства терминала. Предположим, что вы задаете переменную `PAGESIZE` в файле `.profile`:

```
PAGESIZE=36
export PAGESIZE
```

Стандартная функция `getenv` (“*var*”) ищет в окружении переменную оболочки *var* и возвращает ее значение в форме символьной строки или `NULL`, если переменная не определена. Имея в наличии `getenv`, легко изменить программу `p`. Надо лишь добавить несколько объявлений и вызов `getenv` в начале функции `main`.

```
/* p: печатать входные данные порциями (версия 3) */
...
char  *p, *getenv();

progname = argv[0];
    if ((p=getenv("PAGESIZE")) != NULL)
        pagesize = atoi(p);
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
...
```

Необязательные аргументы обрабатываются после переменной окружения, поэтому явно заданный размер страницы будет преобладать над неявным.

Упражнение 6.17. Измените `idiff` так, чтобы окружение просматривалось в поисках имени редактора, который будет использоваться. Измените `2`, `3` и т. д., чтобы использовать `PAGESIZE`. □

История и библиография

Стандартная библиотека ввода-вывода была разработана Деннисом Ритчи (Dennis Ritchie) на основе переносимой библиотеки ввода-вывода Майка Леска (Mike Lesk). Цель обоих пакетов состоит в том, чтобы предоставить достаточно стандартных средств, обеспечивающих перенос программ из UNIX в другие системы без изменений.

Предложенная архитектура `p` базируется на программе Генри Спенсера (Henry Spencer).

Отладчик `adb` был написан Стивом Бурном (Steve Bourne), `sdb` – Говардом Катцеффом (Howard Katseff), а `lint` – Стивом Джонсоном (Steve Johnson).

Программа `idiff` написана «по мотивам» программы, созданной Джо Маранзано (Joe Maranzano). Сама программа `diff` принадлежит Дагу Мак-Илрою (Doug McIlroy), а основывается она на алгоритме, независимо разработанном Гарольдом Стоуном (Harold Stone) и Уэйном Хантом (Wayne Hunt) с Томом Шимански (Tom Szymanski). (Hunt J. W., Szymanski O. G. «A fast algorithm for computing longest common subseguences», CACM, May 1977.) Алгоритм `diff` описан Мак-Илроем М. Д. и Хантом Д. У. в статье «An algorithm for differential file comparison», появившейся в Bell Labs Computing Science Technical Report 41 в 1976 году. Процитируем Мак-Илроя: «Я опробовал как минимум три различных алгоритма, прежде чем на чем-то остановиться. Программа `diff` является наиболее типичным случаем того, когда в программе не довольствуешься просто выполнением задачи, а переделываешь ее до тех пор, пока она не станет совершенной».

7

Системные вызовы UNIX

Данная глава посвящена самому нижнему уровню взаимодействия с операционной системой UNIX – системным вызовам, которые служат точками входа в ядро. Именно эти возможности предоставляет собственно операционная система; все остальное строится на этой основе.

Рассмотрим несколько основных тем. Прежде всего, систему ввода-вывода, лежащую в основе таких библиотечных функций, как `fopen` и `putc`. Также мы продолжим обсуждение файловой системы, в частности каталогов и индексных дескрипторов (inode). Затем обсудим процессы – способы запуска программ из других программ. После этого поговорим о сигналах и прерываниях: что происходит при нажатии клавиши *DELETE* и как правильно обработать это в программе.

Как и в главе 6, многие примеры представляют собой полезные программы, не вошедшие в седьмую верию. Даже если вы не будете использовать именно эти программы, знакомство с ними может подсказать вам идеи для своих собственных проектов.

Системные вызовы детально рассмотрены в man2, а в этой главе будут рассмотрены наиболее важные моменты (без претензии на полноту изложения).

7.1. Низкоуровневый ввод-вывод

Самый нижний уровень ввода-вывода напрямую взаимодействует с операционной системой. Программа читает и пишет файлы фрагментами произвольного размера. Ядро буферизует данные в блоках, размер которых определяется периферийным устройством, и устанавливает очередность работы устройств, оптимизируя производительность для всех пользователей.

Дескрипторы файлов

Весь ввод и вывод выполняется как чтение и запись файлов, поскольку все периферийные устройства, включая и терминал, представлены в файловой системе файлами. Это значит, что единый интерфейс управляет всем взаимодействием между программой и периферийными устройствами.

В самом общем случае перед тем как читать или писать файл, необходимо сообщить системе о своем намерении; этот процесс называется *открытием* файла. Если вы собираетесь записывать в файл, возможно, его надо предварительно *создать*. Система проверяет права пользователя на выполнение операции (существует ли файл? есть ли разрешение на доступ к нему?) и, если все в порядке, возвращает неотрицательное целое число, которое и называется *дескриптором файла*. Всякий раз, когда выполняется файловый ввод-вывод, дескриптор используется вместо имени для идентификации файла. Вся информация об открытом файле поддерживается системой, а программа обращается к нему только по дескриптору. Указатель типа `FILE`, как рассказывалось в главе 6, указывает на структуру, которая кроме прочего содержит дескриптор файла, а макрос `fileno(fp)`, определенный в `<stdio.h>`, возвращает этот дескриптор.

Существуют специальные соглашения, позволяющие сделать терминальный ввод-вывод более удобным. Программа, будучи запущена оболочкой, получает от нее три открытых файла с дескрипторами `0`, `1` и `2`, которые называются стандартным вводом, стандартным выводом и стандартным выводом ошибок. Все они по умолчанию связаны с терминалом, так что если программа ограничивается чтением файла с дескриптором `0` и записью в файлы с дескрипторами `1` и `2`, ей не нужно открывать файлы для ввода-вывода. Если же программа открывает иные файлы, то они получат дескрипторы `3`, `4` и т. д.

Если используется перенаправление ввода-вывода, то оболочка изменяет установленное по умолчанию соответствие для дескрипторов `0` и `1` с терминала на указанные файлы. Обычно файловый дескриптор `2` остается связанным с терминалом, чтобы на него могли выводиться сообщения об ошибках. Такие команды, как `2>`*имя-файла* и `2>&1`, вызовут переопределение умолчаний, причем переназначение файлов выполняется оболочкой, а не программой. (При желании программа может сама совершить переназначение, но это встречается редко.)

Файловый ввод-вывод – read и write

Весь ввод и вывод осуществляется двумя системными вызовами – `read` и `write`, которые доступны из языка Си через функции с теми же именами. В обеих функциях первый аргумент – это дескриптор файла. Второй аргумент – массив байт, служащий источником или приемником данных. Третий аргумент задает количество передаваемых байт.

```
int fd, n, nread, nwritten;
char buf[SIZE];

nread = read(fd, buf, n);
nwritten = write(fd, buf, n);
```

Оба вызова возвращают количество фактически переданных байт. При чтении возвращаемое число может быть меньше ожидаемого, если до конца файла осталось менее `n` байт. (Если файл является терминалом, вызов `read` обычно читает до символа новой строки, а это, как правило, меньше указанного значения.) При достижении конца файла возвращается значение `0`, а в случае ошибки – значение `-1`. При записи возвращается число фактически записанных байт; отличие этого значения от ожидаемого говорит о наличии ошибки.

Хотя количество байт для чтения и записи не ограничивается, на практике чаще всего задается значение `1`, то есть посимвольная передача данных (небуферизованный ввод-вывод), и значение, равное размеру дискового блока, чаще всего 512 или 1024. (Это значение содержится в константе `BUFSIZ` в файле `stdio.h`.)

Проиллюстрируем вышесказанное при помощи программы, копирующей свой ввод в вывод. Так как ввод и вывод могут быть перенаправлены в любой файл или устройство, эта программа копирует что угодно куда угодно: это схематичная реализация команды `cat`.

```
/* cat: минимальная версия */
#define SIZE 512 /* произвольное значение */

main()
{
    char buf[SIZE];
    int n;

    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    exit(0);
}
```

Если размер файла не кратен `SIZE`, то очередной вызов `read` вернет меньшее значение для записи, а следующий после этого вызов `read` вернет значение `0`.

Наиболее эффективно чтение и запись данных выполняются порциями, равными размеру дискового блока, хотя и посимвольный ввод-вывод годится для небольших объемов данных, так как ядро само выполняет буферизацию и основные затраты приходятся на системные вызовы. Редактор `ed`, к примеру, использует побайтное чтение из стандартного ввода. Мы измерили производительность этой версии `cat` на файле длиной 54 000 байт для шести значений `SIZE`:

	Время (пользовательское+системное, сек)	
SIZE	PDP-11/70	VAX-11/750
1	271,0	188,8
10	29,9	19,3
100	3,8	2,6
512	1,3	1,0
1024	1,2	0,6
5120	1,0	0,6

Размер блока составляет 512 байт в системе PDP-11 и 1024 в системе VAX.

Совершенно нормальна ситуация, когда несколько процессов одновременно обращаются к одному файлу; например, один процесс записывает данные, а другой в это время их читает. Такое поведение может сбивать с толку, но иногда бывает и полезным. Даже если вызов `read` вернул `0`, следующий вызов может обнаружить новые данные, если они в это время были записаны. Этот эффект положен в основу программы `readslow`, которая продолжает чтение ввода независимо от того, достигнут ли конец файла. Программа `readslow` удобна для наблюдения за процессом выполнения программ:

```
$ slowprog >temp &
5213                    Идентификатор процесса
$ readslow <temp | grep something
```

Другими словами, медленная программа выполняет вывод в файл, а программа `readslow`, возможно, взаимодействуя с какой-либо другой программой, следит за накоплением данных.

Программа `readslow` идентична программе `cat` за исключением того, что она продолжает выполнять цикл после достижения конца файла. Здесь обращение к низкоуровневому вводу-выводу неизбежно, так как функции стандартной библиотеки продолжают возвращать `EOF` после первого достижения конца файла.

```
/* readslow: непрерывное чтение в ожидании данных */
#define SIZE     512     /* произвольное значение */

main()
{
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

Функция sleep приостанавливает выполнение программы на заданное число секунд; это описано в sleep(3). Не нужно, чтобы readslow тормошила файл, постоянно обращаясь за новыми данными; это потребует слишком много процессорного времени. Таким образом, эта версия readslow копирует свой ввод, достигает конца файла и прерывается ненадолго, затем пытается повторить ввод. Если за время бездействия поступили новые данные, они будут считаны при следующем проходе.

Упражнение 7.1. Добавьте в программу readslow параметр *-n* так, чтобы функция sleep останавливала выполнение на *n* секунд. В некоторых системах команда tail имеет параметр -f (forever, то есть бесконечно), который включает в команде tail режим, подобный readslow. Прокомментируйте это решение. □

Упражнение 7.2. Что произойдет с программой readslow, если размер читаемого файла уменьшится? Как это исправить? Подсказка: прочитайте об fstat в разделе 7.3. □

Создание файла – open, creat, close, unlink

Помимо использования стандартных файлов ввода, вывода и вывода ошибок, вам потребуется явно открывать файлы для чтения и записи. Для этой цели существуют два системных вызова: open и creat.[1]

Функция open аналогична fopen из предыдущей главы за исключением того, что она возвращает не указатель файла, а файловый дескриптор, имеющий тип int.

```
char *name;
int fd, rwmode;

fd = open(name, rwmode);
```

Как и в fopen, параметр name – это строка, содержащая имя файла. А вот параметр типа доступа отличается: rwmode имеет значение 0 для чтения, 1 для записи и 2 для чтения и записи. Функция open возвращает -1 в случае ошибки, а при успешном завершении – дескриптор файла.

Попытка открытия существующего файла приводит к ошибке. Для создания новых файлов и для перезаписи уже существующих предназначен системный вызов creat.

```
int perms;

fd = creat(name, perms);
```

Функция creat возвращает дескриптор файла, если ей удалось его создать, и -1 в противном случае. Если файл не существует, то creat созда-

[1] Кена Томпсона (Ken Thompson) однажды спросили, что бы он изменил в UNIX, если бы довелось перепроектировать систему заново. Он ответил: «Я бы написал creat с буквой e».

ет его с *правами доступа*, указанными в параметре perms. Если же файл уже существует, то creat обрежет его до нулевой длины; вызов creat для уже существующего файла не является ошибкой. (Права доступа не изменятся.) Независимо от значения perms созданный файл открыт для записи.

Как рассказывалось в главе 2, информация о правах доступа к файлу хранится в девяти битах, определяющих разрешение на чтение, запись и исполнение, поэтому их принято представлять трехзначным восьмеричным числом. Например, 0755 определяет разрешение на чтение, запись и выполнение для владельца и на чтение и выполнение для группы и для остальных. Не забывайте начинать восьмеричные числа с 0, как это принято в языке Си.

Для иллюстрации приведем упрощенную версию команды cp. Главное упрощение заключается в том, что наша версия копирует только один файл и не позволяет в качестве второго аргумента указывать каталог. Второй изъян состоит в том, что данная версия не сохраняет права доступа исходного файла (дальше будет показано, как с этим справиться).

```
/* cp: минимальная версия */
#include <stdio.h>
#define PERMS 0644 /* RW для владельца, R для группы и остальных */
char *progname;

main(argc, argv)  /* cp: копировать f1 в f2 */
int argc;
    char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZ];

    progname = argv[0];
    if (argc != 3)
        error("Usage: %s from to", progname);
    if ((f1 = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("write error", (char *) 0);
    exit(0);
}
```

Функция error будет описана в следующем разделе.

Существует ограничение (значение NOFILE в sys/param.h, обычно около 20) количества файлов, одновременно открываемых программой. Следовательно, если программа собирается работать с многими файлами, ей следует позаботиться о повторном использовании файловых де-

скрипторов. Системный вызов close разрывает связь между именем файла и его дескриптором, освобождая таким образом дескриптор для использования с другим файлом. Завершение программы посредством вызова exit или выхода из main вызывает закрытие всех файлов.

Системный вызов unlink удаляет имя; файл удаляется, если это последняя ссылка.

Обработка ошибок – errno

Системные вызовы, описанные в этом разделе, а фактически все системные вызовы, могут завершаться с ошибками. Обычно они сообщают об этом, возвращая значение -1. Иногда бывает полезно знать, какая именно ошибка произошла; с этой целью все системные вызовы, когда это уместно, оставляют номер ошибки во внешней переменной errno. (Значения различных номеров ошибок приведены во введении раздела 2 руководства по UNIX для программиста.) Пользуясь errno, программа может, к примеру, определить, вызвана ошибка открытия файла его отсутствием или же отсутствием прав доступа к нему у пользователя. Для преобразования номеров в текстовые сообщения существует массив строк sys_errlist, индексом в котором служат значения errno.[1]

Наша версия функции error использует эти структуры данных:

```
error(s1, s2) /* вывести сообщение об ошибке и закончить работу */
char *s1, *s2;
{
    extern int errno, sys_nerr;
    extern char *sys_errlist[], *progname;

    if (progname)
        fprintf(stderr, "%s: ", progname);
    fprintf(stderr, s1, s2);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, " (%s)", sys_errlist[errno]);
    fprintf(stderr, "\n");
    exit(1);
}
```

Значение errno инициализируется нулем и не должно превышать sys_nerr. При отсутствии ошибок errno не обнуляется, следовательно, пользователь должен делать это самостоятельно, если программа продолжает работу после возникновения ошибки.

---

[1] В современных реализациях переменная sys_errlist объявлена в stdio.h как константный указатель, поэтому следует исключить предложенное ниже объявление sys_errlist, так как его присутствие в тексте программы будет вызывать ошибку компиляции из-за несоответствия типов в объявлениях sys_errlist, полученных из stdio.h и из текста программы. – *Примеч. науч. ред.*

Вот как выглядят сообщения об ошибках в данной версии cp:

```
$ cp foo bar
cp: can't open foo (No such file or directory)
$ date >foo; chmod 0 foo          Создан файл с запретом на чтение
$ cp foo bar
cp: can't open foo (Permission denied)
$
```

Произвольный доступ – lseek

Обычно ввод-вывод выполняется последовательно: каждая операция чтения или записи начинается там, где закончилась предыдущая. Но при необходимости файл может быть прочитан и записан в произвольном порядке. Системный вызов lseek позволяет перемещаться по файлу, не выполняя при этом чтение или запись:

```
int fd, origin;
long offset, pos, lseek();

pos = lseek(fd, offset, origin);
```

В этом фрагменте текущая позиция в файле с дескриптором fd перемещается на offset байт относительно позиции, определенной параметром origin. Следующая операция чтения или записи начнется с этой позиции. Параметр origin может принимать значения 0, 1 и 2, при этом смещение отсчитывается от начала файла, от текущей позиции и от конца файла соответственно. Функция возвращает новое значение абсолютной позиции или -1 в случае ошибки. Например, чтобы дописать данные в файл, переместитесь в его конец:

```
lseek(fd, 0L, 2);
```

Чтобы вернуться в начало («перемотать»),

```
lseek(fd, 0L, 0);
```

Чтобы определить текущую позицию,

```
pos = lseek(fd, 0L, 1);
```

Обратите внимание на аргумент 0L: смещение – это длинное целое. (Буква «l» в имени lseek означает long, т. е. «длинное», что отличает эту функцию от seek из шестой версии, использующей короткие целые.)

Функция lseek позволяет с некоторой степенью приближения трактовать файлы как большие массивы, правда ценой более медленного доступа. Вот, например, функция, читающая произвольное количество байт из произвольного места файла:

```
get(fd, pos, buf, n) /* прочитать n байт из позиции pos */
    int fd, n;
```

```
    long pos;
    char *buf;
{
    if (lseek(fd, pos, 0) == -1) /* перейти на позицию pos */
        return -1;
    else
        return read(fd, buf, n);
```

Упражнение 7.3. Измените функцию `readslow` так, чтобы она обрабатывала имя файла, если оно передано параметром. Добавьте параметр `-e`:

```
$ readslow -e
```

заставляющий `readslow` перемещаться в конец входного файла перед началом чтения. Как поведет себя `lseek` при работе с программным каналом (pipe)? □

Упражнение 7.4. Перепишите функцию `efopen` из главы 6 с использованием вызова `error`. □

7.2. Файловая система: каталоги

Теперь посмотрим, как перемещаться по иерархии каталогов. Для этого не потребуются новые системные вызовы, просто используем старые в новом контексте. Поясним на примере функции `spname`, которая попытается справиться с неправильным именем файла. Функция

```
n = spname (имя, новое-имя);
```

ищет файл с именем, «достаточно похожим» на заданное параметром *имя*. Если таковое найдено, оно копируется в параметр *новое-имя*. Возвращаемое значение равно `-1`, если ничего похожего не найдено, `0` – если найдено точное соответствие, и `1` – если потребовалось исправление.

Функция `spname` представляет собой удобное дополнение к программе `p`. Если пользователь хочет напечатать файл, но ошибся в написании его имени, программа `p` может попросить его уточнить, что имелось в виду:

```
$ p /urs/srx/ccmd/p/spnam.c      Абсолютно неправильное имя
"/usr/src/cmd/p/spname.c"? y     Предложенное исправление принято
/* spname: возвратить правильно написанное имя файла */
...
```

Напишем функцию так, чтобы она пыталась исправить в каждом элементе имени файла следующие ошибки: одна пропущенная или лишняя буква, одна буква неправильная, пара букв перепутана местами – все эти варианты есть в примере. Такая программа будет просто находкой для невнимательной машинистки.

Перед тем как начать программировать, совершим краткий экскурс в структуру файловой системы. Каталог – это файл, содержащий пере-

чень имен файлов с указанием их расположения. Под «расположением» здесь понимается индекс файла в другой таблице, называемой *таблицей индексных дескрипторов*. Запись в этой таблице, называемая индексным дескриптором (inode) файла, – это то место, где хранится вся информация о нем, за исключением имени. Таким образом, запись в каталоге состоит из двух элементов – номера индексного дескриптора и имени файла. Точная спецификация находится в файле `sys/dir.h`:

```
$ cat /usr/include/sys/dir.h
#define DIRSIZ 14 /* максимальная длина имени файла */

struct direct /* структура записи в каталоге */
{
    ino_t d_ino; /* номер индексного дескриптора */
    char d_name[DIRSIZ]; /* имя файла */
};
$
```

«Тип» данных `ino_t`, определенный с помощью `typedef`, описывает запись в таблице индексных дескрипторов. В системах PDP-11 и VAX это тип `unsigned short`, но в других системах он, скоре всего, будет другим, не стоит полагаться на это в программах, поэтому и использован `typedef`. Полный набор «системных» типов находится в файле `sys/types.h`, который должен быть включен до файла `sys/dir.h`.

Алгоритм работы `spname` достаточно прост. Предположим, надо обработать имя файла `/d1/d2/f`. Основная идея заключается в том, чтобы отбросить первый элемент (/) и искать в корневом каталоге имя, максимально совпадающее со следующим элементом (`d1`), затем в найденном каталоге искать что-либо подобное `d2` и так далее, пока не будут найдены соответствия для всех элементов. Если в очередном каталоге не удалось найти подходящее имя, поиск прекращается.

Вся работа разделена между тремя функциями: собственно `spname` отделяет элементы друг от друга и составляет из них «наиболее вероятное» имя файла. Функция `mindist` вызывается из `spname` и выполняет в заданном каталоге поиск файла с именем, максимально похожим на заданное, обращаясь при этом к функции `spdist`, вычисляющей «расстояние» между двумя именами.

```
/* spname: возвращает правильно написанное имя файла */
/*
 * spname(oldname, newname)  char *oldname, *newname;
 *  returns -1 if no reasonable match to oldname,
 *           0 if exact match,
 *           1 if corrected.
 *  stores corrected name in newname.
 */

#include <sys/types.h>
#include <sys/dir.h>
```

```
spname(oldname, newname)
    char *oldname, *newname;
{
    char *p, guess[DIRSIZ+1], best[DIRSIZ+1];
    char *new = newname, *old = oldname;

    for (;;) {
        while (*old == '/') /* пропустить слэши */
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0')   /* правильно или исправлено */
            return strcmp(oldname,newname) != 0;
        p = guess;  /* скопировать следующий компонент в guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZ)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
            return -1;  /* неудачно */
        for (p = best; *new = *p++; ) /* добавить в конец */
            new++;                    /* нового имени */
    }
}

mindist(dir, guess, best)   /* просмотр dir в поиске guess */
    char *dir, *guess, *best;
{
    /* устанавливает best, возвращает расстояние 0..3 */
    int d, nd, fd;
    struct {
        ino_t ino;
        char  name[DIRSIZ+1];   /* на 1 больше, чем в dir.h */
    } nbuf;

    nbuf.name[DIRSIZ] = '\0';   /* +1 для заключительного '\0' */
    if (dir[0] == '\0')     /* текущий каталог */
        dir = ".";
    d = 3;  /* минимальное расстояние */
    if ((fd=open(dir, 0)) == -1)
        return d;
    while (read(fd,(char *) &nbuf,sizeof(struct direct)) > 0)
        if (nbuf.ino) {
            nd = spdist(nbuf.name, guess);
            if (nd <= d && nd != 3) {
                strcpy(best, nbuf.name);
                d = nd;
                if (d == 0)     /* точное совпадение */
                    break;
            }
        }
    close(fd);
    return d;
}
```

Если имя каталога, переданное в функцию `mindist`, пусто, то поиск выполняется в текущем каталоге (`.`), за один проход обрабатывается один каталог. Обратите внимание, что в качестве буфера для чтения выступает структура, а не символьный массив. Размер определяется посредством `sizeof`, а затем адрес преобразуется в указатель на тип `char`.

Когда запись в каталоге не используется (если файл был удален), то соответствующая запись, содержащая нулевое значение индексного дескриптора, пропускается. Расстояние проверятся выражением

```
if (nd <= d ...)
```

а не

```
if (nd < d ...)
```

так что любой символ считается более подходящим, чем точка «.», которая всегда является первым элементом каталога.

```
/* spdist: возвращает расстояние между двумя именами */
/*
 *  очень грубый показатель правильности написания:
 *  0, если две строки идентичны
 *  1, если два символа переставлены местами
 *  2, если один символ добавлен, удален или не совпадает
 *  3 - иначе
 */

#define EQ(s,t) (strcmp(s,t) == 0)

spdist(s, t)
    char *s, *t;
{
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;       /* точное совпадение */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1]
              && *t == s[1] && EQ(s+2, t+2))
                return 1;   /* перестановка */
            if (EQ(s+1, t+1))
                return 2;   /* 1 несовпадающий символ */
        }
        if (EQ(s+1, t))
            return 2;       /* лишний символ */
    }
    if (*t && EQ(s, t+1))
        return 2;           /* недостающий символ */
    return 3;
}
```

Теперь, когда у нас есть функция spname, очень просто добавить проверку написания к команде p:

```
/* p: печатает ввод порциями (версия 4) */

#include <stdio.h>
#define PAGESIZE    22
char  *progname; /* имя программы для сообщения об ошибке */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp, *efopen();
    int i, pagesize = PAGESIZE;
    char *p, *getenv(), buf[BUFSIZ];

    progname = argv[0];
    if ((p=getenv("PAGESIZE")) != NULL)
        pagesize = atoi(p);
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc == 1)
        print(stdin, pagesize);
    else
        for (i = 1; i < argc; i++)
            switch (spname(argv[i], buf)) {
            case -1:    /* совпадение невозможно */
                fp = efopen(argv[i], "r");
                break;
            case 1:     /* исправлено */
                fprintf(stderr, "\"%s\"? ", buf);
                if (ttyin() == 'n')
                    break;
                argv[i] = buf;
                /* неуадчно… */
            case 0: /* точное совпадение */
                fp = efopen(argv[i], "r");
                print(fp, pagesize);
                fclose(fp);
            }
    exit(0);
}
```

Автоматическое исправление имен файлов надо использовать с определенной осторожностью. Оно хорошо работает в интерактивных программах, таких как p, но не подходит для программ, которые могут работать без общения с пользователем.

Упражнение 7.5. Насколько можно было бы усовершенствовать эвристический алгоритм наилучшего совпадения в `spname`? Например, нет смысла обрабатывать имя файла как каталог, а в данной версии это возможно. □

Упражнение 7.6. Имя `tx` всегда будет считаться подходящим, если каталог заканчивается на `t`*c*, независимо от значения *c*. Можно ли улучшить определение расстояния? Напишите программу и посмотрите, как она понравится пользователям. □

Упражнение 7.7. Функция `mindist` читает каталог поэлементно. Можно ли существенно ускорить программу `p`, выполняя чтение большими блоками? □

Упражнение 7.8. Измените функцию `spname` так, чтобы она возвращала префикс предполагаемого имени, если не удалось найти достаточно близкое соответствие. Что делать, если несколько имен соответствуют этому префиксу? □

Упражнение 7.9. В каких еще программах можно использовать `spname`? Напишите автономную программу, исправляющую полученные аргументы перед тем, как передать их в другую программу, например такую, как

```
$ fix prog имена-файлов...
```

Можно ли написать версию `cd` с использованием `spname`? Как ее проинсталлировать? □

7.3. Файловая система: индексные дескрипторы

В этом разделе будут рассмотрены системные вызовы, имеющие дело с файловой системой и, в частности, с информацией о файлах, такой как размер, даты, права доступа и т. д. Эти системные вызовы дают доступ ко всей той информации, которую мы обсуждали в главе 2.

Давайте теперь обратимся собственно к индексным дескрипторам. Часть их описывается структурой `stat`, определяемой в `sys/stat.h`:

```
struct stat /* структура, возвращаемая stat */
{
dev_t    st_dev;      /* устройство индексного дескриптора */
ino_t    st_ino;      /* номер индексного дескриптора */
short    st_mode      /* биты доступа */
short    st_nlink;    /* количество ссылок на файл */
short    st_uid;      /* идентификатор владельца */
short    st_gid;      /* идентификатор группы владельца */
dev_t    st_rdev;     /* для специальных файлов */
off_t    st_size;     /* размер файла в символах */
time_t   st_atime;    /* время последнего чтения из файла */
```

```
time_t   st_mtime;    /* время последней записи или создания файла */
time_t   st_ctime;    /* время последнего изменения файла или индексного
                         дескриптора */
};
```

Большинство полей описано в комментариях. Такие типы, как `dev_t` и `ino_t`, определены в `sys/types.h`, это обсуждалось в предыдущих разделах. Элемент `st_mode` содержит набор флагов, описывающих файл; для удобства флаги определены также в файле `sys/stat.h`:

```
#define S_IFMT  0170000    /* тип файла */
#define    S_IFDIR 0040000 /* каталог */
#define    S_IFCHR 0020000 /* специальный символьный */
#define    S_IFBLK 0060000 /* специальный блочный */
#define    S_IFREG 0100000 /* обычный */
#define S_ISUID  0004000   /* при запуске установить эффективный
                              идентификатор пользователя как у владельца */
#define S_ISGID  0002000   /* при запуске установить эффективный
                              идентификатор группы как у владельца */
#define S_ISVTX  0001000   /* сохранить перекачанный текст даже после
                              использования */
#define S_IREAD  0000400   /* права на чтение, владелец */
#define S_IWRITE 0000200   /* права на запись, владелец */
#define S_IEXEC  0000100   /* права на выполнение/поиск, владелец */
```

Доступ к индексному дескриптору файла выполняется при помощи двух системных вызовов: `stat` и `fstat`. Вызов `stat` принимает имя файла и возвращает информацию индексного дескриптора для этого файла (или `-1` в случае ошибки), `fstat` делает то же самое из файлового дескриптора для открытого файла (не из указателя `FILE`). Таким образом,

```
char *name;
int fd;
struct stat stbuf;

stat(name, &stbuf);
fstat(fd, &stbuf);
```

Структура `stbuf` заполняется информацией из индексного дескриптора для файла с именем `name` или дескриптором `fd`.

Теперь, когда мы располагаем всеми этими фактами, попробуем написать какой-нибудь полезный код. Начнем с того, что напишем на языке Си программу `checkmail`, которая будет наблюдать за почтовым ящиком пользователя. Если файл увеличивается в размере, программа пишет `You have mail` и подает звуковой сигнал. (Если файл становится меньше, это, видимо, потому, что вы только что прочитали и удалили какое-то письмо, сообщения не требуются). Для первого шага этого вполне достаточно; когда эта программа заработает, можно будет ее усовершенствовать.

```
/* checkmail: следит за почтовым ящиком пользователя */
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
char *progname;
char *maildir = "/usr/spool/mail";  /* зависит от системы */

main(argc, argv)
    int argc;
    char *argv[];
{
    struct stat buf;
    char *name, *getlogin();
    int lastsize = 0;

    progname = argv[0];
    if ((name = getlogin()) == NULL)
        error("can't get login name", (char *) 0);
    if (chdir(maildir) == -1)
        error("can't cd to %s", maildir);
    for (;;) {
        if (stat(name, &buf) == -1) /* нет почтового ящика */
            buf.st_size = 0;
        if (buf.st_size > lastsize)
            fprintf(stderr, "\nYou have mail\007\n");
        lastsize = buf.st_size;
        sleep(60);
    }
}
```

Функция `getlogin(3)` возвращает имя, с которым зарегистрирован пользователь, или `NULL`, если имя не получено. Системный вызов `chdir` переводит `checkmail` в почтовый каталог, так что последующим вызовам `stat` не приходится просматривать каждый каталог, начиная с корневого и заканчивая почтовым. В вашей системе, может быть, придется изменить значение `maildir`. Программа `checkmail` написана так, чтобы попытки предпринимались даже в том случае, когда почтовый ящик не существует, поскольку большинство почтовых программ удаляют почтовый ящик, если он пуст.

Эта программа была представлена в главе 5 для иллюстрации циклов оболочки. Та версия создавала несколько процессов при каждом просмотре почтового ящика, что могло вызвать большую, чем хотелось бы, нагрузку на систему. Версия на языке Си – это единственный процесс, запускающий `stat` для файла каждую минуту. Во что обойдется программа проверки почтового ящика, которая все время работает в фоновом режиме? По оценкам авторов, будет затрачено менее одной секунды в час, что довольно мало, и это чрезвычайно важно.

Иллюстрация к обработке ошибок: sv

Теперь напишем программу под названием `sv` (похожую на `cp`), которая копирует набор файлов в каталог, но при этом перезаписывает файл назначения, только если он не существует или является более

старым, чем исходный файл. Название sv происходит от английского слова save (сохранить) – идея в том, что sv не перезаписывает более новые версии файлов. Команда sv учитывает больше информации из индексного дескриптора, чем checkmail.

Будем запускать sv следующим образом:

```
$ sv file1 file2 ... dir
```

Эта команда должна копировать file1 в dir/file1, file2 в dir/file2 и т. д., за исключением тех случаев, когда файл назначения новее соответствующего исходного файла – в этом случае копия не делается и выдается предупреждение. Для того чтобы избежать многократного копирования ссылок, программа sv не разрешает использовать / в именах исходных файлов.

```
/* sv: сохраняет новые файлы */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    struct stat stbuf;
    char *dir = argv[argc-1];

    progname = argv[0];
    if (argc <= 2)
        error("Usage: %s files... dir", progname);
    if (stat(dir, &stbuf) == -1)
        error("can't access directory %s", dir);
    if ((stbuf.st_mode & S_IFMT) != S_IFDIR)
        error("%s is not a directory", dir);
    for (i = 1; i < argc-1; i++)
        sv(argv[i], dir);
    exit(0);
}
```

Значения времени в индексном дескрипторе задаются в секундах после 0:00 GMT, 1 января 1970, так что более старые файлы имеют меньшие значения в поле st_mtime.

```
sv(file, dir)   /* сохраняет file в dir */
char *file, *dir;
{
    struct stat sti, sto;
    int fin, fout, n;
```

```
    char target[BUFSIZ], buf[BUFSIZ], *index();

    sprintf(target, "%s/%s", dir, file);
    if (index(file, '/') != NULL)  /* strchr() в некоторых системах */
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1)   /* файл назначения не существует */
        sto.st_mtime = 0;   /* пусть он будет старее */
    if (sti.st_mtime < sto.st_mtime)    /* файл назначения более новый */
        fprintf(stderr, "%s: %s not copied\n",
            progname, file);
    else if ((fin = open(file, 0)) == -1)
        error("can't open file %s", file);
    else if ((fout = creat(target, sti.st_mode)) == -1)
        error("can't create %s", target);
    else
        while ((n = read(fin, buf, sizeof buf)) > 0)
            if (write(fout, buf, n) != n)
                error("error writing %s", target);
    close(fin);
    close(fout);
}
```

Вместо стандартных функций ввода-вывода использована `creat` для того, чтобы `sv` имела возможность сохранить код прав доступа исходного файла. (Обратите внимание на то, что `index` и `strchr` – это разные названия одной и той же процедуры; посмотрите в руководстве описание `string(3)`, чтобы узнать, какое имя использует ваша система.)

Несмотря на то что `sv` – это специализированная программа, она иллюстрирует несколько важных идей. Есть множество программ, хотя и не являющихся «системными», но тем не менее способных использовать информацию, хранимую операционной системой, к которой можно получить доступ посредством системных вызовов. Для таких программ ключевым моментом является то, что системные типы данных определены только в стандартных заголовочных файлах, таких как `stat.h` и `dir.h`, и программа включает в себя эти файлы, а не использует собственные определения типов. У такого кода гораздо больше шансов оказаться переносимым из одной системы в другую.

Не удивляйтесь, что как минимум две трети кода программы `sv` – это проверка ошибок. На начальных этапах написания программы есть соблазн сэкономить на обработке ошибок, так как это не входит в постановку основной задачи. А когда программа «заработает», уже не хватает энтузиазма для того, чтобы вернуться назад и вставить проверки, которые превратят частную программу в устойчиво работающий продукт.

Программа `sv` не защищена от всех возможных несчастий, например она не обрабатывает прерывания в неудобное время, но она более осто-

рожна, чем большинство программ. Остановимся буквально на одном моменте – рассмотрим последний оператор `write`. Редко случается, что `write` не выполняется успешно, поэтому многие программы игнорируют такую возможность. Но на дисках заканчивается свободное место; пользователи превышают квоты; происходят обрывы линий связи. Все эти обстоятельства могут послужить причиной ошибок записи, и будет гораздо лучше, если пользователь узнает об этом, вместо того чтобы программа молча делала вид, что все хорошо.

Мораль в том, что проверка ошибок скучна и утомительна, но чрезвычайно важна. Из-за ограниченного пространства и желания сконцентрировать внимание на наиболее интересных вопросах авторы опускали эту часть во многих программах, рассмотренных в данной книге. Но в настоящих программных продуктах вы не можете позволить себе игнорировать ошибки.

Упражнение 7.10. Измените `checkmail` таким образом, чтобы идентифицировать отправителя письма в сообщении `You have mail` (Вы получили письмо). Подсказка: `sscanf`, `lseek`. □

Упражнение 7.11. Измените `checkmail` так, чтобы каталог не изменялся на почтовый до входа в цикл. Будет ли это иметь значение для производительности? (Сложнее.) Можете ли вы написать версию `checkmail`, которой требовался бы только один процесс для оповещения всех пользователей? □

Упражнение 7.12. Напишите программу `watchfile`, которая проверяет файл и печатает его с начала каждый раз, когда он изменяется. Где можно использовать такую программу? □

Упражнение 7.13. Программа `sv` не обладает гибкостью в обработке ошибок. Измените ее так, чтобы она продолжалась даже в том случае, когда она не может обработать некоторые файлы. □

Упражнение 7.14. Сделайте `sv` рекурсивной: если один из исходных файлов – это каталог, пусть каталог и файлы из него обрабатываются одинаково. Сделайте `cp` рекурсивной. Подумайте, должны ли `cp` и `sv` быть одной и той же программой, так чтобы `cp -v` не создавала копию в том случае, когда файл назначения более новый. □

Упражнение 7.15. Напишите программу `random`:

```
$ random имя-файла
```

которая выводит одну строку, случайным образом выбранную из файла. Если `random` обрабатывает файл с именами, то она может быть использована в программе `scapegoat`, которая помогает найти виноватого:

```
$ cat scapegoat
echo "It's all `random people`'s fault!"
$ scapegoat
It's all Ken's fault!
$
```

Убедитесь, что random работает правильно вне зависимости от распределения длин строк. □

Упражнение 7.16. В индексном дескрипторе содержится и другая информация, как, например, адреса на диске, в которых расположены блоки файлов. Исследуйте файл sys/ino.h, а потом напишите программу icat, которая будет читать файлы, определяемые по индексному дескриптору и дисковому устройству. (Естественно, она будет работать только при условии, что рассматриваемый диск читаем.) При каких условиях полезна icat? □

7.4. Процессы

Этот раздел описывает выполнение одной программы из другой. Самый простой способ сделать это – обратиться к стандартной библиотечной функции system, описанной, но осужденной в главе 6. Команда system получает один аргумент – командную строку точно в том виде, как она набрана на терминале (за исключением символа новой строки в конце), и выполняет ее в подоболочке. Если командная строка должна быть составлена из нескольких частей, то могут пригодиться возможности форматирования в памяти, которыми обладает sprintf. В конце данного раздела будет представлена более надежная версия system для использования в интерактивных программах, но сначала надо исследовать части, из которых она состоит.

Низкоуровневое создание процессов – execlp и execvp

Базовая операция – это выполнение другой программы *без ожидания завершения* системным вызовом execlp. Например, чтобы напечатать дату в качестве последнего действия выполняющейся программы, используйте

```
execlp("date", "date", (char *) 0);
```

Первый аргумент execlp – это имя файла команды; execlp получает путь поиска (т. е. $PATH) из окружения и осуществляет такой же поиск, как оболочка. Второй и последующие аргументы – это имя команды и ее параметры; они становятся массивом argv для новой программы. Конец списка помечен нулевым значением (чтобы понять конструкцию execlp, обратитесь к exec(2)).

Вызов execlp перекрывает существующую программу новой, запускает ее и выходит. Первичная программа получает управление обратно только в случае ошибки, например, если файл не найден или не является выполняемым:

```
execlp("date", "date", (char *) 0);
```

```
fprintf(stderr, "Couldn't execute 'date'\n");
exit(1);
```

Разновидность execlp, именуемая execvp, применяется в тех случаях, когда количество аргументов заранее не известно. Вызов выглядит так:

```
execvp(filename, argp);
```

где argp — это массив указателей на аргументы (как argv); последний указатель в массиве должен быть NULL, чтобы execvp имел возможность определить, где кончается список. Как и для execlp, filename — это файл, в котором находится программа, а argp — это массив argv для новой программы; argp[0] — это имя программы.

Ни одна из этих программ не допускает наличия метасимволов <, >, *, кавычек и т. д. в списке аргументов. Если это необходимо, вызывайте посредством execlp оболочку /bin/sh, которая выполнит всю работу. Сформируйте командную строку, которая будет содержать всю команду, как если бы она была напечатана на терминале, затем скажите:

```
execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

Аргумент -c определяет, что следующий аргумент должен рассматриваться как целая командная строка, а не как отдельный аргумент.

Рассмотрим в качестве иллюстрации программу waitfile. Команда

```
$ waitfile имя-файла [ команда ]
```

периодически проверяет указанный файл. Если он не изменился с момента последней проверки, то *команда* выполняется. Если *команда* не определена, то файл копируется на стандартное устройство вывода. Для мониторинга работы troff мы используем waitfile:

```
$ waitfile troff.out echo troff done &
```

Реализация waitfile извлекает время изменения файла при помощи fstat.

```
/* waitfile: ждет, пока файл не перестанет изменяться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int fd;
    struct stat stbuf;
    time_t old_time = 0;
```

```
    progname = argv[0];
    if (argc < 2)
        error("Usage: %s filename [cmd]", progname);
    if ((fd = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    fstat(fd, &stbuf);
    while (stbuf.st_mtime != old_time) {
        old_time = stbuf.st_mtime;
        sleep(60);
        fstat(fd, &stbuf);
    }
    if (argc == 2) {    /* копировать файл */
        execlp("cat", "cat", argv[1], (char *) 0);
        error("can't execute cat %s", argv[1]);
    } else {            /* запустить процесс */
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    exit(0);
}
```

Тут проиллюстрированы оба вызова: `execlp` и `execvp`.

Выбран именно такой подход, потому что он полезен для понимания, но приемлемы и другие варианты. Например, `waitfile` может просто завершаться, после того как файл перестал изменяться.

Упражнение 7.17. Измените программу `watchfile` (упражнение 7.12) таким образом, чтобы она имела те же свойства, что и `waitfile`: если нет параметра *команда*, она копирует файл; в ином случае выполняет команду. Могут ли `watchfile` и `waitfile` совместно использовать исходный код? Подсказка: `argv[0]`. □

Управление процессами – fork и wait

Следующий шаг – это восстановление управления после того, как программа выполнена с помощью `execlp` или `execvp`. Поскольку эти программы просто накладывают новую программу поверх старой, то чтобы сохранить первичную программу, необходимо сначала разделить ее на две копии; одна из них может быть перезаписана, в то время как вторая ожидает окончания новой, наложенной программы. Разделение осуществляется системным вызовом `fork`:

```
proc_id = fork();
```

программа разделяется на две копии, каждая из которых продолжает выполняться. Единственное различие между ними заключается в значении, возвращаемом `fork`, – это *идентификатор процесса*. Для одного из этих процессов (*дочернего*) `proc_id` равен нулю. Для другого (*родительского*) `proc_id` нулю не равен. Таким образом, элементарный способ вызвать другую программу и вернуться из нее выглядит так:

```
if (fork() == 0)
    execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

И этого вполне достаточно, если не считать обработки ошибок. `fork` создает две копии программы. Для дочернего процесса значение, возвращаемое `fork`, равно нулю, поэтому вызывается `execlp`, который выполняет командную строку `commandline` и умирает. Для родительского процесса `fork` возвращает ненулевую величину, поэтому `execlp` пропускается. (В случае наличия ошибок `fork` возвращает `-1`.)

Чаще всего родительская программа ждет, пока закончится дочерняя, прежде чем продолжить свое собственное выполнение. Это реализуется при помощи системного вызова `wait`:

```
int status;

if (fork() == 0)
    execlp(...);          /* дочерняя */
wait(&status);            /* родительская */
```

Здесь еще не обрабатываются никакие аномальные обстоятельства, такие как сбой в работе `execlp` или `fork` или возможность наличия нескольких одновременно выполняющихся дочерних процессов (`wait` возвращает идентификатор процесса для закончившегося дочернего процесса, так что можно сравнить его со значением, возвращаеммым `fork`). Наконец, этот фрагмент не имеет никакого отношения к обработке сколь бы то ни было странного поведения со стороны дочернего процесса. Однако эти три строки – основа стандартной функции `system`.

Значение переменной `status`, возвращаемое `wait`, содержит в младших восьми битах представление системы о коде завершения для дочернего процесса; ноль означает нормальное завершение, а ненулевое значение указывает, что возникли какие-либо проблемы. Следующие по старшинству восемь бит берутся из аргумента функции `exit` или оператора `return` в `main`, вызвавшего завершение дочернего процесса.

Когда программа вызывается оболочкой, создаются три дескриптора: `0`, `1` и `2`, которые указывают на соответствующие файлы, а все остальные файловые дескрипторы доступны для использования. Когда эта программа вызывает другую, правила этикета предписывают убедиться, что выполняются те же условия. Ни `fork`, ни `exec` никоим образом не влияют на открытые файлы; оба процесса – и родительский, и дочерний, имеют один и тот же набор открытых файлов. Если родительская программа буферизует вывод, который должен появиться раньше, чем вывод дочерней, то родительская программа должна сбросить содержимое буфера на диск до вызова `execlp`. И наоборот, если родительская программа буферизует входящий поток, то дочерняя потеряет информацию, прочитанную родителем. Вывод может быть сброшен на диск, но невозможно вернуть назад ввод. Эти соображения возникают, если ввод или вывод осуществляется при помощи стандартной

библиотеки ввода-вывода, описанной в главе 6, так как она обычно буферизует и ввод, и вывод.

Наследование дескрипторов файлов в `execlp` может «подвесить» систему: если вызывающая программа не имеет своего стандартного ввода и вывода, связанного с терминалом, то и вызываемая программа не будет его иметь. Возможно, что это как раз то, что нужно; например, в скрипте для редактора `ed` ввод для команды, запущенной с восклицательным знаком `!`, вероятно, будет производиться из скрипта. Даже в этом случае `ed` должен читать ввод посимвольно, чтобы избежать проблем с буферизацией.

Однако для интерактивных программ, таких как `p`, система должна повторно подключить стандартный ввод-вывод к терминалу. Один из способов сделать это – подключить их к `/dev/tty`.

Системный вызов `dup(fd)` дублирует дескриптор файла `fd` в незанятый дескриптор файла с наименьшим номером, возвращая новый дескриптор, который ссылается на тот же самый открытый файл. Этот код связывает стандартный ввод программы с файлом:

```
int fd;

fd = open("file", 0);
close(0);
dup(fd);
close(fd);
```

Вызов `close(0)` освобождает дескриптор `0` (стандартный ввод), но, как всегда, не влияет на родительскую программу.

Приведем версию `system` для интерактивных программ, использующую для сообщений об ошибках `progname`. Можете пока не обращать внимания на части функции, относящиеся к обработке сигналов; вернемся к ним в следующем разделе.

```
/*
 * Более надежная версия system для интерактивных программ
 */
#include <signal.h>
#include <stdio.h>

system(s)   /* выполнить командную строку s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();
    extern char *progname;

    fflush(stdout);
    tty = open("/dev/tty", 2);
    if (tty == -1) {
        fprintf(stderr, "%s: can't open /dev/tty\n", progname);
        return -1;
```

```
    }
    if ((pid = fork()) == 0) {
        close(0); dup(tty);
        close(1); dup(tty);
        close(2); dup(tty);
        close(tty);
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    close(tty);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

Обратите внимание на то, что `/dev/tty` открывается в режиме `2` (чтения и записи), а затем дублируется для создания стандартного ввода и вывода. Именно так система назначает стандартные устройства ввода, вывода и ошибок при входе в нее. Поэтому можно выводить данные в стандартный ввод:

```
$ echo hello 1>&0
hello
$
```

Это означает, что можно было скопировать файловый дескриптор `2`, чтобы повторно подключить стандартный ввод-вывод, но корректнее и надежнее будет открыть `/dev/tty`. Даже у этого варианта `system` есть потенциально возможные проблемы: файлы, открытые в вызывающей программе, как, например, `tty` в программе `ttyin` в `p`, будут переданы дочернему процессу.

Урок не в том, что именно представленная в этом разделе версия `system` должна использоваться во всех ваших программах (она, например, не подошла бы для неинтерактивного `ed`), а в том, чтобы понять, как управлять процессами и как корректно использовать базовые возможности; значение слова «корректно» зависит от конкретного приложения и может не совпадать со стандартной реализацией `system`.

7.5. Сигналы и прерывания

В этом разделе поэтапно рассмотрим процесс обработки сигналов (таких как прерывания), поступающих из внешнего мира, а также ошибок программы. Ошибки программ возникают в основном из-за непра-

вильных обращений к памяти, при выполнении специфических инструкций или из-за операций с плавающей точкой. Наиболее распространенные сигналы, поступающие из внешнего мира: *прерывание* (*interrupt*) – этот сигнал посылается, когда вы нажимаете клавишу *DEL*; *выход* (*quit*) – порождается символом FS (*ctl-*); *отключение* (*hangup*) – вызван тем, что повешена телефонная трубка, и *завершение* (*terminate*) – порождается командой `kill`. Когда происходит одно из вышеуказанных событий, сигнал посылается всем процессам, запущенным с данного терминала, и если не существует соглашений, предписывающих иное, сигнал завершает процесс. Для большинства сигналов создается дамп памяти, который может потребоваться для отладки. (См. `adb(1)` и `sdb(1)`.)

Системный вызов `signal` изменяет действие, выполняемое по умолчанию. Он имеет два аргумента: первый – это номер, который определяет сигнал, второй – это или адрес функции, или же код, предписывающий игнорировать сигнал или восстанавливать действия по умолчанию. Файл `<signal.h>` содержит описания различных аргументов. Так,

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

приводит к игнорированию прерывания, в то время как

```
signal(SIGINT, SIG_DFL);
```

восстанавливает действие по умолчанию – завершение процесса. Во всех случаях `signal` возвращает предыдущее значение сигнала. Если второй аргумент – это имя функции (которая должна быть объявлена в этом же исходном файле), то она будет вызвана при возникновении сигнала. Чаще всего эта возможность используется для того, чтобы позволить программе подготовиться к выходу, например удалить временный файл:

```
#include <signal.h>
char *tempfile = "temp.XXXXXX";

main()
{
    extern onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    mktemp(tempfile);

    /* Обработка ... */

    exit(0);
}

onintr()  /* очистить в случае прерывания */
{
```

```
    unlink(tempfile);
    exit(1);
}
```

Зачем нужны проверка и повторный вызов `signal` в `main`? Вспомните, что сигналы посылаются во *все* процессы, запущенные на данном терминале. Соответственно, когда программа запущена не в интерактивном режиме (а с помощью `&`), командный процессор позволяет ей игнорировать прерывания, таким образом, программа не будет остановлена прерываниями, предназначенными для не фоновых процессов. Если же программа начинается с анонсирования того, что все прерывания должны быть посланы в `onintr`, невзирая ни на что, это сводит на нет попытки командного процессора защитить программу, работающую в фоновом режиме.

Решение, представленное выше, позволяет проверить состояние управления прерываниями и продолжать игнорировать прерывания, если они игнорировались ранее. Код учитывает тот факт, что `signal` возвращает предыдущее состояние конкретного сигнала. И если сигналы ранее игнорировались, процесс будет и далее их игнорировать; в противном случае они должны быть перехвачены.

В более сложной программе может потребоваться перехватить прерывание и интерпретировать его как запрос на отмену выполняемой операции и возврат в ее собственный цикл обработки команд. Возьмем, например, текстовый редактор: прерывание слишком долгой печати не должно приводить к выходу из программы и потере всей сделанной работы. В этой ситуации можно написать такой код:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    setjmp(sjbuf); /* сохранение текущей позиции в стеке*/
for (;;) {
    /* основной цикл обработки */
    }
    ...
}

onintr() /* переустановить в случае прерывания */
{
    signal(SIGINT, onintr); /* переустановить для следующего прерывания */
    printf("\nInterrupt\n");
    longjmp(sjbuf, 0);      /* возврат в сохраненное состояние */ }
```

Файл setjmp.h объявляет тип jmp_buf как объект, в котором может сохраняться положение стека; sjbuf объявляется как объект такого типа. Функция setjmp(3) сохраняет запись о месте выполнения программы. Значения переменных не сохраняются. Когда происходит прерывание, инициируется обращение к программе onintr, которая может напечатать сообщение, установить флаги или сделать что-либо другое. Функция longjmp получает объект, сохраненный в setjmp, и возвращает управление в точку программы, следующую за вызовом setjmp. Таким образом, управление (и положение стека) возвращаются к тому месту основной программы, где происходит вход в основной цикл.

Обратите внимание на то, что сигнал снова устанавливается в onintr, после того как произойдет прерывание. Это необходимо, так как сигналы при их получении автоматически восстанавливают действие по умолчанию.

Некоторые программы просто не могут быть остановлены в произвольном месте, например в процессе обработки сложной структуры данных, поэтому необходимо иметь возможность обнаруживать сигналы. Возможно следующее решение – надо сделать так, чтобы программа обработки прерываний установила флаг и возвратилась обратно вместо того, чтобы вызывать exit или longjmp. Выполнение будет продолжено с того самого места, в котором оно было прервано, а флаг прерывания может быть проверен позже.

С таким подходом связана одна трудность. Предположим, что программа читает с терминала в то время, когда послано прерывание. Надлежащим образом вызывается указанная подпрограмма, которая устанавливает флаги и возвращается обратно. Если бы дело действительно обстояло так, как было указано выше, то есть выполнение программы возобновлялось бы «с того самого места, где оно было прервано», то программа должна была бы продолжать читать с терминала до тех пор, пока пользователь не напечатал бы новую строку. Такое поведение может сбивать с толку, ведь пользователь может и не знать, что программа читает, и он, вероятно, предпочел бы, чтобы сигнал вступал в силу незамедлительно. Чтобы разрешить эту проблему, система завершает чтение, но со статусом ошибки, который указывает, что произошло; errno устанавливается в EINTR, определенный в errno.h, чтобы обозначить прерванный системный вызов.

Поэтому программы, которые перехватывают сигналы и возобновляют работу после них, должны быть готовы к «ошибкам», вызванным прерванными системными вызовами. (Системные вызовы, по отношению к которым надо проявлять осторожность, – это чтение с терминала, ожидание и пауза). Такая программа может использовать код, приведенный ниже, для чтения стандартного ввода:

```
#include <errno.h>
extern int errno;
```

```
...
if (read(0, &c, 1) <= 0)  /* EOF или прерывание */
    if (errno == EINTR) { /* EOF, вызванный прерыванием */
        errno = 0; /* переустановить для следующего раза */
        ...
    } else {       /* настоящий конец файла */
        ...
    }
```

И последняя тонкость, на которую надо обратить внимание, если перехват сигналов сочетается с выполнением других программ. Предположим, что программа обрабатывает прерывания и, к тому же, содержит метод (как ! в ed), посредством которого могут выполняться другие программы. Тогда код будет выглядеть примерно так:

```
if (fork() == 0)
    execlp(...);
signal(SIGINT, SIG_IGN); /* предок игнорирует прерывания */
wait(&status);           /* пока выполняется потомок */
signal(SIGINT, onintr);  /* восстановить прерывания */
```

Почему именно так? Сигналы посылаются всем вашим процессам. Предположим, что программа, которую вы вызвали, обрабатывает свои собственные прерывания, как это делает редактор. Если вы прерываете дочернюю программу, она получит сигнал и вернется в свой основной цикл, и, вероятно, прочитает ваш терминал. Но вызывающая программа также выйдет из состояния ожидания дочерней программы и прочитает ваш терминал. Наличие двух процессов чтения терминала все запутывает, так как на самом деле система «подкидывает монетку», чтобы решить, какая программа получит каждую из строк ввода. Чтобы избежать этого, родительская программа должна игнорировать прерывания до тех пор, пока не выполнится дочерняя. Это умозаключение отражено в обработке сигналов в system:

```
#include <signal.h>

system(s)   /* выполнить командную строку s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();

    ...
    if ((pid = fork()) == 0) {
        ...
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    ...
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
```

```
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

В отступление от описания, функция signal очевидно имеет несколько странный второй аргумент. На самом деле это указатель на функцию, которая возвращает целое число, и это также тип самой функции signal. Два значения, SIG_IGN и SIG_DFL, имеют правильный тип, но выбираются таким образом, чтобы они не совпадали ни с какими возможными реальными функциями. Для особо интересующихся приведем пример того, как они описываются для PDP-11 и VAX; описания должны быть достаточно отталкивающими, чтобы побудить к использованию signal.h.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

Сигналы alarm

Системный вызов alarm(*n*) вызывает отправку вашему процессу сигнала SIGALRM через *n* секунд. Сигнал alarm может применяться для того, чтобы убедиться, что нечто произошло в течение надлежащего промежутка времени если что-то произошло, SIGALRM может быть выключен, если же нет, то процесс может вернуть управление, получив сигнал alarm.

Чтобы пояснить ситуацию, рассмотрим программу, называемую timeout, она запускает некоторую другую команду; если эта команда не закончилась к определенному времени, она будет аварийно прервана, когда alarm выключится. Например, вспомните команду watchfor из главы 5. Вместо того чтобы запускать ее на неопределенное время, можно установить часовой лимит:

```
$ timeout -3600 watchfor dmg &
```

Код в timeout иллюстрирует практически все, о чем говорилось в двух предыдущих разделах. Потомок создан; предок устанавливает аварийный сигнал и ждет, пока потомок закончит свою работу. Если alarm приходит раньше, то потомок уничтожается. Предпринимается попытка вернуть статус выхода потомка.

```
/* timeout: устанавливает временное ограничение для процесса */
#include <stdio.h>
#include <signal.h>
int pid;        /* идентификатор дочернего процесса */
```

```
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int sec = 10, status, onalarm();

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        sec = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc < 2)
        error("Usage: %s [-10] command", progname);
    if ((pid=fork()) == 0) {
        execvp(argv[1], &argv[1]);
        error("couldn't start %s", argv[1]);
    }
    signal(SIGALRM, onalarm);
    alarm(sec);
    if (wait(&status) == -1 || (status & 0177) != 0)
        error("%s killed", argv[1]);
    exit((status >> 8) & 0377);
}

onalarm()   /* завершить дочерний процесс в случае получения alarm */
{
    kill(pid, SIGKILL);
}
```

Упражнение 7.18. Можете ли вы предположить, как реализован `sleep`? Подсказка: `pause(2)`. При каких условиях (если такие условия существуют) `sleep` и `alarm` могут создавать помехи друг для друга? □

История и библиография

В книге не представлено подробного описания реализации системы UNIX, в частности из-за того, что существуют имущественные права на код. Доклад Кена Томпсона (Ken Thompson) «UNIX implementation» (Реализация UNIX), изданный в «BSTJ» в июле 1978 года, описывает основные идеи. Эту же тему поднимают статьи «The UNIX system – a retrospective» (Система UNIX в ретроспективе) в том же номере «BSTJ» и «The evolution of the UNIX time-sharing system» (Эволюция UNIX – системы разделения времени), напечатанная в материалах Symposium on Language Design and Programming Methodology в журнале издательства Springer-Verlag «Lecture Notes in Computer Science» № 79 в 1979 году. Оба труда принадлежат перу Денниса Ритчи (Dennis Ritchie).

Программа `readslow` была придумана Питером Вейнбергером (Peter Weinberger) в качестве простого средства для демонстрации зрителям игры шахматной программы Belle Кена Томсона и Джо Кондона (Joe Condon) во время шахматного турнира. Belle записывала состояние игры в файл; наблюдатели опрашивали файл с помощью `readslow`, чтобы не занимать слишком много драгоценных циклов. (Новая версия оборудования для Belle осуществляет небольшие расчеты на своей главной машине, поэтому больше такой проблемы не существует.)

Том Дафф (Tom Duff) вдохновил нас на написание `spname`. Статья Айвора Дерхема (Ivor Durham), Дэвида Лэмба (David Lamb) и Джеймса Сакса (James Saxe) «Spelling correction in user interfaces» (Проверка орфографии в пользовательских интерфейсах), изданная CACM в октябре 1983 года, представляет несколько отличающийся от привычного проект реализации исправления орфографических ошибок в контексте почтовой программы.

8

Разработка программ

Система UNIX была задумана как среда для разработки программ. В этой главе будут рассмотрены наиболее полезные инструменты разработки. В качестве примера возьмем реальную программу – интерпретатор языка программирования, подобного Бейсику. Этот пример хорошо иллюстрирует, какие проблемы возникают при разработке больших программ. Кроме того, многие программы могут быть представлены как трансляторы, интерпретирующие язык входных данных в некоторую последовательность действий, и поэтому полезно рассказать о средствах разработки языков.

В этой главе мы изучим:

- `yacc` – генератор синтаксических анализаторов, который по описанию грамматики языка генерирует для него синтаксический анализатор;
- `make` – программу, предназначенную для управления процессом компиляции сложных программ;
- `lex` – программу для создания лексических анализаторов, аналогичную `yacc`.

Хотелось бы также обратить внимание на особенности реализации подобных проектов, будет отмечено, что начинать надо с небольшой работающей программы и постепенно расширять ее; будет описана эволюция языка и рассказано о средствах, использованных при реализации.

Процесс разработки языка проведем в шесть этапов, на каждом из которых будет получен логически законченный результат, имеющий самостоятельную ценность. Эти этапы основаны на реальной последовательности написания программы.

1. Калькулятор, выполняющий четыре арифметических действия – `+`, `-`, `*`, `/` – над числами с плавающей точкой, с учетом скобок. Каждое

выражение печатается на отдельной строке, результат выводится немедленно.

2. Переменные с именами от `a` до `z`. Унарный минус и обработка некоторых ошибок.
3. Имена переменных произвольной длины, встроенные функции `sin`, `exp` и т. п., полезные константы, такие как π (записывается как `PI` из-за ограниченных типографских возможностей), и оператор возведения в степень.
4. Внутренние изменения: вместо выполнения «на лету» генерируется код для всех выражений, который затем выполняется. Новых функций нет, но закладывается основа для п. 5.
5. Операторы управления: операторы `if-else` и `while`, объединение операторов в блоки фигурными скобками, операторы сравнения `>`, `<=` и другие.
6. Рекурсивные функции и процедуры, принимающие аргументы. Операторы ввода и вывода строк и чисел.

Получившийся язык описан в главе 9, где на его примере рассмотрено программное обеспечение для подготовки документов. Справочное руководство находится в приложении 2.

Это достаточно длинная глава, так как написание нетривиальной программы требует внимания к очень многим деталям. Предполагается, что читатель знаком с языком Си и имеет под рукой второй том справочного руководства по UNIX, потому что в этой книге просто не хватит места для объяснения всех нюансов. Проявите упорство и будьте готовы перечитать главу дважды. Полный текст окончательной версии приведен в приложении 3 – так легче увидеть, как скомпонована программа.

Авторы потратили немало времени, обсуждая, как назвать этот язык, но так и не придумали ему достойного имени. Мы остановились на названии `hoc` (high-order calculator – высокоорганизованный калькулятор). Соответственно, версии называются `hoc1`, `hoc2` и т. д.

8.1. Этап 1: Калькулятор, выполняющий четыре операции

В этом разделе рассмотрена программа `hoc1`, выполняющая практически те же действия, что и карманный калькулятор, но уступающая ему в мобильности. Она выполняет только четыре действия: `+`, `-`, `*` и `/`, но зато обрабатывает скобки произвольной глубины вложенности, что доступно не всем карманным калькуляторам. Если после ввода выражения нажата клавиша *RETURN*, результат вычислений появляется на следующей строке:

```
$ hoc1
4*3*2
        24
(1+2) * (3+4)
        21
1/2
        0.5
355/113
        3.1415929
-3-4
hoc1: syntax error is on line 5   Унарный минус пока не реализован
$
```

Грамматика

С тех пор как для описания Алгола была использована форма Бэкуса–Наура, языки описываются с помощью формальных грамматик. Абстрактное представление грамматики `hoc1` простое и короткое:

```
list:       expr \n
            list expr \n
expr:       NUMBER
            expr + expr
            expr - expr
            expr * expr
            expr / expr
            ( expr )
```

Другими словами, список (`list`) состоит из последовательности выражений (`expr`), каждое из которых заканчивается символом новой строки (`\n`). Выражение может быть числом либо парой чисел с оператором между ними либо выражением в скобках.

Это неполное описание. Среди прочего, в нем отсутствует определение приоритетов и ассоциативности операторов, нет и разъяснения смысла структурных компонентов. И хотя список определен в терминах выражения, а выражение – в терминах `NUMBER`, само `NUMBER` нигде не определено. Эти детали необходимо уточнить при переходе от схематичного представления к работающей программе.

Обзор yacc

Программа `yacc` – это *генератор синтаксических анализаторов*[1], то есть программа, преобразующая формальное описание грамматики языка, подобное приведенному выше, в программу синтаксического

[1] Имя `yacc` – это аббревиатура «yet another compiler-compiler» (еще один компилятор компиляторов), как назвал его автор, Стив Джонсон (Steve Johnson), комментируя количество аналогичных программ в тот момент (примерно в 1972 году). Но успех имели лишь немногие компиляторы, среди них и `yacc`.

анализа, выполняющую синтаксический разбор выражений. Кроме того, yacc предоставляет возможность связать смысловые значения с грамматическими элементами таким образом, что в процессе синтаксического анализа оценивается также и их семантика. При использовании yacc необходимо соблюдать определенную последовательность действий.

Во-первых, надо описать грамматику, как это было сделано выше, только более подробно. Так определяется синтаксис языка. На этом этапе yacc можно использовать для поиска ошибок и неопределенностей в грамматике.

Во-вторых, каждое *порождающее правило* этой грамматики может быть дополнено действием – описанием того, что должно быть выполнено, когда в разбираемой программе встретится заданная грамматическая форма. Эта часть пишется на Си с соблюдением соглашений о взаимодействии с грамматикой. Так определяется семантика языка.

В-третьих, необходим *лексический анализатор*, который будет читать исходный текст и выделять в нем фрагменты, поддающиеся интерпретации синтаксическим анализатором. Примером такой лексической единицы, состоящей из нескольких символов, является NUMBER, а также односимвольные операторы + и *. Подобный лексический фрагмент традиционно называется *лексемой*.

Наконец, нужна управляющая программа, запускающая синтаксический анализатор, построенный с помощью yacc.

Программа yacc преобразует грамматику и семантические операции в функцию анализатора, называемую yyparse, и размещает сгенерированный код Си в отдельном файле. При отсутствии ошибок синтаксический анализатор, лексический анализатор и управляющая программа могут быть откомпилированы, скомпонованы – возможно, с другими подпрограммами на Си – и выполнены.

В процессе выполнения эта программа вызывает лексический анализатор, выделяющий лексемы, распознает грамматические (синтаксические) структуры и выполняет для каждой из них определенные семантикой действия. Функция лексического анализа должна называться yylex, так как именно это имя используется программой yyparse для получения очередной лексемы. (Все имена, используемые yacc, начинаются с y.)

Входной файл yacc имеет такой вид:

```
%{
необязательная секция
операторы Си, такие как #include, объявления и т. п.
%}
объявления yacc: лексические единицы, грамматические переменные,
приоритеты и ассоциативность
%%
```

```
грамматические правила и действия
%%
дополнительные операторы Си (необязательно):
main() { ...; yyparse(); ... }
yylex() { ... }
...
```

Этот текст обрабатывается yacc, и результат выводится в файл с именем y.tab.c, имеющий формат:

```
операторы Си, расположенные между %{ и %}, если имеются
операторы Си, расположенные после второго %%, если имеются:
main() { ...; yyparse(); ... }
yylex() { ... }
...
yyparse() { синтаксический анализатор, вызывающий yylex() }
```

То, что yacc создает текст на Си, а не скомпилированный объектный (.o) файл, типично для UNIX. Это наиболее гибкий подход – сгенерированный код переносим и может использоваться в любом другом месте.

Сам yacc является мощным средством. Усилия, затраченные на его изучение, окупаются многократно. Сгенерированные им анализаторы компактны, эффективны и корректны (хотя за семантическую обработку отвечает пользователь); большинство рутинных проблем грамматического анализа решаются автоматически. Языковые программы легко создаются и (пожалуй, это более важно) могут быть быстро изменены по мере развития языка.

Этап 1

Исходный текст hoc1 состоит из описания грамматики и действий, лексической подпрограммы yylex и функции main, хранимых в файле hoc.y. (Имена файлов yacc традиционно заканчиваются на .y, но сам yacc не навязывает это обозначение, в отличие от cc и c.) Первая половина hoc.y содержит описание грамматики:

```
$ cat hoc.y
%{
#define YYSTYPE double  /* тип данных для стека yacc */
%}
%token  NUMBER
%left   '+' '-'   /* левоассоциативные, одинаковый приоритет */
%left   '*' '/'   /* левоассоциативные, более высокий приоритет */
%%
list:     /* ничего */
    | list '\n'
    | list expr '\n'    { printf("\t%.8g\n", $2); }
    ;
expr:     NUMBER    { $$ = $1; }
    | expr '+' expr { $$ = $1 + $3; }
```

```
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr { $$ = $1 / $3; }
    | '(' expr ')'  { $$ = $2; }
    ;
%%
    /* конец грамматики */
...
```

В этих нескольких строках немало новой информации. Не будем объяснять здесь все подробности, в частности детали работы синтаксического анализатора, – это описано в руководстве по yacc.

Правила разделяются вертикальной чертой «|». С каждым грамматическим правилом может быть связано действие, которое будет выполняться, когда на входе будет опознано данное правило. Действие описывается последовательностью операторов Си, заключенной в фигурные скобки { и }. Внутри действия переменные $*n* (т. е. $1, $2 и т. д.) ссылаются на значения, возвращаемые *n*-й компонентой правила, а $$ – это значение, возвращаемое правилом в целом. Так, например, в правиле

```
expr:   NUMBER  { $$ = $1; }
```

$1 получает значение, возвращаемое при обработке лексемы NUMBER; это значение возвращается в качестве значения expr. Явное присваивание $$ = $1 может быть опущено, так как $$ всегда принимает значение $1, если ему явно не присвоено что-то другое.

В следующей строке при выполнении правила

```
expr:   expr '+' expr   { $$ = $1 + $3; }
```

значение результата равно сумме значений компонент expr. Обратите внимание, что знаку '+' соответствует $2; пронумерованы все компоненты.

В предыдущей строке выражение, заканчивающееся символом новой строки (\n), воспринимается как список и выводится его значение. Если после этой конструкции ввод прекращается, то процесс разбора корректно завершается. Список (list) может быть и пустой строкой, именно так обрабатываются пустые строки на входе.

В yacc разрешен свободный формат входной информации; здесь представлен рекомендуемый стандарт.

В данной реализации факт распознавания вызывает непосредственное выполнение выражения. В более сложных случаях (включая hoc4 и выше) в процессе разбора создается код, который будет выполнен позднее.

Может оказаться полезным наглядное представление процесса разбора в виде *дерева*, подобно изображенному на рис. 8.1, обход которого в процессе вычислений осуществляется в направлении от листьев к вершине.

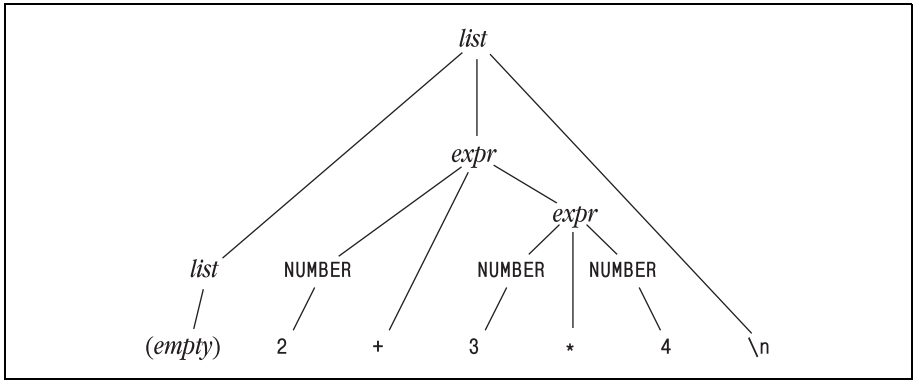

Рис. 8.1. *Дерево разбора для выражения 2 + 3 * 4*

Значения не полностью распознанных правил хранятся в стеке; таким способом они передаются от одного правила к следующему. По умолчанию тип данных в стеке `int`, но здесь он переопределен для обработки чисел с плавающей точкой. Определение

```
#define YYSTYPE double
```

назначает стеку тип `double`.

Синтаксические классы, которые предстоит обрабатывать лексическому анализатору, должны быть объявлены, если только они не являются односимвольными литералами, такими как `'+'` и `'-'`. Объявление `%token` декларирует один или несколько таких объектов. Для определения левой или правой ассоциативности вместо `%token` следует указать `%left` или `%right` соответственно. (Левая ассоциативность означает, что выражение `a-b-c` будет обработано как `(a-b)-c`, а не как `a-(b-c)`.) Приоритет определяется порядком описания: лексемы, объявляемые одновременно, имеют одинаковый приоритет, чем позже объявлена лексема, тем выше ее приоритет. При таком способе грамматика получается неоднозначной (то есть разбор может быть выполнен несколькими способами), но дополнительная информация в объявлениях устраняет неопределенность.

Оставшаяся часть кода состоит из функций второй половины файла `hoc.y`:

Продолжение `hoc.y`

```
#include <stdio.h>
#include <ctype.h>
char    *progname;  /* для сообщений об ошибках */
int lineno = 1;

main(argc, argv)    /* hoc1 */
    char *argv[];
{
    progname = argv[0];
```

```
    yyparse();
}
```

Функция `main` вызывает `yyparse`, выполняющую разбор. Переход от выражения к выражению полностью определяется последовательностью порождающих правил грамматики. С тем же успехом можно было бы вызывать `yyparse` в цикле, определив для `list` в качестве действия вывод результата и немедленный возврат.

В свою очередь, `yyparse` циклически вызывает функцию `yylex` для каждой входной лексемы. В нашем случае `yylex` устроена просто: она пропускает пробелы и табуляции, преобразует последовательности цифр в числовые значения, ведет нумерацию входных строк для сообщений об ошибках, а все прочие символы возвращает без изменений. Поскольку грамматикой определены только символы `+`, `-`, `*`, `/`, `(`, `)` и `\n`, появление любого другого символа заставит `yyparse` сообщить об ошибке. Возвращаемое значение `0` сигнализирует функции `yyparse` о достижении конца файла.

Продолжение `hoc.y`

```
yylex()     /* hoc1 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {   /* число */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    if (c == '\n')
        lineno++;
    return c;
}
```

Переменная `yylval` используется для связи между синтаксическим и лексическим анализаторами, она определена в `yyparse` и имеет тот же тип, что и стек `yacc`. Функция `yyparse` возвращает *тип* лексемы, а ее значение (если имеется) записывает в переменную `yylval`. Например, число с плавающей точкой имеет тип `NUMBER` и значение, например `12.34`. Некоторые лексемы, в частности такие знаки, как `'+'` и `'\n'`, имеют только тип, но не значение. В таких случаях нет необходимости устанавливать значение `yylval`.

Объявление `%token NUMBER` преобразуется в выходном файле `y.tab.c` в директиву `#define`, поэтому константу `NUMBER` можно использовать в любом месте программы на Си. В `yacc` использованы значения, не пересекающиеся с таблицей символов ASCII.

При наличии синтаксической ошибки yyparse вызывает yyerror, передавая строку с сообщением syntax error. Ожидается, что пользователь yacc предоставит какую-нибудь функцию yyerror; наша версия просто передает строку другой функции, warning, а она уже выводит некоторую информацию. В более новых версиях hoc сразу используется warning.

```
yyerror(s)  /* вызывается для обработки синтаксической ошибки yacc */
    char *s;
{
    warning(s, (char *) 0);
}

warning(s, t)   /* выводит предупреждение */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

На этом функции в hoc.y заканчиваются.

Компиляция программы yacc происходит в два этапа:

```
$ yacc hoc.y              Создает y.tab.c
$ cc y.tab.c -o hoc1      Создает исполняемую программу в hoc1
$ hoc1
2/3
        0.66666667
-3-4
hoc1: syntax error near line 1
$
```

Упражнение 8.1. Исследуйте структуру файла y.tab.c. (В hoc1 его длина составляет около 300 строк.) □

Вносим изменения – унарный минус

Было заявлено, что в yacc очень легко проводить изменение языка. В качестве примера введем в hoc1 унарный минус, чтобы выражения типа

```
-3-4
```

вычислялись, а не отбрасывались как синтаксические ошибки.

В hoc.y надо добавить ровно две строчки. В конец раздела приоритетов добавляется новая лексема UNARYMINUS, унарный минус получает наивысший приоритет:

```
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS      /* новое */
```

В грамматику добавляется еще одно порождающее правило для expr:

```
expr:     NUMBER          { $$ = $1; }
     | '-'  expr  %prec UNARYMINUS { $$ = -$2' } /* новое */
```

Определение %prec указывает на то, что знак унарный минус (то есть знак минус перед выражением) имеет приоритет UNARYMINUS (высокий), а действие заключается в изменении знака. Знак минус между двумя выражениями получает приоритет по умолчанию.

Упражнение 8.2. Добавьте в hoc1 операторы % (деление нацело или остаток) и унарный +. Совет: обратитесь к frexp(3). □

Экскурс в make

Вводить две команды для того, чтобы скомпилировать новую версию hoc1, – это невыносимо. Несложно создать командный файл, который бы выполнял эту задачу, но есть и более хороший способ, который в дальнейшем можно будет обобщить для случая, когда исходных файлов в программе несколько. Программа make читает в спецификации, как компоненты программы зависят друг от друга и как их обрабатывать для того, чтобы создать новую версию программы. Она проверяет время изменения различных компонент, вычисляет минимальный объем рекомпиляции, необходимый для получения новой актуальной версии, и запускает процессы. Программа make в состоянии понять замысловатые многоступенчатые процессы типа yacc, поэтому можно помещать такие задачи в спецификацию make, не разбивая их на отдельные шаги.

Разумнее всего применять make, когда создаваемые программы настолько велики, что располагаются в нескольких исходных файлах, но иногда она бывает полезна и для таких небольших программ, как hoc1. Приведем спецификацию make для hoc1, предполагается, что она находится в файле под названием makefile.

```
$ cat makefile
hoc1:    hoc.o
         cc hoc.o -o hoc1
$
```

Здесь написано, что hoc1 зависит от hoc.o и что hoc.o преобразуется в hoc1 посредством запуска компилятора Си cc и помещения выходных данных в hoc1. При этом make уже знает, как преобразовать исходный файл yacc в hoc.y в объектный файл hoc.o:

```
$ make                    Выполнить первое действие в makefile, hoc1
yacc  hoc.y
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
```

```
cc hoc.o -o hoc1
$ make                    Еще раз
`hoc1' is up to date.     make видит, что это не нужно
$
```

8.2. Этап 2: Переменные и обработка ошибок

Следующий (небольшой) шаг заключается в добавлении hoc1 «памяти» для создания hoc2. Дополнительная память представлена 26 переменными с именами от a до z. Этот шаг не отличается особым изяществом, но он полезен в качестве промежуточного и прост в реализации. Введем также обработку некоторых ошибок. Что касается hoc1, его подход к синтаксическим ошибкам таков: выводится соответствующее сообщение, и программа завершается, способ обработки арифметических ошибок (деление на ноль и т. д.) также заслуживает порицания:

```
$ hoc1
1/0
Floating exception - core dumped
$
```

Изменения, которые требуется произвести, чтобы добавить эти новые возможности, невелики – приблизительно 35 строчек кода. Лексический анализатор yylex должен распознавать буквы как переменные; а в грамматику необходимо добавить порождающее правило вида

```
expr:     VAR
        | VAR '=' expr
```

Выражение может содержать присваивание; так возникают множественные присваивания, например

```
x = y = z = 0
```

Простейший способ хранения значений переменных состоит в создании массива из 26 элементов; имя переменной, состоящее из одной буквы, может служить индексом массива. Но если грамматика должна обрабатывать и имена переменных, и значения в одном и том же стеке, то надо проинформировать yacc о том, что его стек содержит объединение элементов типов double и int, а не только double. Такая операция осуществляется при помощи объявления %union, которое помещается в начале файла. Для установки стека в базовый тип (например, double) подходит #define или typedef, но для объединения типов необходимо использовать механизм %union, потому что yacc проверяет непротиворечивость таких выражений, как $$=$2.

Представим грамматическую часть hoc.y для hoc2:

```
$ cat hoc.y
%{
```

```
double  mem[26];    /* память для переменных 'a'..'z' */
%}
%union {        /* тип стека */
    double  val;    /* фактическое значение */
    int index;  /* индекс в mem[] */
}
%token  <val>   NUMBER
%token  <index> VAR
%type   <val>   expr
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%%
list:     /* ничего */
    | list '\n'
    | list expr '\n'    { printf("\t%.8g\n", $2); }
    | list error '\n'   { yyerrok; }
    ;
expr:     NUMBER
    | VAR       { $$ = mem[$1]; }
    | VAR '=' expr  { $$ = mem[$1] = $3; }
    | expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0.0)
            execerror("division by zero", "");
        $$ = $1 / $3; }
    | '(' expr ')'  { $$ = $2; }
    | '-' expr  %prec UNARYMINUS  { $$ = -$2; }
    ;
%%
    /* конец грамматики */
...
```

Объявление `%union` сообщает, что элементы стека хранят или `double` (число, обычный случай), или `int`, который является индексом массива `mem`. В объявления `%token` добавлен индикатор типа. Объявление `%type` указывает, что `expr` представляет собой член объединения `<val>`, то есть принадлежит к типу `double`. Информация о типе позволяет `yacc` генерировать ссылку на соответствующий член объединения. Обратите внимание на то, что `=` является правоассоциативным, в то время как другие операторы левоассоциативны.

Обработка ошибок введена в нескольких местах. Самой очевидной является проверка делителя на равенство нулю; если он равен нулю, то вызывается функция `execerror`.

Еще одна проверка отлавливает сигнал об исключительной ситуации при операции с плавающей точкой, который означает, что произошло

переполнение при выполнении операции с плавающей точкой. Обработчик сигнала устанавливается в `main`.

Чтобы закончить с обработкой ошибок, добавим порождающее правило для `error`. В грамматике `yacc` слово `error` является зарезервированным, оно предоставляет способ упреждения синтаксических ошибок и восстановления после них. Если происходит ошибка, `yacc` в конце концов пытается использовать это порождающее правило, признает ошибку грамматически «правильной» и тогда устраняет ее. Действие `yyerrok` устанавливает в синтаксическом анализаторе (parser) флаг, который разрешает возврат в состояние, поддающееся интерпретации синтаксическим анализатором. Устранение ошибок не отличается простотой в любом синтаксическом анализаторе, поэтому знайте, что сейчас были предприняты лишь самые элементарные меры. Возможности `yacc` также были рассмотрены очень поверхностно.

Действия в грамматике `hoc2` не претерпели значительных изменений. Ниже приведена функция `main`, в которую была добавлена функция `setjmp` для сохранения «чистого» состояния, из которого производится возобновление работы после ошибки. Функция `execerror` вызывает соответствующую `longjmp`. (См. описание `setjmp` и `longjmp` в разделе 7.5.)

```
...
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

main(argc, argv)    /* hoc2 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}

execerror(s, t) /* восстановление после ошибки времени исполнения */
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch()  /* перехват исключений в операциях с плавающей точкой */
{
    execerror("floating point exception", (char *) 0);
}
```

Было решено, что для удобства отладки `execerror` будет вызывать `abort` (см. `abort(3)`), создающую дамп оперативной памяти, который может

быть просмотрен с помощью adb или sdb. Когда программа станет достаточно устойчивой, вызов abort будет заменен на longjmp.

Лексический анализатор hoc2 несколько отличается от hoc1. Введена дополнительная проверка на буквы в нижнем регистре, и, так как yylval теперь имеет тип union, соответствующий член объединения должен быть определен до выхода из yylex. Вот изменившиеся фрагменты:

```
yylex()         /* hoc2 */
...
if (c == '.' || isdigit(c)) {   /* число */
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    if (islower(c)) {
        yylval.index = c - 'a'; /* только ASCII */
        return VAR;
    }
```

Еще раз обратите внимание на различие типа лексемы (например, NUMBER) и ее значения (например, 3.1416).

Теперь покажем на примере, как выглядят нововведения hoc2: переменные и обработка ошибок:

```
$ hoc2
x = 355
        355
y = 113
        113
p = x/z                     z не определена и поэтому равна нулю
hoc2: division by zero near line 4      Обработка ошибки
x/y
        3.1415929
1e30 * 1e30                 Переполнение
hoc2: floating point exception near line 5
...
```

Для того чтобы обнаружить переполнение при выполнении операции с плавающей точкой на PDP-11, требуются дополнительные средства, но на большинстве других машин hoc2 работает в том виде, как показано выше.

Упражнение 8.3. Добавьте возможность запоминать последнее вычисленное значение, чтобы не приходилось заново вводить его в последовательности расчетов. В качестве решения можно предложить сделать его переменной, например p (от previous – предыдущий). □

Упражнение 8.4. Измените hoc так, чтобы точку с запятой можно было использовать как символ, завершающий выражения, эквивалентный символу новой строки. □

8.3. Этап 3: Произвольные имена переменных; встроенные функции

В новую версию, hoc3, добавляется несколько важных новых возможностей и соответствующий объем дополнительного кода. Самым главным новым свойством является доступ к встроенным функциям:

```
sin     cos     atan    exp     log     log10
sqrt    int     abs
```

Также добавлен оператор возведения в степень ^; он обладает наивысшим приоритетом и является правоассоциативным.

Лексическому анализатору приходится справляться со встроенными именами, длина которых превышает один символ, поэтому не требуется особых усилий для того, чтобы разрешить произвольную длину имен переменных. Для того чтобы отслеживать эти переменные, потребуется более сложная таблица символов, но как только она создана, можно предварительно загрузить в нее имена и значения некоторых полезных констант:

```
PI       3.14159265358979323846   π
E        2.71828182845904523536   Основание натурального логарифма
GAMMA    0.57721566490153286060   Константа Эйлера-Маскерони
DEG     57.29577951308232087680   Градусов в радиане
PHI      1.61803398874989484820   Золотое отношение
```

Получается удобный калькулятор:

```
$ hoc3
1.5^2.3
        2.5410306
exp(2.3*log(1.5))
        2.5410306
sin(PI/2)
        1
atan(1)*DEG
        45
...
```

Поведение программы в целом также было несколько улучшено. В hoc2 в случае присваивания x=*expr* происходило не только собственно присваивание, но и вывод значения, поскольку все выражения выводятся:

```
$ hoc2
x = 2 * 3.14159
        6.28318      Для присваивания переменной выводится значение
...
```

В hoc3 вводится различие между присваиваниями и выражениями; теперь значения печатаются только для выражений:

```
$ hoc3
```

```
x = 2 * 3.14159      Присваивание: значение не выводится
x                    Выражение:
        6.28318          значение выводится
...
```

В результате всех этих изменений программа становится достаточно большой (около 250 строк), поэтому лучше разбить ее на отдельные файлы для обеспечения удобства редактирования и более быстрой компиляции. Теперь у нас не один файл, а пять:

`hoc.y`	Грамматика, `main`, `yylex` (как и раньше)
`hoc.h`	Глобальные структуры данных
`symbol.c`	Функции таблицы символов: `lookup`, `install`
`init.c`	Встроенные функции и константы; `init`
`math.c`	Интерфейсы к математическим функциям: `Sqrt`, `Log` и т. д.

Поэтому необходимо научиться организовывать многофайловую программу на Си и подробнее изучить `make`, чтобы заставить ее поработать для нас.

Вернемся ненадолго к `make`. Сначала посмотрим на код символьной таблицы. У символа есть имя, тип (или `VAR`, или `BLTIN`) и значение. Если символ принадлежит к типу `VAR`, то его значение имеет тип `double`; если же символ является встроенной функцией, то его значение – это указатель на функцию, которая возвращает `double`. Наличие такой информации необходимо в файлах `hoc.y`, `symbol.c` и `init.c`. Можно было бы просто создать три копии, но при этом слишком уж просто сделать ошибку или забыть обновить один из экземпляров при внесении изменений. Вместо этого поместим общую информацию в заголовочный файл `hoc.h`, а любой файл, нуждающейся в такой информации, будет включать его в себя. (Суффикс `.h` присутствует традиционно, но ни одна из программ не принуждает к использованию именно такого обозначения.) В `makefile`, кроме того, добавлены записи о зависимости файлов от `hoc.h` – так, чтобы они компилировались, когда он изменяется.

```
$ cat hoc.h
typedef struct Symbol { /* элемент таблицы символов */
    char    *name;
    short   type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;        /* if VAR */
        double  (*ptr)();   /* if BLTIN */
    } u;
    struct Symbol   *next;  /* чтобы связать с другим */
} Symbol;
Symbol  *install(), *lookup();
$
```

Тип `UNDEF` – это `VAR`, которому еще не присвоено значение.

Символы связаны вместе в список при помощи поля `next` структуры `Symbol`. Сам список является локальным по отношению к `symbol.c`; получить к нему доступ можно только через функции `lookup` и `install`. Благодаря этому изменение таблицы символов (при необходимости) не составляет труда. (Однажды это было проделано.) Функция `lookup` просматривает список в поиске конкретного имени и возвращает указатель на `Symbol` с указанным именем, если оно найдено, и ноль в противном случае. Таблица символов использует линейный поиск, который полностью подходит для нашего интерактивного калькулятора, т. к. поиск переменных происходит только во время синтаксического разбора, но не во время выполнения. Функция `install` помещает переменную с соотнесенными ей типом и значением в начало списка. Функция `emalloc` вызывает `malloc`, стандартную функцию распределения памяти (`malloc(3)`), и проверяет результат. Эти три фунции являются содержимым файла `symbol.c`. Файл `y.tab.h` порождается выполнением `yacc -d`; он содержит директивы `#define`, которые `yacc` сгенерировал для таких лексем, как `NUMBER`, `VAR`, `BLTIN` и т. д.

```
$ cat symbol.c
#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0;  /* таблица символов: связанный список */

Symbol *lookup(s)   /* поиск s в таблице символов */
    char *s;
{
    Symbol *sp;

    for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
        if (strcmp(sp->name, s) == 0)
            return sp;
    return 0;   /* 0 ==> не найдено */
}

Symbol *install(s, t, d)  /* внести s в таблицу символов */
    char *s;
    int t;
    double d;
{
    Symbol *sp;
    char *emalloc();

    sp = (Symbol *) emalloc(sizeof(Symbol));
    sp->name = emalloc(strlen(s)+1); /* +1 для '\0' */
    strcpy(sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symlist; /* поместить в начало списка */
    symlist = sp;
    return sp;
}
```

```
char *emalloc(n)    /* проверить значение, возвращенное malloc */
    unsigned n;
{
    char *p, *malloc();

    p = malloc(n);
    if (p == 0)
        execerror("out of memory", (char *) 0);
    return p;
}
$
```

Файл `init.c` **содержит определения констант (**`PI` **и т. д.) и указатели для встроенных функций; они вносятся в таблицу символов функцией** `init`**, которую вызывает** `main`**.**

```
$ cat init.c
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Exp(), Sqrt(), integer();

static struct {     /* Константы */
    char    *name;
    double  cval;
} consts[] = {
    "PI",    3.14159265358979323846,
    "E",     2.71828182845904523536,
    "GAMMA", 0.57721566490153286060,  /* постоянная Эйлера */
    "DEG",  57.29577951308232087680,  /* градусов/радиан */
    "PHI",   1.61803398874989484820,  /* золотое отношение */
    0,   0
};

static struct {     /* Встроенные функции */
    char    *name;
    double  (*func)();
} builtins[] = {
    "sin",  sin,
    "cos",  cos,
    "atan", atan,
    "log",  Log,    /* проверка аргумента */
    "log10", Log10, /* проверка аргумента */
    "exp",  Exp,    /* проверка аргумента */
    "sqrt", Sqrt,   /* проверка аргумента */
    "int",  integer,
    "abs",  fabs,
    0,  0
};

init()  /* вставить константы и встроенные функции в таблицу */
{
```

```
    int i;
    Symbol *s;

    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        s = install(builtins[i].name, BLTIN, 0.0);
        s->u.ptr = builtins[i].func;
    }
}
```

Данные хранятся в таблицах, а не вносятся в код, потому что таблицы легче читать и изменять. Таблицы объявляются как `static`, поэтому они видны только внутри файла, а не во всей программе. К математическим функциям (`Log`, `Sqrt` и т. д.) мы вскоре вернемся.

Теперь, когда фундамент заложен, можно перейти к изменениям грамматики, которые на нем построены.

```
$ cat hoc.y
%{
#include "hoc.h"
extern  double  Pow();
%}
%union {
    double  val;    /* фактическое значение */
    Symbol  *sym;   /* указатель на таблицу символов */
}
%token  <val>   NUMBER
%token  <sym>   VAR BLTIN UNDEF
%type   <val>   expr asgn
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^' /* возведение в степень */
%%
list:     /* ничего */
    | list '\n'
    | list asgn '\n'
    | list expr '\n'    { printf("\t%.8g\n", $2); }
    | list error '\n'   { yyerrok; }
    ;
asgn:     VAR '=' expr { $$=$1->u.val=$3; $1->type = VAR; }
    ;
expr:     NUMBER
    | VAR { if ($1->type == UNDEF)
              execerror("undefined variable", $1->name);
        $$ = $1->u.val; }
    | asgn
    | BLTIN '(' expr ')'    { $$ = (*($1->u.ptr))($3); }
    | expr '+' expr { $$ = $1 + $3; }
```

```
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0.0)
            execerror("division by zero", "");
        $$ = $1 / $3; }
    | expr '^' expr { $$ = Pow($1, $3); }
    | '(' expr ')'  { $$ = $2; }
    | '-' expr  %prec UNARYMINUS  { $$ = -$2; }
    ;
%%
    /* конец грамматики */
...
```

В грамматике вдобавок к `expr` появилось определение `asgn` (для присваивания); строка ввода, содержащая только

```
VAR = expr
```

представляет собой присваивание, и поэтому значение не выводится. Кстати, обратите внимание на то, как легко было добавить в грамматику возведение в степень, обладающее к тому же ассоциативностью справа.

Изменился описатель стека `%union`: вместо ссылки на переменную по ее индексу в 26-элементной таблице используется указатель на объект типа `Symbol`. Заголовочный файл `hoc.h` содержит определение этого типа.

Лексический анализатор распознает имена переменных, просматривает их в таблице символов и решает, являются ли они переменными (`VAR`) или встроенными функциями (`BLTIN`). Значение, возвращаемое `yylex`, принадлежит одному из этих типов; а переменные, определяемые пользователем, и предопределенные переменные (такие как `PI`) имеют тип `VAR`.

Одним из свойств переменной является то, присвоено ли ей значение, поэтому использование неопределенной переменной считается ошибкой, о чем и сообщает `yyparse`. Проверка на наличие у переменной значения должна производиться грамматическим, а не лексическим анализатором. Когда тип `VAR` распознается лексически, его контекст нам еще не известен; не хотелось бы слышать жалобы на то, что переменная `x` не определена в то время, когда она находится в разрешенном контексте, например в левой части присваивания, как `x=1`.

Вот исправленная часть функции `yylex`:

```
yylex()         /* hoc3 */
...
if (isalpha(c)) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
```

```
        do {
            *p++ = c;
        } while ((c=getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
            s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
    }
...
```

В `main` добавлена одна новая строка, которая вызывает функцию инициализации `init`, чтобы та вставила встроенные функции и предопределенные имена (`PI` и т. д.) в таблицу символов.

```
main(argc, argv)    /* hoc3 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}
```

Осталось рассказать только о файле `math.c`. В некоторых математических функциях необходим интерфейс для обработки ошибок, например стандартная функция `sqrt`, не выдавая никаких сообщений об ошибках, просто возвращает ноль, если ее аргумент отрицателен. В коде файла `math.c` есть проверка на наличие ошибки, заимствованная из раздела 2 справочного руководства по UNIX, см. главу 7. Это решение характеризуется большей надежностью и переносимостью, чем тесты, которые можно написать самостоятельно, т. к. ограниченность, присущая стандартным функциям, лучше всего отражена в «официальном» коде. Заголовочный файл `math.h` содержит объявления типов для стандартных математических функций. Заголовочный файл `errno.h` содержит имена возможных ошибок.

```
$ cat math.c
#include <math.h>
#include <errno.h>
extern  int errno;
double  errcheck();

double Log(x)
    double x;
{
    return errcheck(log(x), "log");
```

```
}
double Log10(x)
    double x;
{
    return errcheck(log10(x), "log10");
}

double Sqrt(x)
    double x;
{
    return errcheck(sqrt(x), "sqrt");
}

double Exp(x)
    double x;
{
    return errcheck(exp(x), "exp");
}
double Pow(x, y)
    double x, y;
{
    return errcheck(pow(x,y), "exponentiation");
}

double integer(x)
    double x;
{
    return (double)(long)x;
}

double errcheck(d, s)   /* проверка результата обращения к библиотеке*/
    double d;
    char *s;
{
    if (errno == EDOM) {
        errno = 0;
        execerror(s, "argument out of domain");
    } else if (errno == ERANGE) {
        errno = 0;
        execerror(s, "result out of range");
    }
    return d;
}
$
```

Если запустить yacc **на новой грамматике, появляется интересное (и грамматически неправильное) диагностическое сообщение:**

```
$ yacc hoc.y

conflicts: 1 shift/reduce
$
```

Сообщение «shift/reduce» (сдвиг/свертка) означает, что грамматика `hoc3` неоднозначна: единственная строка ввода

```
x = 1
```

может быть синтаксически проанализирована двумя способами (рис. 8.2).

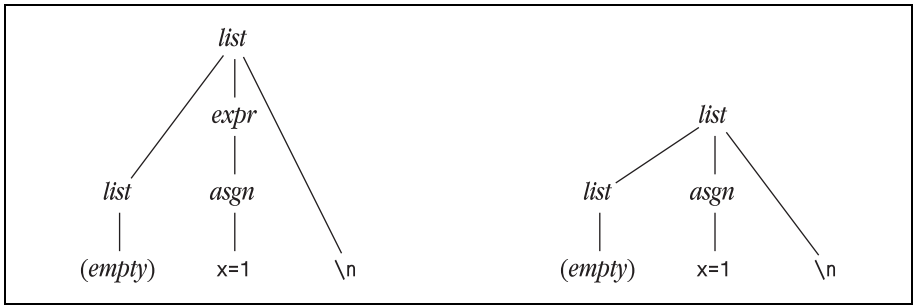

***Рис. 8.2.** Варианты синтаксического анализа грамматики hoc3*

Синтаксический анализатор может посчитать, что *asgn* должно быть сведено к *expr*, а затем к *list*, следуя левому дереву грамматического разбора, а может решить, что надо сразу же использовать следующий `\n` («сдвиг») и преобразовать все в *list* без промежуточных правил, как на правом дереве. Сталкиваясь с такой неоднозначной ситуацией, `yacc` выбирает сдвиг, потому что в реально существующих грамматиках именно такой способ является верным. Следует научиться понимать сообщения, подобные приведенному выше, чтобы не сомневаться в том, правильный ли сделан выбор.[1] Если запустить `yacc` с параметром `-v`, будет выведен значительного размера файл `y.output`, в котором даны пояснения об источнике конфликтов.

Упражнение 8.5. В своем нынешнем виде `hoc3` разрешает операции, подобные

```
PI = 3
```

Хорошо ли это? Как изменить `hoc3` так, чтобы запретить присваивание «константам»? □

Упражнение 8.6. Добавьте встроенную функцию `atan2(y,x)`, которая возвращала бы угол, тангенс которого равен `y/x`. Добавьте встроенную функцию `rand()`, которая возвращала бы случайное число с плавающей точкой, равномерно распределенное по интервалу (0, 1). Как изменить грамматику таким образом, чтобы встроенные функции могли иметь различное количество аргументов? □

[1] Сообщение `yacc` «reduce/reduce conflict» (конфликт свертка/свертка) информирует о серьезной проблеме, чаще всего оно является симптомом наличия ошибки в грамматике, а не внутренней неоднозначности.

Упражнение 8.7. Как добавить возможность выполнения команд без выхода из hoc (подобно ! в других UNIX-программах)? □

Упражнение 8.8. Исправьте код в файле math.c так, чтобы использовать таблицу вместо множества по существу идентичных функций. □

Снова о make

Программа для hoc3 теперь занимает не один, а пять файлов, поэтому описание в makefile тоже будет более сложным.

```
$ cat makefile
YFLAGS = -d     # вынуждает создание y.tab.h
OBJS = hoc.o init.o math.o symbol.o # аббревиатура

hoc3:   $(OBJS)
    cc $(OBJS) -lm -o hoc3

hoc.o:  hoc.h

init.o symbol.o:    hoc.h y.tab.h

pr:
    @pr hoc.y hoc.h init.c math.c symbol.c makefile

clean:
    rm -f $(OBJS) y.tab.[ch]
$
```

Строка YFLAGS = -d добавляет параметр -d в командную строку yacc, генерируемую make; он указывает yacc, что требуется вывести файл y.tab.h с директивами #define. Строка OBJS=... вводит краткую запись для конструкции, которая неоднократно будет использоваться в дальнейшем. Синтаксис отличается от синтаксиса переменных оболочки – обязательными являются круглые скобки. Флаг -lm вызывает просмотр математической библиотеки в поиске математических функций.

Теперь hoc3 зависит от четырех файлов .o, при этом некоторые из них зависят от файлов .h. После того как эти взаимозависимости установлены, make может сделать вывод о том, какая рекомпиляция необходима, после внесения изменений в любой из рассматриваемых файлов. Если хочется просто посмотреть, что будет делать make, не запуская процессы по-настоящему, введите

```
$ make -n
```

С другой стороны, если надо привести время создания файлов в согласованное состояние, то параметр -t (touch – дотронуться) обновит их, не осуществляя рекомпиляции.

Обратите внимание на то, что добавлено было не только множество взаимозависимостей исходных файлов, также были введены различные сервисные программы, все они аккуратно собраны в одном месте. По умолчанию make выполняет первое, что написано в makefile, но если

указать имя элемента, который обозначает правило зависимостей (например, `symbol.o` или `pr`), то будет выполнен этот элемент. Пустая зависимость понимается как указание на то, что элемент никогда не бывает «новейшим», поэтому при каждом требовании действие будет выполняться. Например,

```
$ make pr | lpr
```

выводит запрошенные данные на построчно печатающий принтер. (Начальный символ `@` в `@pr` подавляет отображение на экране команд, которые выполняет `make`.) А команда

```
$ make clean
```

удаляет выходные файлы `yacc` и файлы `.o`.

В качестве способа хранения всех относящихся к делу расчетов в одном файле механизм пустых зависимостей во многих случаях предпочтительнее, чем использование командного файла. К тому же область применения `make` не ограничивается разработкой программ, она весьма полезна и для выполнения множества операций, имеющих временные зависимости.

Экскурс в lex

Программа `lex` создает лексические анализаторы (lexical analyzers) подобно тому, как `yacc` создает синтаксические (parsers): программист пишет спецификацию лексических правил языка, используя регулярные выражения и фрагменты Си, которые должны выполняться, когда найдена соответствующая строка; `lex` преобразовывает ее в распознаватель (recognizer). Программы `lex` и `yacc` взаимодействуют по тому же самому механизму, что и лексические анализаторы, которые мы уже писали. Не будем обсуждать тонкости `lex`; этот разговор вообще заведен в основном для того, чтобы заинтересовать вас и побудить к дальнейшему изучению. Подробная информация по работе с `lex` представлена в томе 2B справочного руководства по UNIX.

Для начала представим программу `lex` из файла `lex.l`; она заменит применявшуюся до этого момента функцию `yylex`.

```
$ cat lex.l
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t]   { ; }   /* пропускать пробелы и табуляции */
[0-9]+\.?|[0-9]*\.[0-9]+ {
    sscanf(yytext, "%lf", &yylval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
    Symbol *s;
```

```
    if ((s=lookup(yytext)) == 0)
        s = install(yytext, UNDEF, 0.0);
    yylval.sym = s;
    return s->type == UNDEF ? VAR : s->type; }
\n  { lineno++; return '\n'; }
.   { return yytext[0]; }        /* все остальное */
$
```

Каждое «правило» представляет собой регулярное выражение, оно похоже на выражения egrep или awk, за исключением того, что lex распознает управляющие символы в стиле Си, такие как \t и \n. Действие заключается в фигурные скобки. Правила проверяются одно за другим, а конструкции типа * и + соответствуют строке максимально возможной длины. Если правило соответствует следующей части ввода, то действие выполняется. Строка ввода, для которой было найдено соответствие, доступна в строке lex, которая называется yytext.

Для того чтобы использовать lex, надо изменить makefile::

```
$ cat makefile
YFLAGS = -d
OBJS = hoc.o lex.o init.o math.o symbol.o

hoc3:   $(OBJS)
    cc $(OBJS) -lm -ll -o hoc3

hoc.o:  hoc.h

lex.o init.o symbol.o:  hoc.h y.tab.h
...
$
```

Программа make знает, как перейти от файла .l к надлежащему .o; все, что ей нужно от нас – это информация о зависимостях. (Еще надо добавить lex-библиотеку -ll к списку, просматриваемому cc, потому что распознаватель, генерируемый lex, не является автономным.[1]) Вывод полностью автоматический и выглядит впечатляюще:

```
$ make
yacc -d hoc.y

conflicts: 1 shift/reduce
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
```

[1] Пользователям свободно распространяемых UNIX-подобных операционных систем необходимо проверить, каким вариантом lex они на самом деле пользуются (это можно сделать, обратившись к справочному руководству по lex). Скорее всего, окажется, что для построения лексических анализаторов в системе предлагается вариант программы, называемый flex. В этом случае в списке библиотек для поиска необходимо заменить -ll на -lfl. – *Примеч. науч. ред.*

```
lex  lex.l
cc  -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc  -c init.c
cc  -c math.c
cc  -c symbol.c
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -o hoc3
$
```

Если изменился один файл, то для создания новой версии достаточно одной команды:

```
$ touch lex.l              Изменяет время изменения lex.l
$ make
lex  lex.l
cc  -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc hoc.o lex.o init.o math.o symbol.o -ll -lm -o hoc3
$
```

Мы долго спорили о том, считать ли lex отступлением от основной темы и описать этот язык кратко или же представить его как основной инструмент лексического анализа для сложных языков. Каждый вариант имеет свои «за» и «против». Основная проблема, порождаемая применением lex (не считая того, что пользователю приходится учить еще один язык), состоит в том, что у него есть тенденция к снижению скорости выполнения и к созданию распознавателей (recognizers), которые больше и медленнее, чем их эквиваленты на Си. К тому же бывает тяжело настроить ввод lex, если выполняется что-то необычное, например обработка ошибок или ввод из файлов. В контексте hoc это не так уж важно. Ограниченный объем книги не позволяет привести описание версии анализатора на основе lex, поэтому (к сожалению) для последующего лексического анализа вернемся к Си. Самостоятельная реализация версии на lex будет хорошим упражнением.

Упражнение 8.9. Сравните размеры двух версий hoc3. Подсказка: size(1). □

8.4. Этап 4: Строим вычислительную машину

Мы двигаемся по направлению к hoc5, интерпретатору для языка с управляющей логикой. Промежуточным этапом на этом пути является создание hoc4, эта программа обеспечивает те же функции, что и hoc3, но реализована она в рамках интерпретатора hoc5. На самом деле hoc4 написана именно так, потому что в этом случае у нас есть две программы, которые должны вести себя идентично, что удобно для отладки. По мере синтаксического анализа входных данных hoc4 генерирует

код для простого компьютера вместо того, чтобы сразу же вычислять ответы. Когда разбор оператора закончен, порожденный код выполняется (интерпретируется) для подсчета результата.

Простой компьютер – это *машина со стековой организацией*: когда встречается операнд, он помещается в стек (точнее, генерируется код для проталкивания его в стек); большинство операторов работают с элементами, находящимися на вершине стека. Например, для обработки присваивания

```
x = 2 * y
```

порождается следующий код:

```
constpush       Протолкнуть константу в стек
   2            ... константу 2
varpush         Протолкнуть указатель на символьную таблицу в стек
   y            ... для переменной y
eval            Оценить: заменить указатель значением
mul             Перемножить два верхних элемента; произведение
                  занимает их место
varpush         Протолкнуть указатель на символьную таблицу в стек
   x            ... для переменной x
assign          Сохранить значение в переменной, вытолкнуть указатель
pop             Очистить значение вершины стека
STOP            Конец последовательности команд
```

Когда выполняется код, выражение вычисляется и результат сохраняется в `x`, как указано в комментариях. Заключительный вызов `pop` убирает значение из стека, потому что больше оно не потребуется.

Машины со стековой организацией обычно приводят к простым интерпретаторам, и наш случай не является исключением – это просто массив, содержащий операторы и операнды. Операторы являются машинными командами; каждый из них представляет собой вызов функции с аргументами (если они заданы), которые следуют за командой. В стеке уже могут находиться другие операнды, как в примере, рассмотренном выше.

Код таблицы символов для `hoc4` идентичен коду для `hoc3`; инициализация в `init.c` и математические функции в `math.c` также не отличаются от предыдущей версии. Грамматика такая же, как в `hoc3`, а вот действия совершенно другие. По существу, каждое действие порождает машинные команды и все сопутствующие им аргументы. Например, три позиции порождаются для `VAR` в выражении: команда `varpush`, указатель на таблицу символов для переменной и команда `eval`, которая при исполнении заменит указатель на таблицу символов значением. Код для `*` (умножения) состоит из одного слова `mul`, т. к. операнды уже находятся в стеке.

```
$ cat hoc.y
%{
```

```
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol  *sym;   /* указатель на таблицу символов */
    Inst    *inst;  /* машинная команда */
}
%token  <sym>   NUMBER VAR BLTIN UNDEF
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^' /* возведение в степень */
%%
list:     /* ничего */
    | list '\n'
    | list asgn '\n'  { code2(pop, STOP); return 1; }
    | list expr '\n'  { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }
    ;
asgn:     VAR '=' expr  { code3(varpush,(Inst)$1,assign); }
    ;
expr:     NUMBER    { code2(constpush, (Inst)$1); }
    | VAR       { code3(varpush, (Inst)$1, eval); }
    | asgn
    | BLTIN '(' expr ')' { code2(bltin, (Inst)$1->u.ptr); }
    | '(' expr ')'
    | expr '+' expr { code(add); }
    | expr '-' expr { code(sub); }
    | expr '*' expr { code(mul); }
    | expr '/' expr { code(div); }
    | expr '^' expr { code(power); }
    | '-' expr  %prec UNARYMINUS  { code(negate); }
    ;
%%
    /* конец грамматики */
...
```

`Inst` — это тип данных машинных команд (указатель на функцию, возвращающую `int`), о нем еще будет сказано дальше. Обратите внимание на то, что аргументами `code` являются названия функций, то есть указатели на функции или другие значения, которые приведены к указателям на функции.

Несколько изменена функция `main`. Синтаксический анализатор теперь возвращается после каждого оператора или выражения; порождаемый им код выполняется. В конце файла `yyparse` возвращает ноль.

```
main(argc, argv)    /* hoc4 */
    char *argv[];
```

```
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); yyparse(); initcode())
        execute(prog);
    return 0;
}
```

Лексический анализатор мало изменился. Основное отличие состоит в том, что теперь числа не используются незамедлительно, а сохраняются. Проще всего реализовать это, поместив их в таблицу символов вместе с переменными. Представим измененную часть `yylex`:

```
yylex()         /* hoc4 */
...
if (c == '.' || isdigit(c)) {   /* число */
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
...
```

Каждый элемент стека интерпретатора является либо значением с плавающей точкой, либо указателем на элемент таблицы символов; тип данных стека – это объединение типов, перечисленных выше. Сама машина представляет собой массив указателей или на функции (`mul` и т. д.), выполняющие операцию, или на данные в таблице символов. В заголовочный файл `hoc.h` добавлены описанные структуры данных и объявления функций, чтобы они были видны в любом месте программы (при необходимости). (Кстати, все изменения были помещены в один файл, а не в два. Для больших программ лучше распределить заголовочную информацию по нескольким файлам – так, чтобы каждый файл присоединялся только тогда, когда это действительно необходимо.)

```
$ cat hoc.h
typedef struct Symbol { /* элемент таблицы символов */
    char    *name;
    short   type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;        /* if VAR */
        double  (*ptr)();   /* if BLTIN */
    } u;
    struct Symbol   *next;  /* связывает с другим */
} Symbol;
Symbol  *install(), *lookup();
```

```
typedef union Datum {   /* тип стека интерпретатора */
    double  val;
    Symbol  *sym;
} Datum;
extern  Datum pop();

typedef int (*Inst)();  /* машинная команда */
#define STOP    (Inst) 0

extern  Inst prog[];
extern  eval(), add(), sub(), mul(), div(), negate(), power();
extern  assign(), bltin(), varpush(), constpush(), print();
$
```

Функции, которые выполняют машинные команды и управляют стеком, хранятся в новом файле, `code.c`**. Его длина составляет около 150 строк, поэтому он будет приведен по частям.**

```
$ cat code.c
#include "hoc.h"
#include "y.tab.h"

#define NSTACK  256
static  Datum   stack[NSTACK];  /* стек */
static  Datum   *stackp;    /* следующая свободная ячейка стека */

#define NPROG   2000
Inst    prog[NPROG];    /* машина */
Inst    *progp;     /* следующая свободная ячейка для генерирования кода */
Inst    *pc;        /* счетчик команд во время исполнения */

initcode()  /* инициализация для генерирования кода */
{
    stackp = stack;
    progp = prog;
}
...
```

Стек управляется вызовами функций `push` **и** `pop`**:**

```
push(d)     /* втолкнуть d в стек */
    Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack overflow", (char *) 0);
    *stackp++ = d;
}
Datum pop() /* вытолкнуть верхние элементы из стека и вернуть */
{
    if (stackp <= stack)
        execerror("stack underflow", (char *) 0);
    return *--stackp;
}
```

Машина генерируется во время синтаксического разбора посредством обращений к функции code, просто помещающей команду в следующую свободную ячейку массива prog. Она возвращает местоположение ячейки (которое не обрабатывается в hoc4).

```
Inst *code(f)   /* установить одну машинную команду или операнд */
    Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("program too big", (char *) 0);
    *progp++ = f;
    return oprogp;
}
```

Работает машина просто; настолько просто, насколько мала функция, «запускающая» машину, когда она создана:

```
execute(p)  /* запустить машину */
    Inst *p;
{
    for (pc = p; *pc != STOP; )
        (*(*pc++))();
}
```

Каждый цикл выполняет функцию, на которую указывает машинная команда, на которую указывает счетчик команд pc, и увеличивает pc, чтобы тот был готов к следующей команде. Машинная команда с кодом STOP завершает цикл. Некоторые машинные команды, такие как constpush и varpush, также увеличивают pc, чтобы пропустить аргументы, следующие за командой

```
constpush() /* втолкнуть константу в стек */
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()   /* втолкнуть переменную в стек */
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push(d);
}
```

Оставшаяся часть машины очень проста. Например, все арифметические операции по существу являются одинаковыми и были созданы редактированием одного прототипа. Представим функцию add:

```
add()       /* сложить два верхних элемента стека */
{
    Datum d1, d2;
```

```
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}
```

Остальные функции настолько же просты.

```
eval()      /* оценить переменную в стеке */
{
    Datum d;
    d = pop();
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}
assign()    /* присвоить верхнее значение следующему */
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}
print()     /* вытолкнуть верхнее значение из стека и напечатать его */
{
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}
bltin()     /* оценить встроенную функцию на вершине стека */
{
    Datum d;
    d = pop();
    d.val = (*(double (*)())(*pc++))(d.val);
    push(d);
}
```

Наибольшая сложность здесь заключается в приведении типа к `bltin`, которое указывает, что `*pc` следует привести к типу «указатель на функцию, возвращающую `double`», и что функция выполняется, с `d.val` в качестве аргумента.

Если все работает правильно, диагностика в `eval` и `assign` не нужна; мы оставили ее на случай, если ошибка в какой-либо программе вызывает сбой стека. Затраты времени и пространства незначительны по сравнению с полученным преимуществом – возможностью обнаружить

ошибку, возникшую в результате неаккуратной модификации. (Авторам не раз случалось бывать в таких ситуациях.)

Благодаря способности Си манипулировать указателями на функции код получается компактным и производительным. В качестве альтернативы можно предложить сделать операторы константами и объединить семантические функции в большой оператор `switch` в `execute`, отнеситесь к этому как к несложному упражнению.

Третий экскурс в make

Исходный текст `hoc` продолжает расти, поэтому возможность механически отслеживать, что изменилось и что от чего зависит, становится все более и более ценной. Программа `make` замечательна тем, что она автоматизирует работу, которую в других обстоятельствах пришлось бы выполнять вручную (что наверняка привело бы к возникновению ошибок) или же с помощью специального командного файла.

В `makefile` было внесено два усовершенствования. Первое основано на следующем наблюдении: несмотря на то что несколько файлов зависят от констант, определенных `yacc` в `y.tab.h`, нет необходимости рекомпилировать их до тех пор, пока константы не изменятся, так как изменения кода (написанного на Си) в `hoc.y` не влияют ни на что другое. В новой версии `makefile` файлы `.o` зависят от нового файла `x.tab.h`, который обновляется только при изменении *содержимого* `y.tab.h`. Второе улучшение заключается в том, что правило для `pr` (печать исходного файла) теперь зависит от исходных файлов, так что выводятся только измененные файлы.

Первое новшество значительно ускоряет сборку больших программ, если их грамматика статична, а семантика – нет (обычная ситуация). Второе же изменение экономит горы бумаги.

Вот новый вариант `makefile` для `hoc4`:

```
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc4:   $(OBJS)
        cc $(OBJS) -lm -o hoc4

hoc.o code.o init.o symbol.o:   hoc.h

code.o init.o symbol.o: x.tab.h

x.tab.h: y.tab.h
    -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr: hoc.y hoc.h code.c init.c math.c symbol.c
    @pr $?
    @touch pr

clean:
    rm -f $(OBJS) [xy].tab.[ch]
```

Знак «-» перед cmp указывает, что make должна выполняться, даже если cmp закончилась неудачно; благодаря этому процесс работает, даже если x.tab.h не существует. (Команда cmp с параметром -s не порождает вывода, а устанавливает код завершения.) Символ $? раскрывается в список элементов правила, которые устарели. К сожалению, соглашение об обозначениях, принятое в make, мало похоже на подобное соглашение в оболочке.

Проиллюстрируем описанное на примере; будем считать все файлы новыми. Тогда

```
$ touch hoc.y              Изменить дату hoc.y
$ make
yacc -d hoc.y

conflicts: 1 shift/reduce
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cc hoc.o code.o init.o math.o symbol.o -lm -o hoc4
$ make -n pr               Вывести изменившиеся файлы
pr hoc.y
touch pr
$
```

Обратите внимание на то, что рекомпилирован был только hoc.y, т. к. файл y.tab.h не изменился.

Упражнение 8.10. Сделайте размеры stack и prog динамическими, чтобы hoc4 никогда не выходил за границы отведенной ему области памяти (если при обращении к malloc память может быть получена). □

Упражнение 8.11. Измените hoc4 так, чтобы вместо вызова функции в execute использовался switch для типа операции. Сравните количество строк исходного текста и скорость исполнения двух версий. Предположите, какую из них было бы легче расширять и поддерживать? □

8.5. Этап 5: Управляющая логика и операторы отношения

Эта версия, hoc5, использует преимущества нового интерпретатора. В ней появляются операторы if-else и while, подобные операторам Си, есть возможность группировки операторов при помощи фигурных скобок, предусмотрен оператор print. В hoc5 включен полный набор операторов отношения (>, >= и т. д.), а также операторы И и ИЛИ (&& и ||). (Последние два оператора не гарантируют оценку выражений слева направо, имеющуюся в активе Си; они оценивают оба условия, даже если в этом нет необходимости.)

В грамматику добавлены лексемы, нетерминальные символы и порождающие правила для `if`, `while`, фигурных скобок и операторов отношения. Описание грамматики несколько увеличивается в размере, но не становится намного сложнее (может быть, за исключением того, что касается `if` и `while`):

```
$ cat hoc.y
%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol  *sym;   /* указатель на таблицу символов */
    Inst    *inst;  /* машинная команда */
}
%token  <sym>   NUMBER PRINT VAR BLTIN UNDEF WHILE IF ELSE
%type   <inst>  stmt asgn expr stmtlist cond while if end
%right  '='
%left   OR
%left   AND
%left   GT GE LT LE EQ NE
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS NOT
%right  '^'
%%
list:     /* ничего */
    | list '\n'
    | list asgn '\n'  { code2(pop, STOP); return 1; }
    | list stmt '\n'  { code(STOP); return 1; }
    | list expr '\n'  { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }
    ;
asgn:     VAR '=' expr  { $$=$3; code3(varpush,(Inst)$1,assign); }
    ;
stmt:     expr      { code(pop); }
    | PRINT expr    { code(prexpr); $$ = $2; }
    | while cond stmt end {
        ($1)[1] = (Inst)$3; /* тело цикла */
        ($1)[2] = (Inst)$4; }   /* конец, если условие cond ложно */
    | if cond stmt end {    /* if без части else */
        ($1)[1] = (Inst)$3; /* часть then */
        ($1)[3] = (Inst)$4; }   /* конец, если условие cond ложно */
    | if cond stmt end ELSE stmt end {  /* if с частью else */
        ($1)[1] = (Inst)$3; /* часть then */
        ($1)[2] = (Inst)$6; /* часть else */
        ($1)[3] = (Inst)$7; }   /* конец, если условие cond ложно */
    | '{' stmtlist '}'  { $$ = $2; }
    ;
cond:     '(' expr ')'  { code(STOP); $$ = $2; }
    ;
```

```
while:    WHILE { $$ = code3(whilecode, STOP, STOP); }
    ;
if:   IF    { $$=code(ifcode); code3(STOP, STOP, STOP); }
    ;
end:      /* ничего */     { code(STOP); $$ = progp; }
    ;
stmtlist: /* ничего */     { $$ = progp; }
    | stmtlist '\n'
    | stmtlist stmt
    ;
expr:     NUMBER    { $$ = code2(constpush, (Inst)$1); }
    | VAR       { $$ = code3(varpush, (Inst)$1, eval); }
    | asgn
    | BLTIN '(' expr ')'
        { $$ = $3; code2(bltin,(Inst)$1->u.ptr); }
    | '(' expr ')'  { $$ = $2; }
    | expr '+' expr { code(add); }
    | expr '-' expr { code(sub); }
    | expr '*' expr { code(mul); }
    | expr '/' expr { code(div); }
    | expr '^' expr { code (power); }
    | '-' expr  %prec UNARYMINUS  { $$ = $2; code(negate); }
    | expr GT expr  { code(gt); }
    | expr GE expr  { code(ge); }
    | expr LT expr  { code(lt); }
    | expr LE expr  { code(le); }
    | expr EQ expr  { code(eq); }
    | expr NE expr  { code(ne); }
    | expr AND expr { code(and); }
    | expr OR expr  { code(or); }
    | NOT expr  { $$ = $2; code(not); }
    ;
%%
```

В этой грамматике пять конфликтов «сдвиг/свертка», все они подобны упоминавшемуся в разделе, посвященном `hoc3`.

Обратите внимание на то, что команды `STOP` теперь порождаются в нескольких местах, чтобы завершить последовательность; как и раньше, `progp` – это местоположение следующей машинной команды, которая будет сгенерирована. При выполнении эти команды `STOP` завершают цикл в `execute`. Порождающее правило для `end` на самом деле является подпрограммой, вызываемой из различных мест, генерирующей `STOP` и возвращающей местоположение машинной команды, которая следует за ним.

Код, порожденный для `while` и `if`, требует отдельного изучения. Когда встречается ключевое слово `while`, генерируется операция `whilecode`, и его позиция в машине возвращается как значение порождающего правила

```
while:  WHILE
```

Однако в то же время резервируются две последующие позиции в стеке машины, они будут заполнены позже. Затем генерируется код, представляющий собой выражение, которое составляет условную часть `while`. Значение, возвращаемое `cond`, – это начало кода для условия.

После того как распознан весь оператор `while`, две дополнительные позиции, которые были зарезервированы после команды `whilecode`, заполняются адресами тела цикла и оператора, следующего за циклом. (Затем будет генерироваться код для этого оператора.)

```
| while cond stmt end {
        ($1)[1] = (Inst)$3; /* тело цикла */
        ($1)[2] = (Inst)$4;  } /* конец, если условие cond ложно */
```

`$1` – это то место в машине, где хранится `whilecode`; следовательно, `($1)[1]` и `($1)[2]` – это две следующие позиции.

Прояснить все помогает рис. 8.3.

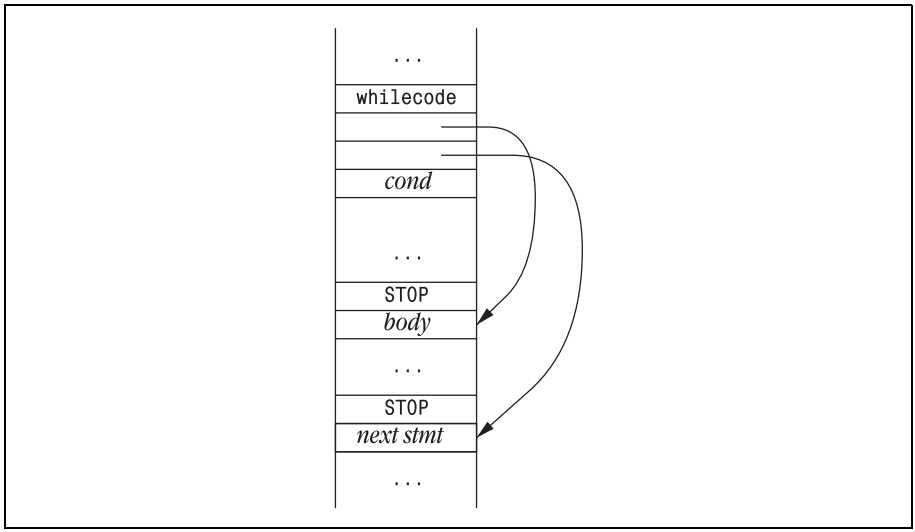

Рис. 8.3. *Порождение кода для while*

Что касается `if`, все происходит аналогично, за исключением того, что резервируются три ячейки, для частей `then` и `else` и для оператора, который следует за `if`. Вскоре мы еще поговорим о том, как все это работает.

В данной версии лексический анализатор становится более длинным, в основном для того, чтобы «вылавливать» дополнительные операторы:

```
yylex()         /* hoc5 */
...
switch (c) {
    case '>':   return follow('=', GE, GT);
    case '<':   return follow('=', LE, LT);
```

```
    case '=':   return follow('=', EQ, '=');
    case '!':   return follow('=', NE, NOT);
    case '|':   return follow('|', OR, '|');
    case '&':   return follow('&', AND, '&');
    case '\n':  lineno++; return '\n';
    default:    return c;
    }
}
```

Функция `follow` смотрит на один символ вперед и помещает его обратно в поток ввода при помощи `ungetc`, если это не то, что ожидалось.

```
follow(expect, ifyes, ifno)  /* смотрит на следующий символ в поиске >=,
                                и т.п. */
{
    int c = getchar();

    if (c == expect)
        return ifyes;
    ungetc(c, stdin);
    return ifno;
}
```

В новом `hoc.h` содержится больше объявлений функций, например для всех отношений, но в остальном он похож на `hoc.h` из `hoc4`. Вот его последние несколько строк:

```
$ cat hoc.h
...
typedef int (*Inst)();  /* машинная команда */
#define STOP    (Inst) 0

extern  Inst prog[], *progp, *code();
extern  eval(), add(), sub(), mul(), div(), negate(), power();
extern  assign(), bltin(), varpush(), constpush(), print();
extern  prexpr();
extern  gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern  ifcode(), whilecode();
$
```

Большая часть `code.c` также не отличается от версии `hoc4`, хотя в него и включено множество очевидных новых функций для обработки операторов отношения. Функция `le` (less than or equal to – меньше или равно) представляет собой типичный пример:

```
le()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}
```

А вот две функции, whilecode и ifcode, нельзя назвать очевидными. Для того чтобы понять их, необходимо осознать, что execute продвигается вдоль последовательности команд, пока не встретит STOP, а затем завершается возвратом. Генерирование кода по ходу синтаксического разбора организовано так, чтобы каждая последовательность машинных команд, которая должна быть обработана одним вызовом execute, завершалась STOP. Тело while и условие, части then и else оператора if, все они обрабатываются рекурсивными вызовами execute, которые возвращаются на родительский уровень после того, как закончат выполнение своей задачи. Контроль над этими рекурсивными задачами осуществляется функциями whilecode и ifcode, которые соответствуют операторам while и if.

```
whilecode()
{
    Datum d;
    Inst *savepc = pc;  /* тело цикла */

    execute(savepc+2);  /* условие */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc)));  /* тело */
        execute(savepc+2);
        d = pop();
    }
    pc = *((Inst **)(savepc+1));  /* следующий оператор */
}
```

Ранее упоминалось, что за операцией whilecode следует указатель на тело цикла, указатель на следующий оператор и затем – начало условной части. Когда вызывается whilecode, счетчик команд pc уже увеличен и указывает на указатель тела цикла. Тогда pc+1 указывает на следующий оператор, а pc+2 – на условие.

Функция ifcode очень похожа на whilecode; в этом случае pc первоначально указывает на часть then, pc+1 – на часть else, pc+2 – на следующий оператор, а pc+3 – на условие.

```
ifcode()
{
    Datum d;
    Inst *savepc = pc;  /* часть then */

    execute(savepc+3);  /* условие */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* часть else? */
        execute(*((Inst **)(savepc+1)));
    pc = *((Inst **)(savepc+2));     /* следующий оператор */
}
```

Код инициализации в `init.c` также несколько увеличился за счет таблицы ключевых слов, эти слова хранятся в таблице символов вместе со всем остальным ее содержимым:

```
$ cat init.c
...
static struct {     /* Ключевые слова */
    char    *name;
    int kval;
} keywords[] = {
    "if",       IF,
    "else",     ELSE,
    "while",    WHILE,
    "print",    PRINT,
    0,      0,
};
...
```

В `init` добавлен еще один цикл для размещения ключевых слов.

```
for (i = 0; keywords[i].name; i++)
        install(keywords[i].name, keywords[i].kval, 0.0);
```

Управление таблицей символов не изменяется; в `code.c` есть функция `prexpr`, которая вызывается, когда выполняется оператор вида `print` *expr*.

```
prexpr()    /* вывести числовое значение */
{
    Datum d;
    d = pop();
    printf("%.8g\n", d.val);
}
```

Это не функция `print`, которая автоматически вызывается для печати окончательного результата вычислений; та выталкивает данные из стека и добавляет в вывод знак табуляции.

Теперь `hoc5` – это очень даже полезный калькулятор, хотя для серьезного программирования требуется больше возможностей. В упражнениях предложено реализовать некоторые из них.

Упражнение 8.12. Измените `hoc5` так, чтобы выводить генерируемую им машину в пригодном для чтения виде (для отладки). □

Упражнение 8.13. Добавьте операторы присваивания из Си, такие как `+=`, `*=` и т. д., и операторы инкремента и декремента `++` и `--`. Измените `&&` и `||` так, чтобы они поддерживали оценивание слева направо и своевременное завершение (без оценки второго операнда, если она не важна), как в Си. □

Упражнение 8.14. Добавьте в `hoc5` оператор `for`, подобный оператору Си. Добавьте `break` и `continue`. □

Упражнение 8.15. Как бы вы изменили грамматику или лексический анализатор (или оба) hoc5 для того, чтобы он меньше заботился о расположении символов новой строки? Как добавить точку с запятой, чтобы она стала синонимом символа новой строки? Как ввести соглашение об обозначениях для комментария? Какой следует использовать синтаксис? □

Упражнение 8.16. Добавьте в hoc5 обработку прерываний так, чтобы вышедший из-под контроля расчет можно было остановить, не потеряв при этом уже подсчитанные значения переменных. □

Упражнение 8.17. Неудобно создавать программу в файле, запускать ее, а затем редактировать файл для того, чтобы внести тривиальное изменение. Измените hoc5 так, чтобы в нем была команда редактирования, которая перемещала бы пользователя в редактор, в котором уже была бы прочитана копия его hoc-программы. Подсказка: подумайте о машинной команде text. □

8.6. Этап 6: Функции и процедуры; ввод–вывод

Заключительным этапом эволюции hoc (по крайней мере в этой книге) является значительное расширение его функциональности – введение функций и процедур. Добавлена также возможность печати символьных строк и чисел и чтения значений с устройства стандартного ввода. Кроме того, hoc6 принимает имена файлов в качестве аргументов, в том числе имя «-» для стандартного ввода. Вместе эти изменения составляют 235 добавочных строк кода, которых в результате получается 810, но зато hoc из калькулятора превращается в язык программирования. Не будем приводить все добавленные строки в этой главе; в приложении 3 представлен весь листинг целиком, там можно увидеть, как все части согласуются друг с другом.

Для грамматики вызовы функций представляют собой выражения, а вызовы процедур – операторы. Об этом будет рассказано в приложении 2, где приведено еще несколько примеров. Сейчас рассмотрим пример определения и применения процедуры, которая выводит все числа Фибоначчи, значения которых меньше ее аргумента:

```
$ cat fib
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        print b
        c = b
        b = a+b
        a = c
```

```
    }
    print "\n"
}
$ hoc6 fib -
fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
...
```

Этот пример иллюстрирует использование файлов: имя «-» принадлежит стандартному вводу.

А вот функция, подсчитывающая факториал:

```
$ cat fac
func fac() {
    if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
$ hoc6 fac -
fac(0)
        1
fac(7)
        5040
fac(10)
        3628800
...
```

Внутри функции и процедуры (как и в оболочке) обращение к аргументам выглядит как `$1`, но разрешено и присваивать им значения. Функции и процедуры рекурсивны, но локальными переменными являются только аргументы; все остальные переменные – глобальные, то есть они доступны в любой точке программы.

Функции и процедуры в `hoc` различаются, потому что это обеспечивает наличие дополнительных проверок, которые облегчают реализацию стека. Очень легко забыть `return` или, наоборот, добавить лишнее выражение и испортить стек.

Чтобы превратить `hoc5` в `hoc6`, в грамматику внесено значительное количество изменений, но они локализованы. Необходимы новые лексемы и нетерминальные символы, а в объявлении `%union` появляется новый член, который обрабатывает счетчик аргументов:

```
$ cat hoc.y
...
%union {
    Symbol  *sym;   /* указатель на таблицу символов */
    Inst    *inst;  /* машинная команда */
    int narg;       /* количество аргументов */
}
%token  <sym>   NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token  <sym>   FUNCTION PROCEDURE RETURN FUNC PROC READ
%token  <narg>  ARG
```

```
%type   <inst>  expr stmt asgn prlist stmtlist
%type   <inst>  cond while if begin end
%type   <sym>   procname
%type   <narg>  arglist
...
list:     /* ничего */
    | list '\n'
    | list defn '\n'
    | list asgn '\n'  { code2(pop, STOP); return 1; }
    | list stmt '\n'  { code(STOP); return 1; }
    | list expr '\n'  { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }
    ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
    | ARG '=' expr
        { defnonly("$"); code2(argassign,(Inst)$1); $$=$3;}
    ;
stmt:     expr  { code(pop); }
    | RETURN { defnonly("return"); code(procret); }
    | RETURN expr
            { defnonly("return"); $$=$2; code(funcret); }
    | PROCEDURE begin '(' arglist ')'
        { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
    | PRINT prlist  { $$ = $2; }
...
expr:     NUMBER { $$ = code2(constpush, (Inst)$1); }
    | VAR    { $$ = code3(varpush, (Inst)$1, eval); }
    | ARG    { defnonly("$"); $$ = code2(arg, (Inst)$1); }
    | asgn
    | FUNCTION begin '(' arglist ')'
        { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
    | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
...
begin:    /* ничего */     { $$ = progp; }
    ;
prlist:   expr          { code(prexpr); }
    | STRING        { $$ = code2(prstr, (Inst)$1); }
    | prlist ',' expr   { code(prexpr); }
    | prlist ',' STRING { code2(prstr, (Inst)$3); }
    ;
defn:     FUNC procname { $2->type=FUNCTION; indef=1; }
        '(' ')' stmt { code(procret); define($2); indef=0; }
    | PROC procname { $2->type=PROCEDURE; indef=1; }
        '(' ')' stmt { code(procret); define($2); indef=0; }
    ;
procname: VAR
    | FUNCTION
    | PROCEDURE
    ;
arglist:  /* ничего */     { $$ = 0; }
    | expr          { $$ = 1; }
```

```
    | arglist ',' expr  { $$ = $1 + 1; }
    ;
%%

...
```

Порождающие правила для arglist подсчитывают количество аргументов. На первый взгляд может показаться, что необходимо как-то собирать аргументы, но это не так, ведь каждое выражение expr в списке аргументов оставляет свое значение в стеке именно тогда, когда нужно. Поэтому единственное, что требуется – это знать, сколько аргументов в стеке.

Правила для defn вводят новое свойство yacc, вложенное действие. Можно вставлять действие в середину правила – так, чтобы оно выполнялось во время распознавания этого правила. Данное свойство использовано нами для того, чтобы фиксировать факт нахождения в определении процедуры или функции. (В качестве альтернативы можно создать новый символ, аналогичный begin, который опознавался бы в нужное время.) Функция defnonly выводит предупреждение, если конструкция встречается вне определения функции или процедуры, где ее не должно быть. Часто возникает выбор: выявлять ли ошибки синтаксически или семантически; мы уже сталкивались с такой ситуацией при обработке неопределенных переменных. Функция defnonly служит хорошим примером того, что семантическая проверка может быть легче синтаксической.

```
defnonly(s) /* предупреждает, если есть недопустимое определение*/
    char *s;
{
    if (!indef)
        execerror(s, "used outside definition");
}
```

Переменная indef объявляется в hoc.y и определяется в описании действия для defn.

Лексический анализатор дополнен проверкой строк в кавычках и аргументов – символом $, за которым следует число. Управляющие символы, такие как \n, интерпретируются в строках функцией backslash.

```
yylex()         /* hoc6 */
...
if (c == '$') { /* аргумент? */
        int n = 0;
        while (isdigit(c=getc(fin)))
            n = 10 * n + c - '0';
        ungetc(c, fin);
        if (n == 0)
            execerror("strange $...", (char *)0);
        yylval.narg = n;
```

```
            return ARG;
        }
        if (c == '"') { /* строка в кавычках */
            char sbuf[100], *p, *emalloc();
            for (p = sbuf; (c=getc(fin)) != '"'; p++) {
                if (c == '\n' || c == EOF)
                    execerror("missing quote", "");
                if (p >= sbuf + sizeof(sbuf) - 1) {
                    *p = '\0';
                    execerror("string too long", sbuf);
                }
                *p = backslash(c);
            }
...
backslash(c)    /* интерпретировать следующий символ с \ */
    int c;
{
    char *index();  /* 'strchr()' в некоторых системах */
    static char transtab[] = "b\bf\fn\nr\rt\t";
    if (c != '\\')
        return c;
    c = getc(fin);
    if (islower(c) && index(transtab, c))
        return index(transtab, c)[1];
    return c;
}
```

Лексический анализатор, написан ли он на Си или же при помощи такого генератора программ, как `lex`, является примером *конечного автомата*. Наша *специальная* версия на Си стала очень сложной; для других задач `lex`, вероятно, подошел бы лучше, как из-за размера исходного кода, так и из-за простоты изменения.

Большая часть других изменений относится к `code.c`, несколько имен функций введено в `hoc.h`. Машина осталась прежней, только добавился второй стек для отслеживания вызовов вложенных функций и процедур. (Лучше ввести второй стек, чем забивать уже существующий.) Вот начало `code.c`:

```
$ cat code.c
#define NPROG   2000
Inst    prog[NPROG];    /* машина */
Inst    *progp;         /* следующая свободная ячейка для генерирования кода
*/
Inst    *pc;        /* счетчик команд во время исполнения */
Inst    *progbase = prog; /* запуск текущей подпрограммы */
int returning;      /* 1, если встречен оператор return*/

typedef struct Frame {  /* стековый фрейм вызова функции/процедуры */
    Symbol  *sp;    /* элемент таблицы символов */
    Inst    *retpc; /* место возобновления после возврата */
    Datum   *argn;  /* n-й аргумент в стеке */
```

```
    int nargs;      /* количество аргументов */
} Frame;
#define NFRAME  100
Frame   frame[NFRAME];
Frame   *fp;        /* указатель фрейма */

initcode() {
    progp = progbase;
    stackp = stack;
    fp = frame;
    returning = 0;
}
...
$
```

Таблица символов теперь хранит указатели на процедуры и функции, а также на строки для печати, поэтому сделано расширение типа `union` в `hoc.h`:

```
$ cat hoc.h
typedef struct Symbol {     /* элемент таблицы символов */
    char    *name;
    short   type;
    union {
        double  val;        /* VAR */
        double  (*ptr)();   /* BLTIN */
        int (**defn)();      /* FUNCTION, PROCEDURE */
        char    *str;       /* STRING */
    } u;
    struct Symbol   *next;  /* связать с другим */
} Symbol;

$
```

Во время компиляции функция вводится в таблицу символов при помощи функции `define`, которая сохраняет адрес начала в таблице и обновляет следующую свободную позицию за сгенерированным кодом, если компиляция прошла успешно.

```
define(sp)  /* вставить функцию/процедуру в таблицу символов */
    Symbol *sp;
{
    sp->u.defn = progbase;    /* начало кода */
    progbase = progp;   /* следующий код начинается здесь */
}
```

Когда функция или процедура вызывается во время исполнения, все аргументы уже сосчитаны и помещены в стек (первый аргумент лежит глубже всего). За кодом машинной команды `call` следует указатель на таблицу символов и количество аргументов. Формируется элемент структуры `Frame`, содержащий всю полезную информацию о подпрограмме – ее вхождение в таблицу символов, место, куда она возвраща-

ется после вызова, где в стеке находятся аргументы и с каким количеством аргументов она была вызвана. Стековый фрейм создается функцией `call`, которая затем выполняет код подпрограммы.

```
call()      /* вызвать функцию */
{
    Symbol *sp = (Symbol *)pc[0]; /* элемент таблицы      */
                                  /* символов для функции */
if (fp++ >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackp - 1;        /* последний аргумент */
    execute(sp->u.defn);
    returning = 0;
}
```

Эта структура проиллюстрирована на рис. 8.4.

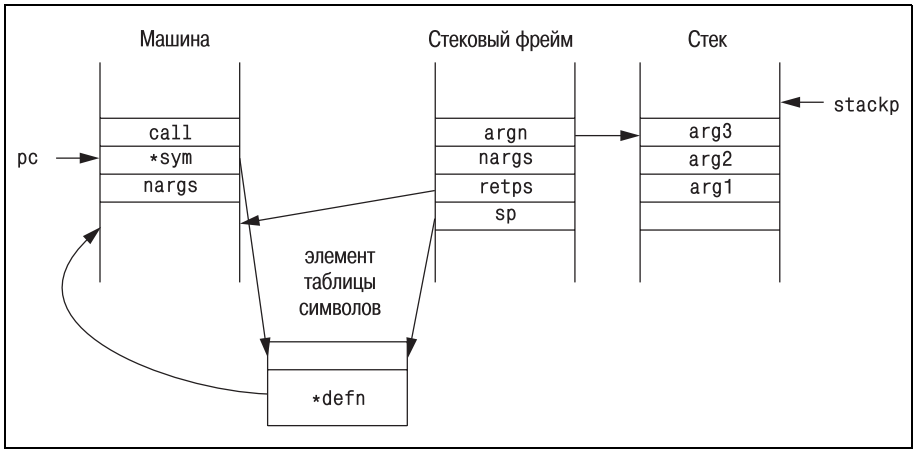

***Рис. 8.4.** Структуры данных для вызова процедуры*

Со временем вызванная подпрограмма возвратится, при выполнении `procret` или `funcret`:

```
funcret()   /* возврат из функции */
{
    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pop(); /* сохранить возвращенное функцией значение*/
    ret();
    push(d);
}
procret()   /* возврат из процедуры */
```

```
{
    if (fp->sp->type == FUNCTION)
        execerror(fp->sp->name,
            "(func) returns no value");
    ret();
}
```

Функция `ret` **выталкивает аргументы из стека, восстанавливает указатель фрейма** `fp` **и устанавливает счетчик команд.**

```
ret()       /* общий возврат из процедуры или функции */
{
    int i;
    for (i = 0; i < fp->nargs; i++)
        pop();    /* вытолкнуть аргументы */
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}
```

В нескольких подпрограммах интерпретатора необходимо произвести незначительные изменения для обработки ситуации, когда `return` **встречается во вложенном операторе. Это решается (не очень красиво, но вполне адекватно) при помощи флага, названного** `returning`**, принимающего значение «истина», если был встречен оператор** `return`**. Функции** `ifcode`**,** `whilecode` **и** `execute` **завершаются раньше, если установлен** `returning`**;** `call` **переустанавливает его в** `0`**.**

```
ifcode()
{
    Datum d;
    Inst *savepc = pc;  /* часть then */

    execute(savepc+3);  /* условие */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* часть else? */
        execute(*((Inst **)(savepc+1)));
    if (!returning)
        pc = *((Inst **)(savepc+2)); /* следующий оператор */
}
whilecode()
{
    Datum d;
    Inst *savepc = pc;

    execute(savepc+2);  /* условие */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc)));  /* body */
        if (returning)
```

```
            break;
        execute(savepc+2);  /* условие */
        d = pop();
    }
    if (!returning)
        pc = *((Inst **)(savepc+1)); /* следующий оператор */
}
execute(p)
    Inst *p;
{
    for (pc = p; *pc != STOP && !returning; )
        (*((++pc)[-1]))();
}
```

Аргументы для использования или присваивания выбираются из памяти при помощи функции `getarg`**, которая осуществляет арифметические операции над стеком:**

```
double *getarg()    /* возвращает указатель на аргумент */
{
    int nargs = (int) *pc++;
    if (nargs > fp->nargs)
        execerror(fp->sp->name, "not enough arguments");
    return &fp->argn[nargs - fp->nargs].val;
}
arg()   /* протолкнуть аргумент в стек */
{
    Datum d;
    d.val = *getarg();
    push(d);
}

argassign()     /* сохранить вершину стека в аргументе */
{
    Datum d;
    d = pop();
    push(d);    /* оставить значение в стеке */
    *getarg() = d.val;
}
```

Печать строк и чисел осуществляется функциями `prstr` **и** `prexpr`**.**

```
prstr()     /* вывести строковое значение */
{
    printf("%s", (char *) *pc++);
}
prexpr()    /* вывести числовое значение */
{
    Datum d;
    d = pop();
    printf("%.8g ", d.val);
}
```

Переменные считывает функция varread. Она возвращает 0 при достижении конца файла; в остальных случаях она возвращает 1 и устанавливает указанные переменные.

```
varread()   /* читать в переменную */
{
    Datum d;
    extern FILE *fin;
    Symbol *var = (Symbol *) *pc++;
  Again:
    switch (fscanf(fin, "%lf", &var->u.val)) {
    case EOF:
        if (moreinput())
            goto Again;
        d.val = var->u.val = 0.0;
        break;
    case 0:
        execerror("non-number read into", var->name);
        break;
    default:
        d.val = 1.0;
        break;
    }
    var->type = VAR;
    push(d);
}
```

Когда встретится конец текущего входного файла, функция varread вызывает moreinput, которая открывает следующий файл (если он существует). Функция moreinput учитывает больше информации об обработке входных файлов, чем необходимо в данном случае (см. приложение 3).

Мы подходим к завершению разработки hoc. Для сравнения приведем количество непустых строк в каждой версии:

```
hoc1     59
hoc2     94
hoc3    248     (версия с lex 229)
hoc4    396
hoc5    574
hoc6    809
```

Естественно, подсчеты были произведены программами:

```
$ sed '/^$/d' `pick *.[chyl]` | wc -l
```

Язык отнюдь не закончен, по крайней мере, в том смысле, что легко можно придумать всевозможные полезные расширения, но в этой книге остановимся на сделанном. Внести некоторые изменения, которые, вероятно, могут оказаться весьма ценными, предложено в упражнениях.

Упражнение 8.18. Измените hoc6 так, чтобы разрешить использовать формальные параметры, имеющие имена, в подпрограммах как альтернативу $1 и т. д. □

Упражнение 8.19. Сейчас все переменные, кроме параметров, являются глобальными. Большая часть механизма добавления локальных переменных, сохраняемых в стеке, уже готова. Можно предложить создать объявление auto, которое готовит на стеке место для перечисленных переменных; переменные, которые не объявлены таким способом, будут считаться глобальными. Надо будет расширить и таблицу символов, сделав так, чтобы сначала производился поиск локальных переменных, а уже затем – глобальных. Как это будет взаимодействовать с аргументами, имеющими имена? □

Упражнение 8.20. Как бы вы добавили в hoc массивы? Как передавать их функциям и процедурам? Как возвращать их? □

Упражнение 8.21. Обобщите хранение строк так, чтобы переменные могли хранить строки вместо чисел. Какие потребуются операторы? Самым сложным в этой задаче является управление памятью: надо убедиться, что строки хранятся так, что когда они не нужны, они освобождаются, чтобы не возникало утечек памяти. В качестве промежуточного шага добавьте улучшенные возможности форматирования выходных данных, например доступ к какой-либо форме функции printf из Си. □

8.7. Оценка производительности

Для того чтобы получить представление о том, насколько хорошо работает hoc, было проведено его сравнение с некоторыми другими программами UNIX, реализующими калькуляторы. Таблицу, представленную в данном разделе, следует воспринимать с некоторой долей скептицизма, но, тем не менее, она показывает, что наша реализация вполне разумна. Все времена приведены в секундах пользовательского времени на PDP-11/70. Решались две задачи. Первым был расчет функции Аккермана ack(3,3). Это хорошая проверка для механизма вызова функций, т. к. для данного расчета необходимо 2432 вызова различной глубины вложенности.

```
func ack() {
        if ($1 == 0) return $2+1
        if ($2 == 0) return ack($1-1, 1)
        return ack($1-1, ack($1, $2-1))
}
ack(3,3)
```

Вторым испытанием был подсчет чисел Фибоначчи, не превышающих 1000, который должен был быть выполнен сто раз. В этом примере много арифметических операций и редкие вызовы функции.

```
proc fib() {
        a = 0
        b = 1
        while (b < $1) {
                c = b
                b = a+b
                a = c
        }
}
i = 1
while (i < 100) {
        fib(1000)
        i = i + 1
}
```

Сравнивались четыре языка: `hoc`, `bc(1)`, `bas` (древний диалект Бейсика, который работает только на PDP-11) и Си (с использованием типа `double` для всех переменных).

Таблица 8.1. Секунды затраченного времени (PDP-11/70)

программа	`ack(3,3)`	`100×fib(1000)`
`hoc`	**5,5**	**5,0**
`bas`	**1,3**	**0,7**
`bc`	**39,7**	**14,9**
`c`	**<0,1**	**<0,1**

Числа в табл. 8.1 представляют собой суммы пользовательского и системного времени работы процессора, измеренные `time`. Можно также оснастить программу на Си инструментальными средствами, позволяющими определить, сколько времени использует каждая из функций. Программу надо перекомпилировать с параметром `-p`, включающим режим профилирования. Если изменить `makefile`

```
hoc6:   $(OBJS)
        cc $(CFLAGS) $(OBJS) -lm -o hoc6
```

(команда `cc` использует переменную `CFLAGS`), а затем сказать

```
$ make clean; make CFLAGS=-p
```

то в получившейся программе будет содержаться профилирующий код. Когда программа выполняется, она оставляет после себя файл с данными `mon.out`, который интерпретируется программой `prof`.

Чтобы вкратце пояснить эти понятия, представим испытание `hoc6` программой Фибоначчи, описанной выше.

```
$ hoc6 <fibtest              Запустить тест
$ prof hoc6 | sed 15q        Проанализировать
```

```
      name %time  cumsecs  #call  ms/call
      _pop  15.6     0.85 32182    0.03
     _push  14.3     1.63 32182    0.02
    mcount  11.3     2.25
       csv  10.1     2.80
      cret   8.8     3.28
   _assign   8.2     3.73  5050    0.09
     _eval   8.2     4.18  8218    0.05
  _execute   6.0     4.51  3567    0.09
  _varpush   5.9     4.83 13268    0.02
       _lt   2.7     4.98  1783    0.08
  _constpu   2.0     5.09   497    0.22
      _add   1.7     5.18  1683    0.05
   _getarg   1.5     5.26  1683    0.05
  _yyparse   0.6     5.30     3   11.11
$
```

Измерения, полученные таким способом, так же подвержены случайным колебаниям, как и измерения программы `time`, поэтому лучше рассматривать их не как абсолютные величины, а просто как индикаторы. Представленные величины показывают, как можно сделать `hoc` быстрее, *если это нужно*. Около трети всего времени исполнения тратится на размещение данных в стеке и их выборку из стека.

Затраты окажутся еще больше, если добавить время для функций связывания подпрограммы Си, `csv` и `cret` (`mcount` является частью профилирующего кода, компилируемого посредством `cc -p`.) Если заменить вызовы функций макросами, разница должна быть значительной.

Чтобы проверить это предположение, изменим `code.c`, заменив вызовы `push` и `pop` макросами управления стеком:

```
#define push(d) *stackp++ = (d)
#define popm()  *--stackp       /* функция все еще нужна */
```

(Функция `pop` все еще необходима в качестве кода машинной операции, поэтому просто заменить все `pop` нельзя.) Новая версия работает на 35% быстрее, значения времени из табл. 8.1 сокращаются с 5,5 до 3,7 секунд и с 5,0 до 3,1.

Упражнение 8.22. В макросах `push` и `popm` проверка ошибок не выполняется. Прокомментируйте разумность такой архитектуры. Можно ли совместить проверку ошибок, предоставляемую функциями, со скоростью выполнения макросов? □

8.8. Оглянемся назад

Из этой главы вы узнали много важных вещей. Во-первых, средства разработки языков вполне доброжелательно настроены к программистам. Они позволяют сконцентрироваться на интересной части работы –

проектировании языка, потому что экспериментировать в них очень просто. Использование грамматики предоставляет организационную структуру для реализации – функции связываются вместе при помощи грамматики и вызываются в нужное время по мере осуществления грамматического разбора.

Второй, более философский момент заключается в том, чтобы воспринимать имеющееся задание как разработку языка, а не «написание программы». Организация программы в качестве процессора языка способствует регулярности синтаксиса (то есть пользовательского интерфейса) и структурирует реализацию. Это также гарантирует, что новые возможности будут согласованы с уже существующими. Конечно же, понятие «язык» не ограничивается стандартными языками программирования, только в этой книге уже были представлены такие специализированные языки, как `eqn` и `pic`, а также сами `yacc`, `lex` и `make`.

В главе также содержатся полезные советы о применении различных инструментальных средств. Например, программа `make` просто бесценна. По существу, она избавляет от целого класса ошибок, появляющихся из-за того, что какую-то подпрограмму забыли перекомпилировать. Она гарантирует, что лишняя (избыточная) работа не выполняется. К тому же, она предоставляет удобный способ упаковывания группы связанных или взаимозависимых операций в один файл.

Заголовочные файлы представляют собой хорошее средство для организации объявлений данных, которые должны быть видимыми в нескольких файлах. Централизуя информацию, они исключают появление ошибок, обусловленных использованием несовместимых версий, особенно в паре с `make`. Кроме того, важно разместить данные и функции в файлах так, чтобы они не были видимыми тогда, когда этого не требуется.

Есть несколько тем, которым не было уделено особого внимания из-за недостатка места. Например, не рассказывалось о том, в какой степени применялись все *остальные* средства UNIX во время разработки семейства `hoc`. Каждая версия программы находится в отдельном каталоге, идентичные файлы связаны ссылками; `ls` и `du` многократно используются для того, чтобы следить за тем, что где находится. Ответы на многие другие вопросы дают программы. Например, где объявлена данная переменная? Обратитесь к `grep`. Что изменилось в этой версии? Поможет `diff`. Как интегрировать изменения в данную версию? Используйте `idiff`. Каков размер этого файла? Спросите у `wc`. Пора сделать резервную копию? Примените утилиту `cp`. Как копировать только те файлы, которые изменились после последнего резервного копирования? Используйте `make`. Такой общий подход типичен для повседневной разработки программ в системе UNIX: совокупность небольших инструментальных средств, использованных по отдельности или (при необходимости) объединенных вместе, помогает автоматизировать работу, которую иначе пришлось бы выполнять вручную.

История и библиография

Программа yacc была создана Стивом Джонсоном (Steve Johnson). Формально класс языков, для которых yacc может генерировать синтаксические анализаторы, называется LALR(1): синтаксический разбор происходит слева направо, при этом просмотр ввода происходит с опережением максимум на одну лексему. Понятие отдельного описания для того, чтобы разрешить проблему приоритетов и неоднозначности в грамматике, появилось только в yacc. См. «Deterministic parsing of ambiguous grammars» А. В. Ахо (A. V. Aho), С. С. Джонсона (S. C. Johnson) и Д. Д. Ульмана (J. D. Ullman) (издано CACM в августе 1975 года). Использованы некоторые передовые алгоритмы и структуры данных для создания и хранения таблиц синтаксического анализа.

Хорошее описание базовой теории, лежащей в основе yacc и других генераторов синтаксических анализаторов, приведено в книге А. В. Ахо и Д. Д. Ульмана «Principles of Compiler Design» (Принципы проектирования компиляторов), Addison-Wesley, 1977. Сам yacc описан в томе 2B справочного руководства по UNIX. В том же разделе представлен калькулятор, сопоставимый с hoc2; полезно произвести их сравнение.

Программа lex была изначально создана Майком Леском (Mike Lesk). Теоретические основы представлены в уже упомянутом труде Ахо и Ульмана, а сам язык lex документирован в справочном руководстве по UNIX.

Многие языковые процессоры, включая переносимый компилятор Си, Паскаль, ФОРТРАН 77, Ratfor, awk, bc, eqn и pic, были реализованы с использованием yacc и (в меньшей степени) lex.

Программа make была написана Сту Фелдманом (Stu Feldman). См. «MAKE – a program for maintaining computer programs» (MAKE – программа для поддержки компьютерных программ), изданную Software – Practice & Experience в апреле 1979 года.

В книге «Writing Efficient Programs» (Написание эффективных программ) Джона Бентли (Jon Bentley) (Prentice-Hall, 1982) описаны техники, позволяющие увеличить скорость выполнения программы. Особое внимание обращается на то, что сначала следует выбрать правильный алгоритм, а затем уже – в случае необходимости – усовершенствовать код.

9

Подготовка документов

Одним из первых применений системы UNIX было редактирование и форматирование документов, ведь руководство Bell Labs удалось склонить на покупку первой машины PDP-11 именно за счет обещаний создать систему подготовки документов, а совсем не операционной системы. (К счастью, они получили больше, чем ожидали.)

Первая программа форматирования текстов называлась `roff`. Она была маленькой, быстрой и удобной в работе для тех, кому надо было выводить небольшие документы на построчно печатающий принтер. Следующее средство форматирования, `nroff`, созданное Джо Оссанной (Joe Ossanna), было гораздо более претенциозным. Вместо того чтобы пытаться предоставить в распоряжение пользователей все стили документов, Оссанна сделал `nroff` программируемым, так что многие задачи форматирования решались посредством программирования на языке `nroff`.

С приобретением в 1973 году маленькой наборной машины `nroff` был расширен для обработки многочисленных шрифтов, размеров и всего множества символов, предоставляемого этой машиной. Новая программа получила название `troff` (по аналогии с «эн-рофф» произносится «ти-рофф»). По существу, `nroff` и `troff` – это одна и та же программа, для них принят один и тот же язык ввода; `nroff` игнорирует такие команды, как изменение размера, которые он не может обработать. В книге будет рассказываться о программе `troff`, но большая часть комментариев применима и к `nroff`, с учетом ограничений, касающихся устройств вывода.

Основным достоинством `troff` является гибкость базового языка и его программируемость, можно сделать так, чтобы практически любая задача форматирования была выполнена. Но эта гибкость дорого обошлась программе – с `troff` бывает очень тяжело работать. Справедливос-

ти ради отметим, что практически все программное обеспечение UNIX для подготовки документов было разработано для того, чтобы прикрыть какую-то часть «голой» программы `troff`.

В качестве одного из примеров можно привести макет страницы – общий стиль документа, на котором представлено, как выглядит название, заголовки и абзацы, где выводится номер страницы, каков ее размер и т. д. Все это не встроено в `troff`, а требует программирования. Однако чтобы не заставлять каждого пользователя указывать все детали в каждом документе, создан пакет стандартных команд форматирования. Пользователь пакета не должен говорить: «Следующая строка должна быть выровнена по центру и напечатана большими буквами полужирным шрифтом». Вместо этого он просто говорит: «Следующая строка – это заголовок», и применяется пакетное определение стиля заголовка. Пользователям не надо заботиться о размерах, шрифтах и позициях, они могут думать только о логических компонентах документа: названии, заголовках, абзацах, сносках и т. д.

К сожалению, то, что начиналось как «стандартный» пакет команд форматирования, теперь уже стандартом не является; многие пакеты получили широкое распространение, к тому же, у каждого существует по несколько локальных вариантов. Рассмотрим два универсальных пакета: `ms`, исходный «стандартный» пакет, и `mm`, более новая версия, принятая за стандарт в System V. Поговорим также о пакете `man` для печати страниц руководства (manual).

Основное внимание будет уделено `ms`, потому что он является стандартом седьмой версии, может проиллюстрировать все подобные пакеты и обладает достаточной мощью для того, чтобы выполнить поставленную задачу: пакет `ms` был использован при наборе оригинала этой книги[1]. Правда авторам пришлось немного его расширить, например, добавив команду для записи слов `вот таким шрифтом`.

Такая ситуация является типичной, макропакетов достаточно для выполнения многих задач форматирования, но иногда бывает необходимо вернуться к базовым командам `troff`. В данной книге будет описана только малая часть возможностей `troff`.

Несмотря на то что `troff` предоставляет возможность полного контроля над форматом вывода, использовать программу для такого сложного материала, как таблицы, рисунки и математические выражения, слишком тяжело. Описать такой объект так же сложно, как создать макет страницы. Решение проблемы заключается в использовании (вместо пакетов команд форматирования) специализированных языков для математических выражений, таблиц и рисунков, в которых легко описать то, что требуется. Каждый такой язык управляется от-

[1] Разумеется, авторы говорят об издании этой книги, увидевшем свет в Prentice Hall. – *Примеч. науч. ред.*

дельной командой, которая транслирует его в команды troff. Команды сообщаются посредством каналов.

Подобные процессоры предварительной обработки представляют собою хороший пример, демонстрирующий подход UNIX к реализации программ, – вместо того чтобы расширять troff, усложняя ее, вводятся отдельные команды, работающие во взаимодействии с ней. (Конечно же, при реализации были использованы средства разработки языков, описанные в главе 8.) Далее будут представлены две программы: tbl, которая форматирует таблицы, и eqn, форматирующая математические выражения.

Также в этой главе будет дан ряд полезных советов, касающихся подготовки документов и вспомогательных программ. Примеры, приведенные в главе, представляют собой части документа, описывающего язык hoc из главы 8 и страницу руководства по hoc. Документация на него представлена в приложении 2.

9.1. Макропакет ms

Основная идея макропакета заключается в том, что документ описывается в терминах его логической структуры: название, заголовки разделов, абзацы, и не надо указывать размеры букв, шрифты и расстояния между строками. Пользователь освобожден от тяжелой работы, а документ изолирован от не относящихся к делу подробностей. На самом деле, применяя различные наборы макроопределений с одинаковыми логическими именами, можно создать документы, выглядящие абсолютно по-разному. Например, документ может пройти через стадии технического отчета, материалов конференции, газетной статьи и главы книги с одними и теми же командами форматирования, использованными четырьмя разными макропакетами.

Входными данными для troff, вне зависимости от того, используется ли макропакет, является обычный текст, по которому «рассыпаны» команды форматирования. Существует два вида команд.

Команда первой разновидности состоит из точки в начале строки, за которой следуют одна или две буквы или цифры и, возможно, параметры, как в примере ниже:

```
.PP
.ft B
This is a little bold font paragraph.
```

Имена всех встроенных в troff команд заданы буквами в нижнем регистре, поэтому было принято соглашение называть команды макропакетов именами в верхнем регистре. В данном примере .PP – это команда макропакета ms для нового абзаца, а .ft B – это команда troff, вызывающая изменение шрифта на **полужирный**. (Названия шрифтов

записываются в верхнем регистре; наборы шрифтов, доступных на разных наборных машинах, могут отличаться друг от друга.)

Второй вид команд troff выглядит как символьная строка, начинающаяся с символа обратной косой черты \, которая может находиться в любом месте ввода; например \fB также вызывает изменение шрифта на полужирный. Такая форма команды присуща именно troff; об этом еще будет рассказано позже.

Перед каждым новым абзацем используйте команду .PP, при этом для форматирования большинства документов можно обойтись дюжиной команд ms. Например, в приложении 2, описывающем hoc, есть название, имена авторов, аннотация, автоматическая нумерация разделов и абзацы. В нем использовано всего 14 команд, некоторые из которых являются парными. Страница обретает форму в ms:

```
.TL
Название документа (одна или несколько строк)
.AU
Имена авторов, по одному на строке
.AB
Аннотация, заканчивающаяся .AE
.AE
.NH
Пронумерованный заголовок (автоматическая нумерация)
.PP
Абзац...
.PP
Еще абзац...
.SH
Подзаголовок (без нумерации)
.PP
...
```

Команды форматирования могут встретиться в начале строки. Формат исходных данных между командами свободный (положение символов новой строки не имеет значения), потому что troff переносит слова из строки в строку, чтобы сделать их достаточно длинными (такой процесс называется *заполнением*), и равномерно распределяет дополнительные пробелы между словами, чтобы выровнять края (*выравнивание*). Однако начинать каждое предложение с новой строки – это хорошая привычка, она облегчает дальнейшее редактирование.

Вот начало настоящей документации по hoc:

```
.TL
Hoc - An Interactive Language For Floating Point Arithmetic
.AU
Brian Kernighan
Rob Pike
.AB
```

```
.I Hoc
is a simple programmable interpreter
for floating point expressions.
It has C-style control flow,
function definition and the usual
numerical built-in functions
such as cosine and logarithm.
.AE
.NH
Expressions
.PP
.I Hoc
is an expression language,
much like C:
although there are several control-flow statements,
most statements such as assignments
are expressions whose value is disregarded.
...
```

Команда `.I` выделяет свой аргумент курсивом или включает печать наклонными буквами, если аргумент не задан.

Если используется макропакет, он указывается как аргумент `troff`:

```
$ troff -ms hoc.ms
```

Символы, следующие за `-m`, задают макропакет.[1] При форматировании с помощью `ms` лист статьи о `hoc` выглядит так:

Hoc – An Interactive Language For Floating Point Arithmetic

Brian Kernighan
Rob Pike

ABSTRACT

Hoc is a simple programmable interpreter for floating point expressions. It has C-style control flow, function definition and the usual numerical built-in functions such as cosine and logarithm.

1. Expressions

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded.

[1] Макросы `ms` находятся в файле `/usr/lib/tmac/tmac.s`, а макросы `man` – в `/usr/lib/tmac/tmac.an`.

Отображения

Обычно то, что troff заполняет и выравнивает строки, можно рассматривать как удобство, но иногда это нежелательно, например в текстах программ. Такой неформатированный материал называется отображаемым текстом. Команды ms .DS (display start – начало отображения) и .DE (display end – конец отображения) отделяют текст, который должен быть напечатан в том виде, в каком он введен, структурированно, но без перегруппировки. Приведем следующую часть учебника по hoc, в которой есть фрагмент отображаемого текста:

```
.PP
.I Hoc
is an expression language,
much like C:
although there are several control-flow statements, most statements
such as assignments are expressions whose value is disregarded.
For example, the assignment operator = assigns the value
of its right operand to its left operand, and yields
the value, so multiple assignments work.
The expression grammar is:
.DS
.I
expr:                   number
             |          variable
             |          ( expr )
             |          expr binop expr
             |          unop expr
             |          function ( arguments )
.R
.DE
Numbers are floating point.
```

который выводится как

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns the value of its right operand to its left operand, and yields the value, so multiple assignments work. The expression grammar is:

```
expr:           number
          |     variable
          |     (expr)
          |     expr binop expr
          |     unop expr
          |     function (arguments)
```

Numbers are floating point.

К отображаемому тексту не применяется заполнение и выравнивание. Следовательно, если на текущей странице недостаточно места, то отображаемый материал (и все, что следует за ним) помещается на следующую страницу. У команды .DS есть несколько параметров, в том числе L для выравнивания по левому краю, C для центрирования каждой строки в отдельности и B, центрирующий весь отображаемый текст целиком.

Элементы отображения, приведенного выше, разделены знаками табуляции. По умолчанию шаг табуляции в troff равен половине дюйма, а не восьми пробелам, как обычно. И даже если бы точки табуляции были расставлены через каждые 8 пробелов, то (так как все символы имеют разную ширину) знаки табуляции, обработанные troff, не всегда выглядели бы, как ожидалось.

Изменение шрифта

Макрос ms предоставляет три команды для изменения шрифта. Команда .R меняет шрифт на прямой, обычный шрифт, .I меняет шрифт на курсив, а .B – на полужирный. Введенные без аргументов команды определяют шрифт последующего текста:

```
This text is roman, but
.I
this text is italic,
.R
this is roman again, and
.B
this is boldface.
```

этот текст будет выглядеть так:

This text is roman, but *this text is italic*, this is roman again, and **this is boldface.**

Команды .I и .B обрабатывают необязательный аргумент, который указывает, что изменение шрифта должно применяться только к аргументу. В troff аргументы, содержащие пробелы, должны заключаться в кавычки, причем существует единственный символ, который может быть использован с этой целью – двойные кавычки ".

```
This is roman, but
.I this
is italic, and
.B "these words"
are bold.
```

выводится как

This is roman, but *this* is italic, and **these words** are bold.

Наконец, второй аргумент команды .I или .B выводится прямым шрифтом, непосредственно за первым аргументом (без пробелов). Такая возможность часто используется для выведения знаков пунктуации правильным шрифтом. Сравните последнюю скобку в

```
(parenthetical
.I "italic words)"
```

Здесь она выводится неправильно, то есть

(parenthetical *italic words)*

с последней скобкой в

```
(parenthetical
.I "italic words" )
```

Эта скобка напечатана правильно:

(parenthetical *italic words*)

Различия шрифтов распознаются nroff, но результат выглядит не очень обнадеживающе. Курсивные символы подчеркиваются, а полужирных символов вообще нет (некоторые версии nroff имитируют полужирный шрифт наложением символов).

Разнообразные команды

Сноски вводятся командой .FS, а завершаются командой .FE. Ответственность за идентифицирующие сноску метки, такие как звездочка * или крестик † ложится на вас. Эта сноска была создана таким образом:

```
сноски пометки, такие как звездочка или крестик.\(dg
.FS
\(dg Например, вот такой крестик.
.FE
Эта сноска была создана таким образом...
```

Абзацы с отступом (с цифрой или другой меткой с краю) создаются при помощи команды .IP. Пусть требуется создать:

(1) Первый маленький абзац.

(2) Второй абзац, который специально сделан более длинным, для того чтобы показать, что отступ второй строки будет таким же, как и у первой.

Тогда надо ввести:

```
.IP (1)
Первый маленький абзац.
```

† Например, вот такой крестик.

```
.IP (2)
Второй абзац, который...
```

Признаком конца команды .IP является .PP или .LP (абзац, выровненный по левому краю). Аргументом .IP может быть любая строка; при необходимости защищайте пробелы кавычками. Второй аргумент может быть использован для указания величины отступа.

Применение пары команд .KS и .KE приводит к тому, что текст не разделяется, то есть текст, заключенный между двумя этими командами, будет перенесен на новую страницу, если не поместится целиком на текущую. Если вместо .KS использована .KF, то текст перемещается за последующий текст, в начало следующей страницы, если надо сохранить его целиком на одной странице. Команда .KF использована для всех таблиц в этой книге.

Большинство из значений ms по умолчанию можно изменить, задав регистры чисел, – это переменные troff, используемые ms. Наверное, чаще всего используются регистры, которые управляют размером текста и расстоянием между строками. Обычный размер текста (который вы читаете сейчас) – это «10 пунктов», где пункт – это единица, унаследованная из полиграфии, равная приблизительно 1/72 дюйма. Расстояние между строками обычно равно 12 пунктам. Чтобы изменить эти величины, например на 9 и 11 (как на наших дисплеях), установите регистры чисел PS и VS посредством

```
.nr PS 9
.nr VS 11
```

Существуют еще такие регистры чисел, как LL – для длины строки, PI – для отступа абзаца и PD – для расстояния между абзацами. Они вступают в силу при вводе следующей команды .PP или .LP. Команды ms представлены в табл. 9.1.

Таблица 9.1. Часто используемые в ms команды форматирования (см. также ms(7))

Команда	Смысл
.AB	начало аннотации; заканчивается .AE
.AU	имя автора на следующей строке; разрешено использование нескольких .AU
.B	начало полужирного шрифта или применение полужирного шрифта к аргументу, если он задан
.DS *t*	начало отображаемого текста (незаполняемого); заканчивается .DE *t* = L (выравнивание по левому краю), C (центрирование), B (блочное центрирование)
.EQ *s*	начало математического выражения *s* (ввод eqn); заканчивается .EN
.FS	начало сноски; заканчивается .FE

Таблица 9.1 (продолжение)

Команда	Смысл
.I	начало курсива или применение курсива к аргументу, если он задан
.IP *s*	абзац с отступом *s*
.KF	сохранять текст неразделенным, если нужно, переместить на следующую страницу; заканчивается .KE
.KS	сохранять текст неразделенным на странице; заканчивается .KE
.LP	новый абзац, выровненный по левому краю
.NH *n*	нумерованный заголовок *n*-го уровня; заголовок следует, до .PP или .LP
.PP	новый абзац
.R	вернуться к прямому шрифту
.SH	подзаголовок; заголовок следует, до .PP
.TL	название следует; до следующей команды ms
.TS	начало таблицы (ввод tbl); заканчивается .TE

Макропакет mm

Не будем входить в детали макропакета mm, так как и по сути своей, да и даже в деталях, он очень похож на ms. В нем предоставляется более полный контроль над параметрами, чем в ms, у него больше возможностей (например, он умеет автоматически нумеровать страницы) и более информативные сообщения об ошибках. В табл. 9.2 представляется список команд mm, эквивалентных командам ms из табл. 9.1.

Таблица 9.2. Часто используемые в mm команды форматирования

Команда	Смысл
.AB	начало аннотации; заканчивается .AE
.AU	имя автора следует за командой как первый аргумент
.B	начало полужирного шрифта или применение полужирного шрифта к аргументу, если он задан
.DF	сохранять текст неразделенным, если нужно, переместить на следующую страницу; заканчивается .DE
.DS	начало текста-отображения; заканчивается .DE
.EQ *s*	начало уравнения *s* (ввод eqn); заканчивается .EN
.FS	начало сноски; заканчивается .FE
.I	начало курсива, или применение курсива к аргументу, если он задан
.H *n* "..."	нумерованный заголовок "..." *n* -го уровня
.HU "..."	ненумерованный заголовок "..."

Команда	Смысл
.P	новый абзац. Используйте .nr Pt 1 для абзацев с отступом
.R	вернуться к прямому шрифту
.TL	название следует; до следующей команды mm
.TS	начало таблицы (ввод tbl); заканчивается .TE

Упражнение 9.1. Пропуск завершающей команды, например .AE или .DE обычно приводит к катастрофе. Напишите программу mscheck, которая бы выявляла ошибки во вводе ms (или того пакета, который вы предпочитаете). Предложение: awk. □

9.2. Использование самой программы troff

В реальной жизни иногда случается так, что надо выйти за пределы возможностей ms, mm или какого-то другого пакета для того, чтобы обратиться к средствам, предлагаемым «голой» программой troff. Однако, учитывая то, что это подобно программированию на языке Ассемблера, стоит обращаться к troff только в случае крайней необходимости и действовать осторожно.

Возможны три ситуации: доступ к специальным символам, изменение шрифта и размера внутри строки и несколько базовых функций форматирования.

Названия символов

Получить доступ к необычным символам: греческим буквам, таким как π, графическим символам (•, † и т. д.) и множеству линий и фигур несложно, хотя осуществляется это не слишком систематично. У каждого такого символа есть имя, которое выглядит как *c*, где *c* – это одиночный символ, или как \(*cd*, где *cd* – это пара символов.

Программа troff выводит знак минус из набора ASCII как дефис -, а не как минус –. Чтобы получить настоящий минус, введите \-, а чтобы получить тире, введите \(em, что означает «em dash» (длинное тире), символ —.

В табл. 9.3 перечислены основные специальные символы; в руководстве по troff их гораздо больше (список специальных символов конкретной системы может отличаться от представленного).

Таблица 9.3. Некоторые специальные последовательности символов в troff

Последовательность		Значение
-	-	дефис
\(hy	-	дефис, такой же, как и выше
\-	–	знак минус в текущем шрифте

Таблица 9.3 (продолжение)

Последовательность	Значение
\(mi −	знак минус в математическом шрифте
\(em —	длинное тире
\&	ничто; защищает начальную точку
blank	пробел фиксированной ширины
|	фиксированный пробел половинной ширины
\e	литеральный символ escape, обычно это \
\(bu	маркер абзаца •
\(dg	крестик †
\(*a	α. \(*b=β, \(*c=ξ, \(*p=π и т. д.
\f*X*	изменение шрифта на *X*; *X*=P – изменение на предыдущий шрифт
\f(*XX*	изменение шрифта на *XX*
\s*n*	изменение на размер *n* пунктов; *n* =0 – предыдущий
\s±*n*	относительное изменение размера в пунктах

Бывают случаи, когда необходимо сообщить troff, что она *не* должна интерпретировать символ, главным образом это касается обратной косой черты и начальной точки. Для этого используются команды \e и \&. Последовательность \e гарантированно выводит обратную косую черту, не интерпретированную программой troff, она используется, когда надо, чтобы выходные данные содержали этот символ. А вот \& – это абсолютное ничто, пробел нулевой ширины. Основное назначение этой последовательности в том, чтобы предотвращать интерпретацию программой troff точек в начале строк. При наборе данной главы две указанные команды использовались очень часто. Например, ms-макет страницы, представленный в начале главы, был введен следующим образом:

```
\&.TL
.I "Title of document"
\&.AU
.I "Author name"
\&.AB
\&...
...
```

Естественно, отрывок, приведенный выше, был введен как

```
\e&.TL
\&.I "Title of document"
\e&.AU
...
```

и вы уже можете себе представить, что, в свою очередь, было введено здесь.

Еще один время от времени встречающийся специальный символ – это *фиксированный пробел*, символ обратной косой черты \, за которым следует пробел. Обычные пробелы troff растягивает, для того чтобы выровнять края, а фиксированный пробел никогда не корректируется, он воспринимается как любой другой символ, и его ширину изменять нельзя. С его помощью также можно передать несколько слов как единый аргумент:

```
.I Title\ of\ document
```

Изменение шрифта и размера

Большинство изменений шрифта и формата реализуются макросами, действующими с начала строки, например .I, но бывает и так, что изменение должно быть осуществлено внутри строки. В частности, символ новой строки является разделителем слов, поэтому, если надо изменить шрифт в середине слова, то макросы использовать нельзя. В этом подразделе будет рассказано о том, как troff решает эту проблему (обратите внимание, что решение обеспечивает именно troff, а не макропакет ms).

Для представления команд, действующих внутри строки (а не с ее начала), troff использует символ обратной косой черты. Две самые распространенные команды – это \f для изменения шрифта и \s для изменения размера в пунктах.

Шрифт указывается в команде \f символом, непосредственно следующим за f:

```
a \fBfriv\fIolous\fR \fIvar\fBiety\fR of \fIfonts\fP
```

выводится как

a **friv***olous va***riety** of *fonts*

Изменение шрифта вида \fP возвращает к предыдущему шрифту, каким бы он ни был (есть только один предыдущий шрифт, стековой структуры не существует).

Названия некоторых шрифтов состоят из двух букв. Они указываются в формате \f(XX, где XX – это название шрифта. Например, шрифт на наборной машине авторов, которым напечатаны листинги в этой книге, назывался CW (Courier Constant Width), поэтому слово keyword вводилось как

```
\f(CWkeyword\fP
```

Вводить это достаточно тяжело, поэтому авторы добавили в ms свой собственный макрос .CW, и больше им уже не приходилось вводить и

читать символы обратной косой черты. Этот макрос применялся для набора слов, расположенных в середине строки, например

```
The
.CW troff
formatter ...
```

Форматирование, заданное макросом, в дальнейшем легко можно изменить.

Изменение размера вводится последовательностью \s*n*, где *n* – это одна или две цифры, указывающие новый размер: \s8 переключает на размер в 8 пунктов. Можно задать и относительное изменение размера, для этого перед цифрой ставится знак плюс или минус. Например, чтобы вывести слова «капителью» (SMALL CAPS)

```
\s-2SMALL CAPS\s0
```

\s0 вызывает возвращение размера к его предыдущему значению. Эта команда является аналогом \fP, но, следуя традициям troff, не называется \sP. Расширение ms, реализованное авторами книги, содержит макрос .UC (upper case – верхний регистр) для выполнения этой работы.

Основные команды troff

Жизненный опыт показывает, что даже при наличии хорошего макропакета необходимо знать несколько команд troff для контроля над заполнением и интервалами, расстановкой знаков табуляции и т. д. Команда .br вызывает разрыв, то есть фрагмент входных данных, следующий за .br, появится в выводе на новой строке. Такую возможность используют, например, чтобы разделить длинное название на части в нужном месте:

```
.TL
Hoc - An Interactive Language
.br
For Floating Point Arithmetic
...
```

Команда .nf выключает режим обычного заполнения строк вывода; каждая строка ввода переходит непосредственно в одну строку вывода. Команда .fi включает режим заполнения обратно. Команда .ce центрирует следующую строку.

Команда .bp начинает новую страницу. Команда .sp вызывает появление в выводе пустой строки. Для команды .sp может быть указан аргумент, задающий количество пустых строк или размер пустого пространства.

```
.sp 3          Оставить 3 строки пробелов
.sp .5         Оставить полстроки пробелов
```

```
.sp 1.5i          Оставить 1,5 дюйма
.sp 3p            Оставить 3 пункта
.sp 3.1c          Оставить 3,1 сантиметра
```

Дополнительное пространство внизу страницы игнорируется, так что `.sp` с большим аргументом эквивалентна `.bp`.

Команда `.ta` расставляет точки табуляции (изначально установлены через полдюйма).

```
.ta n n n ...
```

Вышеприведенная команда устанавливает точки табуляции на указанном расстоянии от левого края; как и в команде `.sp`, каждое число *n* указывается в дюймах, если за ним стоит буква «`i`». Точка табуляции с суффиксом `R` выровняет текст по правому краю (в следующей точке), а суффикс `C` вызовет центрирование.

Команда `.ps` *n* устанавливает размер кегля равным *n*; команда `.ft` *X* задает шрифт *X*. Правила, касающиеся увеличения размера и возвращения к предыдущему значению, аналогичны описанным для команд `\s` и `\f`.

Определение макросов

Попытки охватить все аспекты определения макросов могут завести в такие лабиринты `troff`, что оттуда нам будет уже не выбраться, поэтому остановимся только на основных идеях. Например, вот определение макроса `.CW`:

```
.de CW                  Начало определения
\&\f(CW\\$1\fP\\$2      Изменение шрифта вокруг первого аргумента
..                      Конец определения
```

Команда `\$`*n* выводит значение *n*-го аргумента при обращении к макросу; если *n*-й аргумент не задан, вывод пуст. Двойной символ `\` задерживает оценку `\$`*n* во время определения макроса. Последовательность `\&` предотвращает интерпретацию аргумента в качестве команды `troff` в случае, если он начинается с точки, например

```
.CW .sp
```

9.3. Препроцессоры tbl и eqn

Программа `troff` большая и сложная как внутри, так и снаружи, поэтому изменять ее для того, чтобы она могла выполнить какую-то новую работу, очень непросто. Вот почему в разработке программ для набора математических текстов и таблиц был избран другой подход, были созданы специальные языки, реализованные программами `eqn` и `tbl`, которые действуют по отношению к `troff` как процессоры предва-

рительной обработки. На самом деле troff является языком Ассемблера для наборной машины, а eqn и tbl компилируют в него.

Первой появилась eqn. Тогда yacc был впервые использован не для языка программирования.[1] Потом пришло время tbl, по сути своей она похожа на eqn, а вот синтаксисы у них не родственны друг другу. Программа tbl не использует yacc, так как ее грамматика настолько проста, что это не имеет смысла.

Возможность создания каналов в UNIX подводит к мысли о разумности создания отдельных программ. Благодаря каналам происходит не только разбиение задач на части (что необходимо в любом случае – размер самой troff практически равен максимально допустимому для программы на PDP-11), сокращается обмен данными между частями, а также между программистами. Последнее очень важно, ведь это означает, что для «изготовления» препроцессора не надо иметь доступ к исходным текстам. Кроме того, применение каналов избавляет от огромных промежуточных файлов, если только компоненты не запускаются по отдельности специально, для отладки.

Когда отдельные программы сообщаются через каналы, возникают проблемы. Из-за очень большого количества входных и выходных данных несколько ухудшается производительность: обе программы – и eqn и tbl – обычно вызывают увеличение объема данных от входа к выходу в 8 раз. Еще важнее, что информация движется только в одном направлении. Например, не существует способа, которым eqn могла бы выяснить текущий кегль, что приводит к некоторой громоздкости языка. И наконец, сообщения об ошибках: часто бывает нелегко както связать диагностику, полученную в troff, с ситуацией, возникшей в eqn или tbl и вызвавшей появление этого сообщения.

Но несмотря на все это, преимущества разделения перевешивают недостатки, поэтому было написано несколько других процессоров предварительной обработки, использующих такую же модель.

Таблицы

Начнем с краткого описания tbl, поскольку первое, что будет представлено в данном разделе, – это таблица из документа о hoc. Программа tbl считывает файлы ввода или стандартный ввод и конвертирует текст между командами .TS (table start – начало таблицы) и .TE (table end – конец таблицы) в команды troff для вывода таблицы, выравнивая столбцы и заботясь обо всех типографских деталях. Строки .TS и .TE также копируются, поэтому макропакет может дать им соответствующие определения, например сохранять таблицу на одной странице и отключить опцию обтекания текста.

[1] Маловероятно, что eqn появилась бы на свет, если бы в нужный момент под рукой не оказалось yacc.

Для того чтобы создавать сложные таблицы, надо обратиться к руководству по `tbl`, а пока приведем один пример, которого будет вполне достаточно, чтобы пояснить основные свойства. Таблица взята из документа по `hoc`:

```
.TS
center, box;
c s
lfCW l.
\fBTable 1:\fP Operators, in decreasing order of precedence
.sp .5
^       exponentiation (\s-1FORTRAN\s0 **), right associative
! \-    (unary) logical and arithmetic negation
* /     multiplication, division
+ \-    addition, subtraction
> >=    relational operators: greater, greater or equal,
< <=    less, less or equal,
\&== !=  equal, not equal (all same precedence)
&&      logical AND (both operands always evaluated)
||      logical OR (both operands always evaluated)
\&=     assignment, right associative
.TE
```

так создается таблица, представленная ниже.

Table 1: Operators, in decreasing order of precedence

^	exponentiation (FORTRAN **), right associative
! -	(unary) logical and arithmetic negation
* /	multiplication, division
+ -	addition, subtraction
> >=	relational operators: greater, greater or equal,
< <=	less, less or equal,
== !=	equal, not equal (all same precedence)
&&	logical AND (both operands always evaluated)
||	logical OR (both operands always evaluated)
=	assignment, right associative

Слова перед точкой с запятой (`center`, `box`) описывают глобальные свойства таблицы: горизонтально центрировать ее на странице и нарисовать рамку вокруг нее. Также возможны `doublebox` (двойная рамка), `allbox` (сетка, каждый элемент в рамке) и `expand` (растянуть по ширине страницы).

Следующие строки (до точки) описывают формат частей таблицы, которыми в данном случае являются строка названия и тело таблицы. Первая спецификация относится к первой строке таблицы, вторая –

ко второй, а последняя применяется ко всем оставшимся строкам. Для Table 1 есть только две строки спецификации, поэтому вторая из них применяется ко всем строкам таблицы, кроме первой. Форматирование обозначается следующими символами: `c` – для элементов, центрированных в столбце, `r` и `l` – для выравнивания по правому и левому краю соответственно и `n` – для числового выравнивания по десятичной точке. Символ `s` задает «стягивание» столбцов, в данном случае «`c s`» означает, что надо центрировать название относительно всей таблицы, объединив второй и первый столбцы.

Шрифт можно указывать для столбца; определение `lfCW` выводит столбец, выровненный по левому краю, шрифтом `CW`.

За информацией о формате следует собственно текст таблицы. Столбцы разделены символами табуляции. Некоторые команды `troff`, например `.sp`, воспринимаются и внутри таблиц. (Обратите внимание на использованную дважды последовательность `\&`: незащищенные начальные знаки `-` и `=` в столбцах сообщают `tbl`, что в этом месте следует провести линии через таблицу.)

Программа `tbl` создает множество разных таблиц, отличающихся от представленной в этом простом примере: она вставляет текст в рамки, вертикально выравнивает заголовки столбцов и т. д. Самый простой способ использовать эту программу для сложных таблиц – поискать похожий пример в томе 2A справочного руководства по UNIX и адаптировать его к ситуации.

Математические выражения

Второй препроцессор `troff` – это программа `eqn`, преобразующая язык, описывающий математические выражения, в команды `troff` для вывода. Она автоматически обрабатывает изменения шрифта и размера, а также предоставляет названия стандартных математических символов. Входные данные `eqn` обычно находятся между строками `.EQ` и `.EN`, аналогично строкам `.TS` и `.TE` программы `tbl`. Например,

```
.EQ
x sub i
.EN
```

выводит x_i. Если используется макропакет `ms`, то выражение выводится как выключенная формула,[1] при этом необязательный аргумент `.EQ` указывает номер выражения. Например, формула интеграла Коши

[1] Выключенными, то есть набранными с разрывом текста. В математической литературе набирают обычно большие, сложные математические выражения, которые из-за их физического размера неудобно помещать прямо в тексте. Иногда выключенным может оказаться и простое выражение, если ему придается важное значение. Такие выражения обычно нумеруют по краю полосы набора. – *Примеч. науч. ред.*

$$f(\zeta) = \frac{1}{2\pi i}\int_C \frac{f(z)}{z-\zeta}dz \qquad (9.1)$$

записывается как

```
.EQ (9.1)
f( zeta ) ~=~ 1 over {2 pi i} int from C
    f(z) over {z - zeta} dz
.EN
```

Язык eqn основывается на записи озвучивания математических выражений. Отличие произнесенного вслух математического выражения от ввода eqn заключается в том, что фигурные скобки {} используются в eqn как круглые скобки – они аннулируют правила, определяющие приоритетность по умолчанию, а вот обычные скобки не имеют специального значения. Зато у пробелов оно *есть*. Обратите внимание на то, что в примере, приведенном выше, первая ζ (дзета) окружена пробелами: дело в том, что такие ключевые слова, как zeta и over распознаются, только если они заключены в пробелы или фигурные скобки (при этом ни пробелы, ни фигурные скобки в выводе не присутствуют). Если требуется, чтобы в выводе был пробел, используйте символ тильды ~, как в выражении ~=~. Фигурные скобки можно получить при помощи "{" и "}".

Существует несколько классов ключевых слов eqn. Названия греческих букв записываются в нижнем или верхнем регистре, например lambda и LAMBDA (λ и Λ). Другим математическим символам даны названия, такие как sum (сумма), int (интеграл), infinity (бесконечность) и grad (градиент): Σ, ∫, ∞,∇. Существуют позиционные операторы, такие как sub, sup, from, to и over:

$$\sum_{i=0}^{\infty} x_i^2 \rightarrow \frac{1}{2\pi}$$

записывается как

```
sum from i=0 to infinity x sub i sup 2 ~->~ 1 over {2 pi}
```

Есть такие операторы, как sqrt и изменяемые по высоте скобки и т. д. Также eqn умеет создавать столбцы и матрицы объектов. Существуют и команды для управления размерами, шрифтами и позициями, на тот случай, если установленные по умолчанию не подходят.

Небольшие математические выражения, такие как $\log_{10}(x)$, часто помещают прямо в текст, а не в выключенную формулу. В eqn есть ключевое слово delim, которое задает пару символов, заключающих выражение, находящееся внутри строки, в скобки. Обычно для правого и левого ограничителей используется один и тот же символ, в большинстве случаев это знак доллара $. Но в hoc символ $ применяется для ар-

гументов, поэтому для примеров этой книги в качестве ограничителя был выбран символ @. Подходит и %, но избегайте других символов: дело в том, что очень многие из них имеют специальное значение в огромном количестве программ, и при их использовании поведение программы может стать абсолютно необъяснимым. (Авторы столкнулись с этим, когда писали данную главу.)

Итак, после того как задано

```
.EQ
delim @@
.EN
```

находящееся внутри строки выражение, такое как $\sum_{i=0}^{\infty} x_i$, может быть выведено следующим образом:

```
in-line expression
такое как @sum from i=0 to infinity x sub i@ может быть выведено:
```

Этот прием используют для выведения математических выражений внутри таблицы, так, например, сделано в документации по hoc:

```
.TS
center, box;
c s s
lfCW n l.
\fBTable 3:\fP  Built-in Constants
.sp .5
DEG     57.29577951308232087680    180/ pi@, degrees per radian
E       2.71828182845904523536     @e@, base of natural logarithms
GAMMA   0.57721566490153286060     @gamma@, Euler-Mascheroni constant
PHI     1.61803398874989484820     @( sqrt 5 +1)/2@, the golden ratio
PI      3.14159265358979323846     @pi@, circular transcendental number
.TE
```

Внешний вид выходных данных представлен ниже. Видно, как tbl выравнивает числовой столбец по десятичной точке.

Table 3: Built-in Constants

DEG	57.29577951308232087680	$180/\pi$, degrees per radian
E	2.71828182845904523536	e, base of natural logarithms
GAMMA	0.57721566490153286060	γ, Euler-Mascheroni constant
PHI	1.61803398874989484820	$(\sqrt{5}+1)/2$, the golden ratio
PI	3.14159265358979323846	π, circular transcendental number

И последнее. Поскольку eqn выводит курсивом любую буквенную строку, которую он не распознает, ее принято использовать для печати курсивом обычных слов. @Слово@, например, выводится как *Слово*. Но

будьте внимательны: некоторые общеупотребительные слова (такие как from и to) eqn распознает и обрабатывает их специальным образом, к тому же она уничтожает пробелы, так что этот прием следует применять с осторожностью.

Получение выходных данных

Подготовив документ, надо организовать все препроцессоры и саму программу troff так, чтобы получить выходные данные. Если применялась только troff, введите

```
$ troff -ms имена-файлов                    (Или -mm)
```

Если же использовался препроцессор, укажите имена файлов как аргументы для первой команды в конвейере, а остальные команды должны получать данные со своего стандартного ввода, как в

```
$ eqn имена-файлов | troff -ms
```

или

```
$ tbl имена-файлов | eqn | troff -ms
```

Неудобно все время отслеживать, какой из препроцессоров необходим для вывода каждого конкретного документа. Авторы решили, что будет полезно написать программу (ее назвали doctype), которая трассировала бы соответствующую последовательность команд:

```
$ doctype ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms
$ doctype hoc.ms
cat hoc.ms | tbl | eqn | troff -ms
$
```

Программа doctype реализована средствами, подробно описанными в главе 4; в частности программа awk ищет последовательности команд препроцессоров и выводит командную строку для вызова тех из них, которые нужны для форматирования документа. Она также ищет команду .PP (абзац) макропакета ms.

```
$ cat doctype
# doctype: синтезирует командную строку для troff
echo -n "cat $* | "
egrep -h '^\.(EQ|TS|\[|PS|IS|PP)' $* |
sort -u |
awk '
/^\.PP/ { ms++ }
/^\.EQ/ { eqn++ }
/^\.TS/ { tbl++ }
/^\.PS/ { pic++ }
/^\.IS/ { ideal++ }
```

```
/^\.\[/ { refer++ }
END {
    if (refer > 0) printf "refer | "
    if (pic > 0)   printf "pic | "
    if (ideal > 0) printf "ideal | "
    if (tbl > 0)   printf "tbl | "
    if (eqn > 0)   printf "eqn | "
    printf "troff "
    if (ms > 0) printf "-ms"
    printf "\n"
} '
$
```

(Параметр `-h` команды `egrep` вызывает уничтожение заголовков с именами файлов на каждой строке.) К сожалению, этот параметр доступен не во всех версиях системы. Входные данные сканируются, собирается информация о том, какие компоненты задействованы. После того как весь ввод просмотрен, он в надлежащем порядке обрабатывается для подготовки к выводу. Подробности обработки определяются препроцессорами, форматирующими `troff`-документы, пользователю же остается только переложить работу на машину, она сама обо всем позаботится.

Программа `doctype` – это еще один пример (уже была рассмотрена программа `bundle`) программы, создающей другую программу. Одно замечание: пользователю приходится заново вводить строку для оболочки; в одном из упражнений будет предложено исправить эту ситуацию.

Когда дело доходит до выполнения настоящей команды `troff`, помните, что поведение `troff` определяется конкретной системой: в некоторых случаях она непосредственно управляет наборной машиной, тогда как в других системах программа выводит информацию на устройство стандартного вывода, а затем другая специальная программа посылает данные на наборную машину.

Между прочим, первая версия представленной программы не использовала ни `egrep`, ни `sort`; `awk` сама просматривала все входные данные. Но с большими документами такая программа работала очень медленно, поэтому для быстроты поиска была добавлена `egrep`, а для избавления от повторений (дубликатов) – `sort -u`. В случае типового документа издержки на создание двух дополнительных процессов для «просеивания» данных меньше, чем на выполнение `awk` для большого объема входных данных. Чтобы проиллюстрировать это утверждение, проведено сравнение между `doctype` и версией, которая использует только `awk`; в качестве документа для обработки было выбрано содержимое этой главы (оригинал), содержащей порядка 52 000 символов:

```
$ time awk '... doctype without egrep ...' ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms

real       31.0
```

```
user        8.9
sys         2.8
$ time doctype ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms

real        7.0
user        1.0
sys         2.3
$
```

Преимущество версии с тремя процессами очевидно. (Сравнение было проведено на машине, где работал всего один пользователь, при большой загруженности системы преимущество версии, содержащей `egrep`, было бы еще нагляднее.) Обратите внимание, что сначала была создана простая работающая версия, а потом уже она оптимизировалась.

Упражнение 9.2. Как форматировалась эта глава? □

Упражнение 9.3. Если в качестве символа ограничителя в `eqn` используется знак доллара, как получить этот символ в выводе? Подсказка: исследуйте кавычки и предопределенные (зарезервированные) слова `eqn`. □

Упражнение 9.4. Почему

```
$ `doctype имена-файлов`
```

не работает? Измените `doctype` так, чтобы полученная команда не выводилась, а выполнялась. □

Упражнение 9.5. Значительны ли издержки на дополнительную команду `cat` в `doctype`? Перепишите `doctype` так, чтобы исключить дополнительный процесс. Какая из версий проще? □

Упражнение 9.6. Что лучше: использовать `doctype` или записать командный файл, содержащий команды для форматирования конкретного документа? □

Упражнение 9.7. Поэкспериментируйте с различными комбинациями `grep`, `egrep`, `fgrep`, `sed`, `awk` и `sort` и создайте максимально быструю версию `doctype`. □

9.4. Страница руководства

Основной документацией по команде обычно является страница руководства – одностраничное описание в справочном руководстве по UNIX (см. рис. 9.2). Страница руководства хранится в стандартном каталоге, обычно `/usr/man`, в подкаталоге с номером, соответствующим номеру раздела руководства. Страница руководства по `hoc`, например, хранится в `/usr/man/man1/hoc.1` (так как она описывает пользовательскую команду).

Страницы руководства выводятся при помощи команды man(1) – файла оболочки, который запускает nroff -man, поэтому man hoc печатает руководство по hoc. Если одно имя встречается в нескольких разделах (это касается, например, самой команды man, так как в разделе 1 описывается команда, а в разделе 7 – макрос), можно указать для man номер раздела:

```
$ man 7 man
```

выводит только описание макроса. По умолчанию выводятся все страницы с указанным именем (используется nroff), а man -t генерирует набранные страницы при помощи troff.

Автор страницы руководства создает файл в соответствующем подкаталоге /usr/man. Команда man вызывает nroff или troff с макропакетом для печати страницы, что можно увидеть, просматривая команду man в поиске обращения к форматирующим программам. Результат будет выглядеть следующим образом:

```
$ grep roff `which man`
        nroff $opt -man $all ;;
        neqn $all | nroff $opt -man ;;
        troff $opt -man $all ;;
        troff -t $opt -man $all | tc ;;
        eqn $all | troff $opt -man ;;
        eqn $all | troff -t $opt -man | tc ;;
$
```

Многообразие возникает из-за возможности задания различных параметров: nroff или troff, запускать или нет eqn, и т. д. Макросы руководства, вызываемые troff -man, определяют команды troff, которые осуществляют форматирование в нужном стиле. По существу, они похожи на макросы ms, но есть и отличия, в частности это относится к командам изменения шрифтов и вывода названия. Об этих макросах можно прочитать в man(7) (там представлено их краткое описание), основные же нетрудно запомнить. Макет страницы руководства выглядит так:

```
.TH КОМАНДА номер раздела
.SH NAME
команда \- краткое описание функции
.SH SYNOPSIS
.B команда
параметры
.SH DESCRIPTION
Подробный рассказ о программах и параметрах.
Новые абзацы вводятся командой .PP.
.PP
Это новый абзац.
.SH FILES
файлы, используемые командой, например в passwd(1) упоминается /etc/
   passwd
```

```
.SH "SEE ALSO"
Ссылки на родственные документы, в том числе другие страницы
    руководства
.SH DIAGNOSTICS
Описание любого необычного вывода (например, см. cmp(1))
.SH BUGS
Неожиданные свойства (не обязательно ошибки, см. ниже)
```

Если какая-то секция пуста, ее заголовок опускается. Строка `.TH` и секции `NAME`, `SYNOPSIS` и `DESCRIPTION` обязательны.

Строка

```
.TH КОМАНДА номер-раздела
```

представляет имя команды и указывает номер раздела. Разнообразные строки `.SH` идентифицируют секции страницы руководства. Секции `NAME` и `SYNOPSIS` являются специальными; в остальных содержится обычная скучная информация. Секция `NAME` называет *команду* (на этот раз в нижнем регистре) и предоставляет ее однострочное описание. Секция `SYNOPSIS` перечисляет параметры, но не описывает их. Как и в любой другой секции, входные данные могут вводиться в любом виде, поэтому можно задать изменения шрифта с помощью макросов `.B`, `.I` и `.R`. В секции `SYNOPSIS` название команды и параметры напечатаны полужирным шрифтом, а остальная информация – обычным прямым. Например, секции `NAME` и `SYNOPSIS` для `ed`(1) задаются так:

```
.SH NAME
ed \- text editor
.SH SYNOPSIS
.B ed
[
.B \-
] [
.B \-x
] [ name ]
```

Выводятся они следующим образом:

NAME
ed - text editor
SYNOPSIS
ed [-] [-**x**] [name]

Обратите внимание, что вместо простого символа - используется `\-`.

Секция `DESCRIPTION` описывает команду и ее параметры. В большинстве случаев это именно описание команды, а не языка, который команда определяет. Страница руководства `cc`(1) не описывает язык Си; в ней рассказывается о том, как запустить команду `cc`, чтобы скомпилировать программу, написанную на Си, как вызвать оптимизатор, где остаются выходные данные, и т. д. Язык описывается в справочном руководстве по Си, ссылка на которое приводится в секции `SEE ALSO` стра-

ницы cc(1). С другой стороны, не бывает правил без исключений: man(7) – это описание языка макросов man.

Принято выводить курсивом имена команд и теги для параметров (например, «name» на странице про ed) в секции DESCRIPTION. Макросы .I (выводить первый аргумент курсивом) и .IR (выводить первый аргумент курсивом, а второй – прямым шрифтом) делают эту операцию чрезвычайно простой. Макрос .IR нужен потому, что макрос .I пакета man не обрабатывает второй аргумент тем недокументированным, но удобным способом, каким это делает макрос пакета ms.

Секция FILES указывает файлы, неявно используемые командой. DIAGNOSTICS нужна, только если команда выводит какие-то необычные данные. Это могут быть диагностические сообщения, коды завершения или непонятные вариации нормального поведения команды. Секция BUGS также названа не очень удачно. Те дефекты, о которых в ней сообщается, являются не столько ошибками, сколько недостатками, ведь ошибки должны быть исправлены до того, как команда инсталлируется. Для того чтобы прочувствовать, что же попадает в секцию DIAGNOSTICS, а что – в BUGS, просмотрите стандартное руководство.

Пример должен пролить свет на то, как написать страницу руководства. Исходный текст для hoc(1), /usr/man/man1/hoc.1, представлен на рис. 9.1, а рис. 9.2. отображает вывод команды

```
$ man -t hoc
```

Упражнение 9.8. Напишите страницу руководства для doctype. Напишите версию команды man, которая просматривала бы собственный каталог пользователя man в поиске документации на его личные программы. □

9.5. Другие средства подготовки документов

Есть и другие программы, способные помочь в подготовке документов. Команда refer(1) ищет ссылки по ключевым словам, вносит в документ цитаты (внутрь строк) и организует раздел ссылок в конце документа. Определив соответствующие макросы, можно добиться, чтобы refer выводила ссылки именно так, как вам нужно. Уже существуют готовые определения для ряда компьютерных журналов. Команда refer входит в седьмую версию, но не включена в состав некоторых других версий системы.

Команды pic(1) и ideal(1) делают для иллюстраций то, что eqn делает для формул. Картинки представляют собой гораздо большую сложность, чем формулы (по крайней мере, для наборной машины), к тому же, не существует традиционных способов речевого описания картинок, поэтому для изучения и применения этих двух языков необходимо потрудиться.

```
.TH HOC 1
.SH NAME
hoc \- interactive floating point language
.SH SYNOPSIS
.B hoc
[ file ... ]
.SH DESCRIPTION
.I Hoc
interprets a simple language for floating point arithmetic,
at about the level of BASIC, with C-like syntax and
functions and procedures with arguments and recursion.
.PP
The named
.IR file s
are read and interpreted in order.
If no
.I file
is given or if
.I file
is `\-`
.I hoc
interprets the standard input.
.PP
.I Hoc
input consists of
.I expressions
and
.IR statements .
Expressions are evaluated and their results printed.
Statements, typically assignments and function or procedure
definitions, produce no output unless they explicitly call
.IR print .
.SH "SEE ALSO"
.I
Hoc \- An Interactive Language for Floating Point Arithmetic
by Brian Kernighan and Rob Pike.
.br
.IR bas (1),
.IR bc (1)
and
.IR dc (1).
.SH BUGS
Error recovery is imperfect within function and procedure definitions.
.br
The treatment of newlines is not exactly user-friendly.
```

Рис. 9.1. */usr/man/man1/hoc.1*

HOC(1) HOC(1)

NAME

hoc – interactive floating point language

SYNOPSIS

hoc [file ...]

DESCRIPTION

Hoc interprets a simple language for floating point arithmetic, at about the level of BASIC, with C-like syntax and functions and procedures with arguments and recursion.

The named *files* are read and interpreted in order. If no *file* is given or if *file* is '-' *hoc* interprets the standard input.

Hoc input consists of *expressions* and *statements*. Expressions are evaluated and their results printed. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call

SEE ALSO

Hoc – An Interactive Language for Floating Point Arithmetic by Brian Kernighan and Rob Pike.

bas(1), *bc*(1) and *dc*(1).

BUGS

Error recovery is imperfect within function and procedure definitions.

The treatment of newlines is not exactly user-friendly.

8th Edition 1

Рис. 9.2. *hoc(1)*

Чтобы получить минимальное представление о `pic`, посмотрите на простую иллюстрацию (рис. 9.3) и на ее представление в `pic`.

```
.PS
.ps -1
box invis "document"; arrow
box dashed "pic"; arrow
box dashed "tbl"; arrow
box dashed "eqn"; arrow
box "troff"; arrow
box invis "typesetter"
[ box invis "macro" "package"
  spline right then up -> ] with .ne at 2nd last box.s
```

```
.ps +1
.PE
```

Все рисунки в этой книге сделаны при помощи pic. В состав седьмой версии pic и ideal не входят, но сейчас они уже общедоступны.

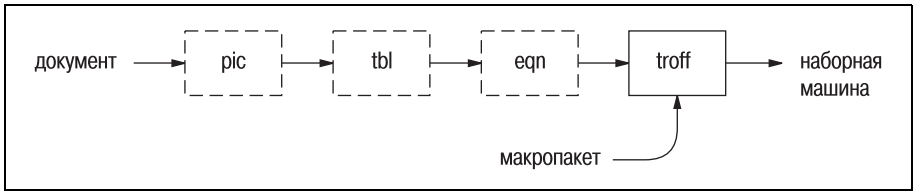

***Рис. 9.3.** Последовательность обработки документа*

И refer, и pic, и ideal представляют собой препроцессоры troff. Существуют также программы, которые просматривают документы и анализируют их на предмет всяких малоинтересных вещей. Самой известной из них является программа spell(1), информирующая о возможных орфографических ошибках в файлах; авторы данной книги все время ее используют. Команды style(1) и diction(1) анализируют пунктуацию, грамматику и стилистику. На базе этих программ был создан Writer's Workbench (инструментальные средства автора) – набор программ, помогающих усовершенствовать стиль. Программы Writer's Workbench удобно применять для распознавания речевых штампов, слов-паразитов и фраз, дискриминирующих женщин.

Программа spell является стандартной. Остальные могут присутствовать в конкретной системе, а могут и не присутствовать, это легко проверить с помощью man:

```
$ man style diction wwb
```

или же можно просмотреть содержимое /bin и /usr/bin.

История и библиография

Программа troff, написанная покойным Джо Оссанной (Joe Ossanna) для наборной машины Graphics Systems CAT-4, имеет богатую родословную, восходящую к RUNOFF, которая была написана Д. Е. Зальтцером (J. E. Saltzer) для совместимых систем с разделением времени в Массачусетском технологическом институте в начале шестидесятых годов. Основные принципы этих программ похожи, как и синтаксис их команд, но troff, без сомнения, самая сложная и мощная из них, а наличие eqn и других препроцессоров делает ее существенно более полезной. Есть и более новые наборные программы с более цивилизованным форматом ввода. Самыми известными являются TeX Дональда Кнута (Donald Knuth) («TeX and Metafont: New Directions in Typeset-

ting» (TeX и Metafont: новые направления в наборе текста), издано Digital Press в 1979 году[1]) и Scribe Брайана Рейда (Brian Reid) («Scribe: a high-level approach to computer docu-ment formatting» (Scribe: высокоуровневый подход к форматированию компьютерного документа), 7th Symposium on the Principles of ProgrammingLanguages, 1980). Доклад «Document Formatting Systems: Survey, Concepts and Issues» (Системы форматирования документов: обзор, концепции и проблемы) Ричарда Фуруты (Richard Furuta), Джеффри Скофилда (Jeffrey Scofield) и Алана Шоу (Alan Shaw) («Computing Surveys», сентябрь 1982 года) предоставляет обзор исследований в данной области.

Первая статья об `eqn` «A system for typesetting mathematics» (Система для набора математических выражений), CACM, март 1975 года была написана Брайаном Керниганом (Brian Kernighan) и Лориндой Черри (Lorinda Cherry). Макропакет `ms`, программы `tbl` и `refer` созданы Майком Леском (Mike Lesk); информация по ним представлена только в томе 2A справочного руководства по UNIX.

Программа `pic` описана в статье «PIC – a language for typesetting graphics» (PIC – язык для набора графики) Брайана Кернигана, изданной в «Software-Practice and Experience» в январе 1982 года. Программа `ideal` представлена в работе «A high-level language for describing pictures» (Язык высокого уровня для описания изображений) Криса Ван Вика (Chris Van Wyk) («ACM Transactions on Graphics», апрель 1982 года).

Команда `spell` превратилась из командного файла, написанного Стивом Джонсоном (Steve Johnson) в программу на Си Дага Мак-Илроя (Doug McIlroy). В седьмой версии `spell` использует механизм хеширования для быстрого просмотра и набор правил для автоматического удаления суффиксов и префиксов, позволяющий хранить небольшой словарь. См. статью Дага Макилроя Development of a spelling list (Разработка словаря для проверки орфографии), «IEEE Transactions on Communications», январь 1982 года.

Программы `style` и `diction` описаны в статье «Computer aids for writers» (Компьютерные вспомогательные средства для писателей) Лоринды Черри (SIGPLAN Symposium on Text Manipulation, Портленд, Орегон, июнь 1981 года).

---

[1] По системе TeX доступна литература и на русском языке. Хотелось бы обратить внимание читателя на две книги, которые можно использовать и как ученые пособия, и как прекрасные справочники по TeX: Кнут. Д. Е. «Все про TeX». – Протвино: АО RDTeX, 1993 и Гусенс М., Миттельбах Ф., Самарин А. «Путеводитель по пакету LaTeX и его расширению LaTeX 2e». – М.: Мир, 1999. – *Примеч. науч. ред.*

10

Эпилог

Операционной системе UNIX уже гораздо больше 10 лет, а количество использующих ее компьютеров растет так быстро, как никогда. Для системы, созданной не для продажи (вообще без такого намерения), это необычайный успех.

Основная причина такого коммерческого успеха, вероятно, кроется в переносимости системы – свойстве, означающем, что все, кроме небольших компонент компиляторов и ядра, одинаково выполняется на любом компьютере. Производителям, устанавливающим программное обеспечение UNIX на своих компьютерах, сравнительно несложно добиться того, чтобы система заработала на новом оборудовании, к тому же они извлекают выгоду из расширяющегося рынка UNIX-программ.

Но надо отметить, что система UNIX была популярна задолго до того, как стала коммерчески значимой, и даже до того, как она заработала на всем остальном, кроме PDP-11. В 1974 году доклад в CACM Денниса Ритчи и Кена Томпсона вызвал интерес университетского сообщества, и к 1975 году системы шестой версии стали общепринятыми в университетах. В середине семидесятых годов рассказы о UNIX передавались из уст в уста: несмотря на то что система не поддерживалась и на нее не было гарантии, люди, использовавшие ее, были настолько полны энтузиазма, что убеждали других опробовать эту систему. Однажды попробовав работать с UNIX, пользователи оставались с ней навсегда; кстати, еще одна причина нынешнего успеха UNIX заключается в том, что поколение программистов, использовавшее университетские UNIX-системы, теперь ожидает найти ее и на своем рабочем месте.

Что было главным? Благодаря чему система стала столь популярной? Она была разработана небольшим количеством исключительно талантливых людей (двумя), единственным стремлением которых было

создать среду, удобную для разработки программ, и которые смогли достичь этого идеала. Первые системы (на которых еще не сказывалось влияние рынка) были достаточно маленькими для того, чтобы их мог понять один человек. Джон Лайонс (John Lions) рассказывал о ядре шестой версии на базовом курсе по операционным системам в университете Нового Южного Уэльса в Австралии. В заметках, сделанных при подготовке к занятиям, он написал «... the whole documentation is not unreasonably transportable in a student's briefcase» (...всю документацию можно унести в портфеле студента). (Это было исправлено в более поздних версиях.)

В этой исходной системе были соединены многие новшества, в том числе обработка потоков (каналы), регулярные выражения, теория языков (`yacc`, `lex` и т. д.) и некоторые специальные компоненты, такие как алгоритм `diff`. Вместе они образовали ядро, обладающее свойствами, которые «редко встречались даже в более крупных операционных системах». В качестве примера рассмотрим структуру ввода-вывода: иерархическую файловую систему, которая нечасто встречалась в то время; устройства, инсталлированные как имена в файловой системе, так что они не требовали специальных обслуживающих программ; и где-то с дюжину необходимых системных вызовов, таких как `open` с ровно двумя аргументами. Программное обеспечение было написано на языке высокого уровня и распространялось вместе с системой, его можно было изучать и модифицировать.

С тех пор система UNIX превратилась в одну из стандартных операционных систем, присутствующих на компьютерном рынке, а с доминированием на рынке возникла ответственность и необходимость предоставления тех возможностей, которыми обладали конкурирующие системы. В результате за последние десять лет ядро увеличилось в размере в 10 раз, хотя усовершенствовано, конечно же, было не настолько впечатляюще. Этот рост сопровождался появлением излишка плохо продуманных программ, которые не были основаны на уже существующем окружении. В угоду предоставлению новых возможностей программы «облеплялись» параметрами, и было уже невозможно понять, для чего же изначально создавалась программа. Поскольку система далеко не всегда распространялась вместе с исходными текстами, то освоить хорошие манеры в написании программ было нелегко.

Однако, к счастью, даже в новых больших версиях осталось то, что сделало популярным первые версии. Принципы, составляющие основу UNIX, – простота структуры, сбалансированность средств, построение новых программ на базе уже имеющихся, программируемость интерпретатора команд, древовидная иерархическая файловая система и т. д. – получили широкое распространение и вытеснили представление о монолитных системах, предшествовавших UNIX. Конечно же, UNIX не будет существовать вечно, но системам, идущим ей на смену, придется воспользоваться многими ее фундаментальными идеями.

В предисловии было сказано, что UNIX – это особый подход к программированию, своя философия. Теперь, когда книга прочитана, вы должны понимать, что имелось в виду, и почувствовать элементы этого особого стиля программирования во всех приведенных примерах.

Во-первых, не делайте сами того, что может сделать машина. Используйте такие программы, как `grep`, `wc` и `awk`, чтобы автоматизировать работу, которую пришлось бы выполнять вручную в другой системе.

Во-вторых, не делайте того, что уже сделали другие. Берите уже существующие программы как блоки для построения своих собственных – оболочка и программируемые фильтры «склеят» их вместе. Напишите маленькую программку, предоставляющую удобный интерфейс для уже существующей, которая и выполняет основную работу (как это было проделано с `idiff`). Среда программирования UNIX предлагает столько различных инструментов, которые можно скомбинировать мириадами способов, что часто задача программиста заключается лишь в том, чтобы найти удачную комбинацию.

В-третьих, выполняйте задачу поэтапно. Напишите сначала самую простую версию, которая делала бы что-то полезное, а потом пусть опыт подскажет вам, что еще стоит добавить (если вообще стоит). Не добавляйте новые возможности и параметры до тех пор, пока статистика не покажет, какие свойства будут востребованы.

В-четвертых, создавайте инструментальные средства. Пишите программы, гармонирующие с существующим окружением, улучшая его, вместо того чтобы просто вносить в него новое. Делайте это как следует, ведь такие программы сами становятся частью инструментария каждого пользователя.

В предисловии было сказано, что система несовершенна. Прочитав десять глав о непонятных условностях, традиционных обозначениях, необоснованных различиях и произвольно выбранных ограничениях, вы, конечно же, согласитесь с таким мнением. Однако несмотря на эти недостатки, реальная польза перевешивает встречающиеся время от времени шероховатости, которые могут вызывать раздражение. Система UNIX на самом деле хороша в том, для чего она и была создана: она предоставляет удобную среду для программирования.

Поэтому, хотя UNIX и начинает стареть, она все еще жизнеспособна, а ее популярность только увеличивается. И этой популярностью система обязана светлой мысли тех нескольких людей, которые в 1969 году нарисовали на черной доске эскиз конструкции программируемой среды, которая казалась им удобной для пользователя. Они не ожидали, что их система будет установлена на десятках тысяч компьютеров, но все сообщество программистов радуется тому, что это произошло.

Приложение 1

Краткое описание редактора

«Стандартным» редактором UNIX является ed, который изначально был написан Кеном Томпсоном (Ken Thompson). Он была разработан в начале семидесятых годов в качестве вычислительной среды малых машин (в первой системе UNIX программа пользователя не могла превышать 8 Кбайт) с чрезвычайно медленными консолями (10–15 символов в секунду). Основой для ed послужил ранее существовавший редактор qed, в свое время пользовавшийся популярностью.

Технологии развивались, а редактор ed оставался все тем же. Вы практически наверняка найдете в своей системе другие редакторы с гораздо более привлекательными возможностями, самым распространенным из них является визуальный или экранный редактор, выводящий внесенные изменения на терминал непосредственно по мере редактирования.

Зачем же тогда тратить время на устаревшую программу? Дело в том, что хотя ed и стар, но некоторые вещи он делает по-настоящему хорошо. Он доступен во всех системах UNIX, так что не нужно беспокоиться при переходе с одной системы на другую. Он хорошо работает в условиях низкоскоростного телефонного соединения и с любыми видами терминалов. Программу ed несложно запустить из скрипта; многие же экранные редакторы полагают, что они управляют терминалом, и не могут забирать входные данные из файла.

В ed существуют регулярные выражения для задания шаблонов. Вся система пронизана регулярными выражениями, образованными из регулярных выражений ed: в grep и sed они практически идентичны; egrep, awk и lex расширили их; оболочка использует другой синтаксис, но те же идеи для задания шаблонов имен файлов. У некоторых экранных редакторов есть «строковый» режим работы, который является версией ed, и в нем можно применять регулярные выражения.

И наконец, ed быстро работает. Вызов ed, изменение одной строки в файле, запись новой версии и выход из программы иногда занимают меньше времени, чем просто запуск большого редактора, имеющего больше возможностей.

Основы

Редактор ed позволяет работать только с одним файлом. Он работает с копией файла; для того чтобы записать изменения в исходный файл, надо ввести явную команду. В ed есть команды для обработки последовательных строк и строк, соответствующих шаблону, можно осуществлять изменения и в пределах строки.

Каждая команда ed обозначается одним символом, обычно буквой. Большая часть команд допускает задание перед именем команды одного или двух *номеров строк*, которые показывают, к каким строкам должна быть применена команда; если номер не задан, используется значение по умолчанию. Номер строки может определяться ее абсолютным местоположением в файле (1, 2, ...), условным обозначением, таким как $ для последней строки и . для текущей, можно осуществить поиск по шаблону с помощью регулярных выражений, а можно использовать аддитивную комбинацию перечисленных способов.

Давайте посмотрим, как создать файл с помощью ed, сделаем это на примере стихотворения Де Моргана (De Morgan) из главы 1.

```
$ ed poem
?poem                  Предупреждение: файл poem не существует
a                      Добавляем строки
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
.                      Вводим "." для остановки добавления
w poem                 Запись строк в файл poem
121                    ed сообщает, что записан 121 символ
q                      Выход
$
```

Команда a добавляет строки или дописывает их в конец файла; режим дозаписи заканчивается строкой, содержащей лишь точку «.». Никакого указания на режим, в котором вы работаете, нет, поэтому постарайтесь избегать двух наиболее распространенных ошибок: вводите текст только после команды a и вводите новые команды только после ввода точки.

Программа ed никогда не записывает текст в файл автоматически, надо сказать ей, чтобы она это сделала, для этого введите команду w. Если пользователь пытается выйти из программы, не записав изменения, ed выводит ? в качестве предупреждения. Если в этот момент ввести еще одну команду q, будет осуществлен выход без сохранения

изменений. Выход по команде Q происходит в любом случае, вне зависимости от того, записаны изменения или нет.

```
$ ed poem
121                    Файл существует, в нем 121 символ
a                      Дописать несколько строк в конец
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.                      Вводим точку для остановки ввода
q                      Попытка выйти
?                      Предупреждение: изменения не записаны
w                      Имя файла не указано, предполагается, что это poem
263
q                      Теперь можно выходить
$ wc poem              Проверим
      8      46     263 poem
$
```

Выход в оболочку с помощью команды !

Во время работы с ed можно временно выйти из нее, чтобы запустить другую команду оболочки, при этом не надо завершать исполнение ed (выходить из него). В такой ситуации используйте команду редактора !:

```
$ ed poem
263
!wc poem               Запустить wc, не выходя из ed
      8      46     263 poem
!                      Возврат после выполнения команды
q                      Можно выйти, не выполняя w: изменений не было
$
```

Печать

Строки файла пронумерованы 1, 2, ...; можно напечатать *n*-ю строку, введя команду *n*p или просто номер *n*; чтобы вывести строки с *m*-й по *n*-ю, введите *m*, *n*p. Номер последней строки обозначается символом $, так что не приходится пересчитывать строки.

```
1                      Вывести 1-ю строку, аналогично 1p
$                      Вывести последнюю строку, аналогично $p
1,$p                   Напечатать строки с 1-й по последнюю
```

Можно построчно вывести файл, просто нажимая *Return*; можно вернуться на строку назад, введя -. Номера строк можно комбинировать при помощи + и -:

```
$-2,$p                 Напечатать последние три строки
1,2+3p                 Напечатать строки с 1-й по 5-ю
```

Но печатать строки в обратном порядке и заходить за конец файла не разрешено; команды вида $,$+1p и $,1p недопустимы.

Команда l (list – перечень) выводит строки в таком формате, который делает видимыми все символы; это удобно для поиска в файле управляющих символов, для распознавания пробелов и знаков табуляции и т. д. (см. команду vis в главе 6).

Шаблоны

Если файл достаточно большой, не очень-то удобно выводить его весь для того, чтобы найти какую-то определенную строку, поэтому ed предоставляет возможность поиска по шаблону: */шаблон/* ищет следующее вхождение *шаблона*.

```
$ ed poem
263
/flea/                  Ищет следующую строку, содержащую flea
Great fleas have little fleas
/flea/                  Ищет следующую строку
And little fleas have lesser fleas,
//                      Ищет следующую строку по тому же шаблону
And the great fleas themselves, in turn,
??                      Поиск по тому же шаблону в обратном направлении
And little fleas have lesser fleas,
```

Шаблон, использованный последним, ed запоминает, поэтому поиск можно продолжить, просто вводя //. Для поиска в обратном направлении используйте ?*шаблон*? и ??.

Поиск, осуществляемый с помощью /.../ и ?...?, выполняется «по кругу», то есть, дойдя до конца текста, возобновляется с начала, и наоборот:

```
$p                      Выводит последнюю строку. (p необязательно)
  and greater still, and so on.
/flea/                  Следующее вхождение flea находится в начале текста
Great fleas have little fleas
??                      Перешагивает через начало текста, выполняя поиск
                          в обратном направлении
  have greater fleas to go on;
```

Шаблон, такой как /flea/, на самом деле является таким же номером строки, как 1 или $, и поэтому может быть использован соответствующим образом:

```
1,/flea/p               Напечатать строки с 1 до следующего вхождения flea
?flea?+1,$p             Напечатать с предыдущей flea +1 до конца
```

Текущая строка

Редактор ed отслеживает последнюю строку, с которой производились какие-то действия: печать или добавление текста, или чтение файла. Такая строка называется текущей строкой и обозначается «.» (точка). Каждая команда воздействует на точку определенным образом, обычно устанавливая ее в последнюю строку, обработанную командой. Точку можно использовать так же, как $ или обычный номер строки, например 1:

```
$ ed poem
263
.                       Вывести текущую строку, аналогично $ после
прочтения
  and greater still, and so on.
.-1,.p                  Вывести предыдущую и данную строки
While these again have greater still,
  and greater still, and so on.
```

Выражения, задающие номера строк, могут записываться в сокращенной форме:

Краткая запись:	*То же самое :*	*Краткая запись:*	*То же самое:*
-	.-1	+	.+1
-- *или* -2	.-2	++ *или* +2	.+2
-*n*	.- *n*	+ *n*	.+ *n*
$-	$-1	.3	.+3

Добавление, удаление, изменение и вставка

Команда a (*append*) добавляет строки после указанной строки; команда d (*delete*) удаляет строки; команда i (*insert*) вставляет строки перед указанной строкой; команда c (*change*) изменяет строки, являясь комбинацией вставки и удаления.

*n*a	*Добавить текст после строки* n
*n*i	*Вставить текст перед строкой* n
m, *n*d	*Удалить строки с* m *по* n
m, *n*c	*Изменить строки с* m *по* n

Если номера строк не указаны, используется точка. Новый текст для команд a, c и i заканчивается строкой, состоящей из символа «.»; при этом точка устанавливается в последнюю добавленную строку. При удалении точка устанавливается в строку, следующую за последней удаленной, но за последнюю строку $ она не переходит.

0a	*Добавить текст в начало (аналогично* 1i*)*
dp	*Удалить текущую строку, вывести следующую (или последнюю, если в* $*)*

`.,$dp`	*Удалить с этого места до конца, вывести новую последнюю строку*
`1,$d`	*Удалить все*
`?pat?,.-1d`	*Удалить с предыдущего «pat» до строки перед точкой*
`$dp`	*Удалить последнюю строку, вывести новую последнюю строку*
`$c`	*Изменить последнюю строку. (*`$a` *добавляет после последней строки)*
`1,$c`	*Изменить все строки*

Замена и отмена

Было бы слишком мучительно вводить заново целую строку для того, чтобы изменить в ней всего несколько букв. Команда `s` (*substitute*) заменяет одну буквенную строку другой:

`s/old/new/`	*Заменить первую* `old` *на* `new` *в текущей строке*
`s/old/new/p`	*Заменить первую* `old` *на* `new` *и напечатать строку*
`s/old/new/g`	*Заменить каждую* `old` *на* `new` *в текущей строке*
`s/old/new/gp`	*Заменить каждую* `old` *на* `new` *и напечатать строку*

Заменяется лишь самое левое вхождение, соответствующее шаблону, на строке, если только в конце команды не указано «`g`». Команда `s` не выводит измененную строку, если только в конце команды не стоит «`p`». В действительности большинство команд `ed` работает молча, но практически любая команда может выводить результат, если за ней следует `p`.

Если замена не привела к ожидаемому результату, то команда `u` (*undo*) может аннулировать последнюю замену. Точка должна быть установлена в замененную строку.

`u`	*Отменить последнюю замену*
`up`	*Отменить последнюю замену и вывести*

Как и команды `p` и `d`, `s` допускает указание перед именем команды одного или двух номеров строк, на которые команда должна воздействовать:

`/old/s/old/new/`	*Найти следующую* `old`*; заменить на* `new`
`/old/s//new/`	*Найти следующую* `old`*; заменить на* `new` *(шаблон запоминается)*
`1,$s/old/new/p`	*Заменить первую* `old` *на* `new` *в каждой строке; вывести последнюю измененную строку*
`1,$s/old/new/gp`	*Заменить каждую* `old` *на* `new` *в каждой строке; вывести последнюю измененную строку*

Обратите внимание, что `1,$s` применяет команду `s` к каждой строке, но все равно означает только самое левое совпадение в каждой строке; а замыкающее `g` заменяет все вхождения во всех строках. Кроме того, `p` выводит только последнюю обработанную строку; для того чтобы вывести все измененные строки, нужна глобальная команда, о ней будет рассказано позже.

Символ & используется для краткости записи; если он появляется в любом месте с правой стороны команды s, то заменяется тем, соответствие чему задано в левой части:

```
s/big/very &/    Заменить big на very big
s/big/& &/       Заменить big на big big
s/.*/(&)/        Заключить в круглые скобки всю строку (см. .* ниже)
s/and/\&/        Заменить and на & (\ аннулирует специальное значение)
```

Метасимволы и регулярные выражения

В оболочке есть символы со специальным значением, такие как *, > и |, есть подобные символы и в ed, здесь они появляются в шаблонах поиска или в левой части команды s. Такие символы называются *метасимволами*, а использующие их шаблоны – *регулярными выражениями*. Символы и их значения перечислены в табл. 1; разбирая примеры, приведенные ниже, обращайтесь к таблице. Специальное значение некоторых символов отменяется, если им предшествует символ обратной косой черты.

Таблица 1. Регулярные выражения редактора

Символ	Значение
c	любой неспециальный символ *c* соответствует сам себе
c	отменяет специальное значение символа *c*
^	соответствует началу строки, если шаблон начинается с ^
$	соответствует концу строки, если шаблон заканчивается $
.	соответствует любому отдельному символу
[...]	соответствует любому из символов в ...; разрешены диапазоны типа a-z
[^...]	соответствует любому из символов, не входящих в ...; разрешены диапазоны
*r**	соответствует нулю или более вхождений *r*, где *r* – это символ, точка или [...]
&	только в правой части команды s; порождает то, чему ищется соответствие
\(...\)	регулярные выражения с тегами; соответствующая строка доступна как \1 и т. д., как с правой, так и с левой стороны

Шаблон:	*Соответствие:*
/^$/	*пустая строка, т. е. только разделитель строк*
/./	*непустая строка, т. е. как минимум один символ*
/^/	*все строки*
/thing/	thing *в любом месте строки*
/^thing/	thing *в начале строки*
/thing$/	thing *в конце строки*

`/^thing$/`	*строка, которая содержит только* `thing`
`/thing.$/`	`thing` *и еще один символ в конце строки*
`/thing\.$/`	`thing.` *в конце строки*
`/\/thing\//`	`/thing/` *в любом месте строки*
`/[tT]hing`	`thing` *или* `Thing` *в любом месте строки*
`/thing[0-9]/`	`thing`, *затем одна цифра*
`/thing[^0-9]/`	`thing`, *затем не цифра*
`/thing[0-9][^0-9]/`	`thing`, *затем сначала цифра, потом не цифра*
`/thing1.*thing2/`	`thing1`, *затем любая строка, затем* `thing2`
`/^thing1.*thing2$/`	`thing1` *в начале и* `thing2` *в конце*

Регулярные выражения, содержащие символ `*`, выбирают самое левое соответствие и продолжают его как можно дальше. Обратите внимание на то, что `x*` может соответствовать нулю символов, а `xx*` соответствует одному или более символам.

Глобальные команды

Глобальные команды `g` и `v` применяют одну или несколько других команд к множеству строк, заданному регулярным выражением. Команда `g` чаще всего применяется для печати, замены или удаления множества строк:

m,n`g`*/re/cmd*	*Выполнить cmd для всех строк между m и n, соответствующих re*
m,n`v`*/re/cmd*	*Выполнить cmd для всех строк между m и n, не соответствующих re*

Команде `g` или `v` могут предшествовать номера строк, ограничивающих диапазон применения; по умолчанию принят `1,$`.

`g/.../p`	*Вывести все строки, соответствующие регулярному выражению ...*
`g/.../d`	*Удалить все строки, соответствующие ...*
`g/.../s//`*repl*`/p`	*Заменить первое вхождение ... на каждой строке на «repl», вывести измененные строки*
`g/.../s//` *repl*`/gp`	*Заменить каждое вхождение ... на «repl», вывести измененные строки*
`g/.../s/`*pat*`/` *repl*`/`	*На строках, соответствующих ..., заменить первое «pat» на «repl»*
`g/.../s/`*pat*`/` *repl*`/p`	*На строках, соответствующих ..., заменить первое «pat» на «repl» и вывести*
`g/.../s/`*pat*`/` *repl*`/gp`	*На строках, соответствующих ..., заменить все «pat» на «repl» и вывести*
`v/.../s/`*pat*`/` *repl*`/gp`	*На строках, не соответствующих ..., заменить все «pat» на «repl» и вывести*
`v/^$/p`	*Вывести все непустые строки*
`g/.../`*cmd1*`\`	*Чтобы выполнить несколько команд с одной* `g`,
cmd2`\`	*добавьте* `\` *в конец каждой cmd*
cmd3	*кроме последней*

Команды, управляемые глобальной командой `g` или `v`, тоже могут использовать номера строк. Точка устанавливается по очереди в каждую выбранную строку.

```
g/thing/.,.+1p              Вывести каждую строку с thing и следующую
g/^\.EQ/.1,/^\.EN/-s/alpha/beta/gp    Заменить alpha на beta только
                                  между .EQ и .EN, и вывести измененные строки
```

Перемещение и копирование строк

Команда `m` перемещает группу последовательных строк; команда `t` создает копию группы строк в каком-то другом месте.

```
m,nm d      Перемещает строки с m по n в положение после строки d
m,nt d      Копирует строки с m по n в положение после строки d
```

Если исходные строки не указаны, используется точка. Строка назначения не может находиться в диапазоне *m, n-1*. Приведем характерные способы применения `m` и `t`:

```
m+          Переместить текущую строку за следующую (перестановка)
m-2         Переместить текущую строку перед предыдущей
m--         Аналогично, -- равнозначно -2
m-          Ничего не происходит
m$          Переместить текущую строку в конец (m0 перемещает в начало)
t.          Продублировать текущую строку (t$ дублирует в конец)
-,.t.       Продублировать предыдущую и текущую строки
1,$t$       Продублировать все множество строк
g/^/m0      Обратный порядок строк
```

Метки и номера строк

Команда `=` выводит номер для строки `$` (по умолчанию), `.=` выводит номер текущей строки и т. д. Точка не изменяется.

Команда `k`*c* *помечает* строку, к которой происходит обращение, буквой *c* (в нижнем регистре); теперь к этой строке можно обращаться как к `'`*c*. Команда `k` не изменяет точку. Метки удобно использовать при перемещении больших частей текста, так как они привязаны к строкам навсегда:

```
/.../ka     Найти строку ... и пометить ее как a
/.../kb     Найти строку ... и пометить ее как b
'a,'bp      Вывести весь диапазон, чтобы быть уверенным
/.../       Найти строку назначения
'a,'bm.     Переместить выбранные строки за данную
```

Объединение, разбиение и перегруппировка строк

Строки можно объединить при помощи команды `j` (пробелы не добавляются):

```
m,nj        Объединить строки с m по n в одну строку
```

Диапазон по умолчанию – `.,.+1`, так что

```
jp              Объединить текущую строку со следующей и вывести
-,.jp           Объединить предыдущую строку с текущей и вывести
```

Строку можно разбить на части при помощи команды замены и заключения в кавычки разделителя строк:

```
s/part1part2/part1\          Разделить строку на две части
part2/              ...
s/ /\                        Разделить на каждом пробеле;
/g                             получается по одному слову на строке
```

Точка остается на последней созданной строке.

Чтобы работать с частями соответствующих регулярных выражений, а не с целыми, используйте *регулярные выражения с тегами*: если в регулярном выражении появляется конструкция вида `\(...\)`, то часть целого, которой она соответствует, доступна как с правой, так и с левой стороны как `\1`. Может быть задано до девяти регулярных выражений с тегами, на которые ссылаются как на `\1`, `\2` и т. д.

```
s/\(...\)\(.*\)/\2\1/     Переместить первые 3 символа в конец
/\(..*\)\1/               Найти строки, содержащие
                            повторяющуюся соседнюю строку
```

Файловые команды

Команды *чтения* и *записи* `r` и `w` могут применяться с указанными перед ними номерами строк:

```
nr file         Прочитать file; добавить его после строки n; установить
                    точку в последнюю прочитанную строку
m,nw file       Записать строки m-n в file; точка не изменяется
m,nW file       Записать строки m-n в конец file; точка не изменяется
```

По умолчанию диапазоном в командах `w` и `W` является весь файл. Значением по умолчанию *n* в команде `r` является `$`, что не очень-то удачно. Будьте внимательны!

Редактор `ed` запоминает первое имя файла, указанное в командной строке или в команде `r` или `w`. Команда `f` выводит или изменяет имя запомненного файла:

```
f               Вывести имя запомненного файла
f file          Заменить запомненное имя на «file»
```

Команда редактирования `e` заново инициализирует `ed` запомненным или новым файлом:

```
e               Начать редактирование запомненного файла
e file          Начать редактирование «file»
```

Команда e обладает тем же свойством, что и q: если сделанные изменения не были сохранены, то после первого ввода e появится сообщение об ошибке. Команда e реинициализирует вне зависимости от того, сохранены изменения или нет. В некоторых системах ed и e связаны ссылкой, поэтому одна и та же команда (e *имя-файла*) может быть использована как внутри редактора, так и вне его.

Шифрование

Файлы могут быть зашифрованы при записи и расшифрованы при чтении с помощью команды x, при этом будет запрошен пароль. Шифрование аналогично crypt(1). В некоторых системах команда x была заменена на X (верхний регистр), чтобы избежать неумышленного шифрования.

Сводка команд

Список команд ed представлен в табл. 2, а разрешенные номера команд – в табл. 3. Каждой команде предшествует ноль, один или два номера строк, которые указывают, сколько номеров строк можно задавать, и значения по умолчанию, если номера не заданы. За большей частью команд может следовать p, тогда выводится последняя обработанная строка или l для перечисления строк. Точка обычно устанавливается в последнюю обработанную строку; команды f, k, w, x, = и ! не изменяют ее.

Упражнение. Когда вы решите, что знаете ed, опробуйте редактор quiz; см. quiz(6). □

Таблица 2. Команды редактора

Команда	Действие
.a	добавляет текст до тех пор, пока не будет введена строка, состоящая из одного символа «.»
.,.c	изменяет строки; новый текст заканчивается так же, как и в a
.,.d	удаляет строки
e *file*	реинициализирует для *file*. Команда E делает это, даже если изменения не сохранены
f *file*	устанавливает запомненный файл в *file*
1,$g/*re*/*cmds*	выполняет команду ed *cmds* для каждой строки, соответствующей регулярному выражению *re*; несколько *cmds* разделяются \ *newline* (символ новой строки)
.i	вставляет текст перед строкой, заканчивается так же, как и в a
.,.+1j	объединяет строки в одну
.k*c*	помечает строку буквой *c*

Таблица 2 (продолжение)

Команда	Действие
.,.l	перечисляет строки, делает видимыми невидимые символы
.,.m*line*	передвигает строки за *line*
.,.p	печатает строки
q	выход; Q выходит, даже если изменения не сохранены
$r *file*	читает *file*
.,.s/*re*/*new*/	заменяет на *new* то, что соответствует *re*
.,.t*line*	копирует строки после *line*
.u	отменяет последнюю замену на строке (только одну)
1,$v/*re*/*cmds*	выполняет команду ed *cmds* для каждой строки, не соответствующей *re*
1,$w *file*	записывает строки в *file;* W дописывает в конец вместо того, чтобы перезаписывать
x	вход в режим шифрования (или ed -x *имя-файла*)
$=	выводит номер строки
!*cmdline*	выполняет команду UNIX *cmdline*
(.+1)*newline*	выводит строку

Таблица 3. Номера строк редактора

Обозначение	Смысл
n	абсолютный номер строки *n*, *n* = 0, 1, 2, ...
.	текущая строка
$	последняя строка текста
/*re*/	следующая строка, соответствующая *re*; переходит с $ на 1
? *re*?	предыдущая строка, соответствующая *re*; переходит с 1 на $
‘*c*	строка с пометкой *c*
N1+-*n*	строка *N1*± *n* (аддитивная комбинация)
N1,*N2*	строки с *N1* по *N2*
N1;*N2*	устанавливает точку в *N1*, затем оценивает *N2* *N1* и *N2* могут быть заданы любым из перечисленных выше способов

Приложение 2

Руководство по НОС

Нос — интерактивный язык для математических выражений с плавающей точкой

Брайан Кернига́н
Роб Пайк

АННОТАЦИЯ

Нос – это простой программируемый интерпретатор для выражений с плавающей точкой. В нем имеются Си-подобные операторы управления, определение функций и обычные встроенные числовые функции, такие как косинус и логарифм.

1. Выражения

Нос представляет собой язык выражений, подобный Си. В нем существует некоторое количество операторов управления, но большинство операторов (например, присваивание) являются выражениями, значение которых игнорируется. Например, оператор присваивания = присваивает значение своего правого операнда левому операнду и выдает значение, так что работают множественные присваивания. Грамматика выражения такова:

```
expr:       number
          | variable
          | ( expr )
          | expr binop expr
          | unop expr
          | function ( arguments )
```

Используются числа с плавающей точкой. Формат ввода такой, который распознается `scanf(3)`: цифры, десятичная точка, цифры, `e` или `E`, порядок со знаком. Должна быть указана по крайней мере одна цифра или десятичная точка; другие компоненты не обязательны.

Имена переменных образованы буквой, за которой следует буквенно-цифровая строка. Бинарные операции, такие как сложение или логическое сравнение, обозначены *binop*; *unop* – это два оператора отрицания: `!` (логическое отрицание, «не») и `-` (арифметическое отрицание, изменение знака). Операторы перечислены в табл. 1.

Таблица 1. Операторы в порядке понижения приоритета

Оператор	Действие
`^`	возведение в степень (как `**` в Фортране), ассоциативен справа
`! -`	(унарные) логическое и арифметическое отрицание
`* /`	умножение, деление
`+ -`	сложение, вычитание
`> >=`	операторы отношения: больше, больше или равно
`< <=`	меньше, меньше или равно
`== !=`	равно, не равно (все с одинаковым приоритетом)
`&&`	логическое И (всегда оцениваются оба операнда)
`||`	логическое ИЛИ (всегда оцениваются оба операнда)
`=`	присваивание, ассоциативно справа

Функции, как будет рассказано дальше, могут быть определены пользователем. Аргументами функций являются выражения, разделенные запятыми. Существуют и встроенные функции (все они принимают только один аргумент), они описаны в табл. 2.

Таблица 2. Встроенные функции

Функция	Возвращаемое значение
`abs(x)`	$\|x\|$, абсолютная величина x
`atan(x)`	арктангенс x
`cos(x)`	$\cos(x)$, косинус x
`exp(x)`	e^x, экспонента x
`int(x)`	целая часть x, дробная часть отбрасывается
`log(x)`	$\log(x)$, логарифм x по основанию e
`log10(x)`	$\log10(x)$, логарифм x по основанию 10
`sin(x)`	$\sin(x)$, синус x
`sqrt(x)`	корень квадратный из x; $x1/2$

Логические выражения имеют значение 1,0 (истина) и 0,0 (ложь). Как и в Си, любая ненулевая величина воспринимается как истина. Как это всегда бывает с числами с плавающей точкой, установление равенства крайне сомнительно.

В hoc есть также несколько встроенных констант, представленных в табл. 3.

Таблица 3. Встроенные константы

Константа	Значение	Пояснение
DEG	57,29577951308232087680	$180/\pi$, градусов в радиане
E	2,71828182845904523536	e, основание натурального логарифма
GAMMA	0,57721566490153286060	γ, константа Эйлера–Маскерони
PHI	1,61803398874989484820	$(\sqrt{5}+1)/2$, золотое сечение
PI	3,14159265358979323846	π, отношение длины окружности к ее диаметру

2. Операторы и управляющая логика

Операторы hoc имеют следующую грамматику:

```
stmt:           expr
        |       variable = expr
        |       procedure ( arglist )
        |       while ( expr ) stmt
        |       if ( expr ) stmt
        |       if ( expr ) stmt else stmt
        |       { stmtlist }
        |       print expr-list
        |       return optional-expr

stmtlis          (nothing)
        |       stmtlist  stmt
```

По умолчанию присваивание анализируется как оператор, а не как выражение, поэтому присваивания, введенные в интерактивном режиме, не выводят свои значения.

Обратите внимание, что точка с запятой не имеет специального значения в hoc: операторы заканчиваются разделителями строк. Этим объясняются некоторые особенности поведения. Ниже приведены примеры правильно построенных операторов if:

```
if (x < 0) print(y) else print(z)

if (x < 0) {
       print(y)
} else {
       print(z)
}
```

Во втором примере наличие фигурных скобок обязательно: если бы их не было, то символ разделителя строк, введенный в конце строки с `if`, воспринимался бы как конец оператора и выдавалось бы сообщение о синтаксической ошибке.

Синтаксис и семантика средств управляющей логики `hoc` аналогичны действующим в Си. Операторы `while` и `if` практически повторяют операторы Си, только в `hoc` нет операторов `break` и `continue`.

3. Ввод и вывод: read и print

Функция ввода `read`, как и другие встроенные функции, принимает один аргумент. Но в отличие от других встроенных функций, аргумент `read` – это не выражение, а имя переменной. Следующее число (как описано выше) считывается с устройства стандартного ввода и присваивается указанной переменной. Значение, возвращаемое `read`, равняется `1` (истина), если считывание успешно, и `0` (ложь), если достигнут конец файла или произошла ошибка.

Вывод генерируется оператором `print`. Аргументами `print` является список выражений и строк, заключенных в двойные кавычки, которые разделены запятыми (как в Си). Разделители строк необходимо вводить, автоматически `print` их не предоставляет.

Обратите внимание, что, будучи специальной встроенной функцией, `read` принимает только один аргумент, заключенный в скобки, тогда как `print` – это оператор, принимающий не заключенный в скобки список аргументов, разделенных запятыми:

```
while (read(x)) {
       print "value is ", x, "\n"
}
```

4. Функции и процедуры

В `hoc` различаются функции и процедуры (хотя определяются они одним и тем же способом). Отличие есть только в контроле ошибок времени исполнения: для процедуры ошибкой является возвращение величины, а для функции – *не*возвращение.

Синтаксис определения выглядит следующим образом:

function: *func name( ) stmt*

procedure: *proc name( ) stmt*

name может быть именем любой переменной, за исключением встроенных функций. Определение (вплоть до открывающей фигурной скобки или оператора), должно находиться на одной строке, как оператор `if` в примере выше.

В отличие от Си, телом функции или процедуры может быть любой оператор, не обязательно составной (заключенный в фигурные скобки). Поскольку точка с запятой не имеет специального значения в hoc, то нулевое тело формируется пустой парой фигурных скобок.

При вызове функции и процедуры могут принимать аргументы, разделенные запятыми. Аргументы указываются так же, как и в оболочке: $3 ссылается на третий (нумерация начинается с единицы) аргумент. Они передаются по значению и внутри функций семантически эквивалентны переменным. Ссылка на аргумент с номером, превышающим количество аргументов, переданных программе, является ошибкой. Впрочем, контроль над ошибками осуществляется динамически, поэтому функция может иметь непостоянное количество аргументов, если начальные аргументы изменяют количество аргументов, на которые ссылаются (как в Си-функции printf).

Функции и процедуры могут быть рекурсивными, но размер стека ограничен (около сотни вызовов). Покажем это на примере функции Аккермана (Ackermann), определенной в hoc:

```
$ hoc
func ack() {
        if ($1 == 0) return $2+1
        if ($2 == 0) return ack($1-1, 1)
        return ack($1-1, ack($1, $2-1))
}
ack(3, 2)
        29
ack(3, 3)
        61
ack(3, 4)
hoc: stack too deep near line 8
...
```

5. Примеры

Формула Стирлинга:

$$n! \sim \sqrt{2n\pi}(n/e)^n\left(1+\frac{1}{12n}\right)$$

```
$ hoc
func stirl() {
     return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
}
stirl(10)
        3628684.7
stirl(20)
        2.4328818e+18
```

Факториальная функция, $n!$:

```
func fac() if ($1<= 0) return 1 else return $1 * fac($1-1)
```

Отношение факториала к приближенному значению, вычисленному по формуле Стирлинга:

```
i = 9
while ((i = i+1) <= 20) {
        print i, "  ", fac(i)/stirl(i), "\n"
}
10   1.0000318
11   1.0000265
12   1.0000224
13   1.0000192
14   1.0000166
15   1.0000146
16   1.0000128
17   1.0000114
18   1.0000102
19   1.0000092
20   1.0000083
```

Приложение 3

Исходный код HOC

Здесь во всей своей полноте представлен листинг hoc6.

```
*****         hoc.y   ************************************************

%{
#include "hoc.h"
#define        code2(c1,c2)        code(c1); code(c2)
#define        code3(c1,c2,c3)     code(c1); code(c2); code(c3)
%}
%union {
        Symbol        *sym;        /* symbol table pointer */
        Inst        *inst;        /* machine instruction */
        int        narg;        /* number of arguments */
}
%token        <sym>        NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token        <sym>        FUNCTION PROCEDURE RETURN FUNC PROC READ
%token        <narg>       ARG
%type        <inst>        expr stmt asgn prlist stmtlist
%type        <inst>        cond while if begin end
%type        <sym>         procname
%type        <narg>        arglist
%right        '='
%left        OR
%left        AND
%left        GT GE LT LE EQ NE
%left        '+' '-'
%left        '*' '/'
%left        UNARYMINUS NOT
%right        '^'
%%
list:          /* nothing */
        | list '\n'
        | list defn '\n'
```

```
        | list asgn '\n'  { code2(pop, STOP); return 1; }
        | list stmt '\n'  { code(STOP); return 1; }
        | list expr '\n'  { code2(print, STOP); return 1; }
        | list error '\n' { yyerrok; }
        ;
asgn:         VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
        | ARG '=' expr
           { defnonly("$"); code2(argassign,(Inst)$1); $$=$3;}
        ;
stmt:         expr        { code(pop); }
        | RETURN { defnonly("return"); code(procret); }
        | RETURN expr
              { defnonly("return"); $$=$2; code(funcret); }
        | PROCEDURE begin '(' arglist ')'
              { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
        | PRINT prlist        { $$ = $2; }
        | while cond stmt end {
              ($1)[1] = (Inst)$3;        /* body of loop */
              ($1)[2] = (Inst)$4; }        /* end, if cond fails */
        | if cond stmt end {        /* else-less if */
              ($1)[1] = (Inst)$3;        /* thenpart */
              ($1)[3] = (Inst)$4; }        /* end, if cond fails */
        | if cond stmt end ELSE stmt end {        /* if with else */
              ($1)[1] = (Inst)$3;        /* thenpart */
              ($1)[2] = (Inst)$6;        /* elsepart */
              ($1)[3] = (Inst)$7; }        /* end, if cond fails */
        | '{' stmtlist '}'        { $$ = $2; }
        ;
cond:         '(' expr ')'        { code(STOP); $$ = $2; }
        ;
while:         WHILE        { $$ = code3(whilecode,STOP,STOP); }
        ;
if:          IF        { $$ = code(ifcode); code3(STOP,STOP,STOP); }
        ;
begin:         /* nothing */                { $$ = progp; }
        ;
end:         /* nothing */                { code(STOP); $$ = progp; }
        ;
stmtlist: /* nothing */                { $$ = progp; }
        | stmtlist '\n'
        | stmtlist stmt
        ;
expr:         NUMBER { $$ = code2(constpush, (Inst)$1); }
        | VAR         { $$ = code3(varpush, (Inst)$1, eval); }
        | ARG         { defnonly("$"); $$ = code2(arg, (Inst)$1); }
        | asgn
        | FUNCTION begin '(' arglist ')'
              { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
        | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
        | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
        | '(' expr ')'        { $$ = $2; }
```

```
        | expr '+' expr       { code(add); }
        | expr '-' expr       { code(sub); }
        | expr '*' expr       { code(mul); }
        | expr '/' expr       { code(Div); }
        | expr '^' expr       { code (power); }
        | '-' expr   %prec UNARYMINUS   { $$=$2; code(negate); }
        | expr GT expr        { code(gt); }
        | expr GE expr        { code(ge); }
        | expr LT expr        { code(lt); }
        | expr LE expr        { code(le); }
        | expr EQ expr        { code(eq); }
        | expr NE expr        { code(ne); }
        | expr AND expr       { code(and); }
        | expr OR expr        { code(or); }
        | NOT expr            { $$ = $2; code(not); }
        ;
prlist:   expr                  { code(prexpr); }
        | STRING                { $$ = code2(prstr, (Inst)$1); }
        | prlist ',' expr       { code(prexpr); }
        | prlist ',' STRING     { code2(prstr, (Inst)$3); }
        ;
defn:          FUNC procname { $2->type=FUNCTION; indef=1; }
             '(' ')' stmt { code(procret); define($2); indef=0; }
        | PROC procname { $2->type=PROCEDURE; indef=1; }
             '(' ')' stmt { code(procret); define($2); indef=0; }
        ;
procname: VAR
        | FUNCTION
        | PROCEDURE
        ;
arglist:  /* nothing */         { $$ = 0; }
        | expr                  { $$ = 1; }
        | arglist ',' expr      { $$ = $1 + 1; }
        ;
%%
        /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char       *progname;
int        lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf    begin;
int        indef;
char       *infile;        /* input file name */
FILE       *fin;           /* input file pointer */
char       **gargv;       /* global argument list */
int        gargc;

int c;        /* global for use by warning() */
yylex()               /* hoc6 */
```

```
{
        while ((c=getc(fin)) == ' ' || c == '\t')
                ;
        if (c == EOF)
                return 0;
        if (c == '.' || isdigit(c)) {        /* number */
                double d;
                ungetc(c, fin);
                fscanf(fin, "%lf", &d);
                yylval.sym = install("", NUMBER, d);
                return NUMBER;
        }
        if (isalpha(c)) {
                Symbol *s;
                char sbuf[100], *p = sbuf;
                do {
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("name too long", sbuf);
                        }
                        *p++ = c;
                } while ((c=getc(fin)) != EOF && isalnum(c));
                ungetc(c, fin);
                *p = '\0';
                if ((s=lookup(sbuf)) == 0)
                        s = install(sbuf, UNDEF, 0.0);
                yylval.sym = s;
                return s->type == UNDEF ? VAR : s->type;
        }
        if (c == '$') {        /* argument? */
                int n = 0;
                while (isdigit(c=getc(fin)))
                        n = 10 * n + c - '0';
                ungetc(c, fin);
                if (n == 0)
                        execerror("strange $...", (char *)0);
                yylval.narg = n;
                return ARG;
        }
        if (c == '"') {        /* quoted string */
                char sbuf[100], *p, *emalloc();
                for (p = sbuf; (c=getc(fin)) != '"'; p++) {
                        if (c == '\n' || c == EOF)
                                execerror("missing quote", "");
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("string too long", sbuf);
                        }
                        *p = backslash(c);
                }
                *p = 0;
```

```
                yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
                strcpy(yylval.sym, sbuf);
                return STRING;
        }
        switch (c) {
        case '>':       return follow('=', GE, GT);
        case '<':       return follow('=', LE, LT);
        case '=':       return follow('=', EQ, '=');
        case '!':       return follow('=', NE, NOT);
        case '|':       return follow('|', OR, '|');
        case '&':       return follow('&', AND, '&');
        case '\n':      lineno++; return '\n';
        default:        return c;
        }
}

backslash(c)        /* get next char with \'s interpreted */
        int c;
{
        char *index();        /* `strchr()' in some systems */
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if (c != '\\')
                return c;
        c = getc(fin);
        if (islower(c) && index(transtab, c))
                return index(transtab, c)[1];
        return c;
}

follow(expect, ifyes, ifno)        /* look ahead for >=, etc. */
{
        int c = getc(fin);

        if (c == expect)
                return ifyes;
        ungetc(c, fin);
        return ifno;
}

defnonly(s)        /* warn if illegal definition */
        char *s;
{
        if (!indef)
                execerror(s, "used outside definition");
}

yyerror(s)        /* report compile-time error */
        char *s;
{
        warning(s, (char *)0);
}

execerror(s, t)        /* recover from run-time error */
        char *s, *t;
```

```
{
        warning(s, t);
        fseek(fin, 0L, 2);                /* flush rest of file */
        longjmp(begin, 0);
}

fpecatch()        /* catch floating point exceptions */
{
        execerror("floating point exception", (char *) 0);
}

main(argc, argv)        /* hoc6 */
        char *argv[];
{
        int i, fpecatch();

        progname = argv[0];
        if (argc == 1) {        /* fake an argument list */
                static char *stdinonly[] = { "-" };

                gargv = stdinonly;
                gargc = 1;
        } else {
                gargv = argv+1;
                gargc = argc-1;
        }
        init();
        while (moreinput())
                run();
        return 0;
}

moreinput()
{
        if (gargc-- <= 0)
                return 0;
        if (fin && fin != stdin)
                fclose(fin);
        infile = *gargv++;
        lineno = 1;
        if (strcmp(infile, "-") == 0) {
                fin = stdin;
                infile = 0;
        } else if ((fin=fopen(infile, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n", progname, infile);
                return moreinput();
        }
        return 1;
}

run()        /* execute until EOF */
{
        setjmp(begin);
        signal(SIGFPE, fpecatch);
```

```
        for (initcode(); yyparse(); initcode())
                execute(progbase);
}

warning(s, t)        /* print warning message */
        char *s, *t;
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t)
                fprintf(stderr, " %s", t);
        if (infile)
                fprintf(stderr, " in %s", infile);
        fprintf(stderr, " near line %d\n", lineno);
        while (c != '\n' && c != EOF)
                c = getc(fin);        /* flush rest of input line */
        if (c == '\n')
                lineno++;
}

*****         hoc.h   ************************************************

typedef struct Symbol {        /* symbol table entry */
        char        *name;
        short       type;
        union {
                double        val;             /* VAR */
                double        (*ptr)();        /* BLTIN */
                int        (*defn)();          /* FUNCTION, PROCEDURE */
                char        *str;              /* STRING */
        } u;
        struct Symbol        *next;        /* to link to another */
} Symbol;
Symbol        *install(), *lookup();

typedef union Datum {        /* interpreter stack type */
        double  val;
        Symbol  *sym;
} Datum;
extern        Datum pop();
extern        eval(), add(), sub(), mul(), Div(), negate(), power();

typedef int (*Inst)();
#define STOP    (Inst) 0

extern        Inst *progp, *progbase, prog[], *code();
extern        assign(), bltin(), varpush(), constpush(), print(), varread();
extern        prexpr(), prstr();
extern        gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern        ifcode(), whilecode(), call(), arg(), argassign();
extern        funcret(), procret();

*****         symbol.c   *********************************************

#include "hoc.h"
#include "y.tab.h"
```

```
static Symbol *symlist = 0;  /* symbol table: linked list */

Symbol *lookup(s)        /* find s in symbol table */
        char *s;
{
        Symbol *sp;

        for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;
        return 0;        /* 0 ==> not found */
}

Symbol *install(s, t, d)  /* install s in symbol table */
        char *s;
        int t;
        double d;
{
        Symbol *sp;
        char *emalloc();

        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist; /* put at front of list */
        symlist = sp;
        return sp;
}

char *emalloc(n)        /* check return from malloc */
        unsigned n;
{
        char *p, *malloc();

        p = malloc(n);
        if (p == 0)
                execerror("out of memory", (char *) 0);
        return p;
}

*****         code.c   ************************************************

#include "hoc.h"
#include "y.tab.h"
#include <stdio.h>

#define  NSTACK        256

static Datum stack[NSTACK];        /* the stack */
static Datum *stackp;              /* next free spot on stack */

#define        NPROG        2000
Inst        prog[NPROG];        /* the machine */
```

```
Inst        *progp;             /* next free spot for code generation */
Inst        *pc;                /* program counter during execution */
Inst        *progbase = prog;   /* start of current subprogram */
int        returning;           /* 1 if return stmt seen */

typedef struct Frame {        /* proc/func call stack frame */
        Symbol  *sp;          /* symbol table entry */
        Inst    *retpc;       /* where to resume after return */
        Datum   *argn;        /* n-th argument on stack */
        int     nargs;        /* number of arguments */
} Frame;
#define        NFRAME        100
Frame        frame[NFRAME];
Frame        *fp;               /* frame pointer */

initcode() {
        progp = progbase;
        stackp = stack;
        fp = frame;
        returning = 0;
}

push(d)
        Datum d;
{
        if (stackp >= &stack[NSTACK])
                execerror("stack too deep", (char *)0);
        *stackp++ = d;
}

Datum pop()
{
        if (stackp == stack)
                execerror("stack underflow", (char *)0);
        return *--stackp;
}

constpush()
{
        Datum d;
        d.val = ((Symbol *)*pc++)->u.val;
        push(d);
}

varpush()
{
        Datum d;
        d.sym = (Symbol *)(*pc++);
        push(d);
}

whilecode()
{
        Datum d;
```

```
        Inst *savepc = pc;

        execute(savepc+2);        /* condition */
        d = pop();
        while (d.val) {
                execute(*((Inst **)(savepc)));        /* body */
                if (returning)
                        break;
                execute(savepc+2);        /* condition */
                d = pop();
        }
        if (!returning)
                pc = *((Inst **)(savepc+1)); /* next stmt */
}

ifcode()
{
        Datum d;
        Inst *savepc = pc;        /* then part */

        execute(savepc+3);        /* condition */
        d = pop();
        if (d.val)
                execute(*((Inst **)(savepc)));
        else if (*((Inst **)(savepc+1))) /* else part? */
                execute(*((Inst **)(savepc+1)));
        if (!returning)
                pc = *((Inst **)(savepc+2)); /* next stmt */
}

define(sp)        /* put func/proc in symbol table */
        Symbol *sp;
{
        sp->u.defn = (Inst)progbase;        /* start of code */
        progbase = progp;        /* next code starts here */
}

call()                /* call a function */
{
        Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
                                      /* for function */
        if (fp++ >= &frame[NFRAME-1])
                execerror(sp->name, "call nested too deeply");
        fp->sp = sp;
        fp->nargs = (int)pc[1];
        fp->retpc = pc + 2;
        fp->argn = stackp - 1;        /* last argument */
        execute(sp->u.defn);
        returning = 0;
}

ret()                /* common return from func or proc */
{
        int i;
```

```
        for (i = 0; i < fp->nargs; i++)
                pop();          /* pop arguments */
        pc = (Inst *)fp->retpc;
        --fp;
        returning = 1;
}

funcret()         /* return from a function */
{
        Datum d;
        if (fp->sp->type == PROCEDURE)
                execerror(fp->sp->name, "(proc) returns value");
        d = pop();          /* preserve function return value */
        ret();
        push(d);
}

procret()         /* return from a procedure */
{
        if (fp->sp->type == FUNCTION)
                execerror(fp->sp->name,
                        "(func) returns no value");
        ret();
}

double *getarg()          /* return pointer to argument */
{
        int nargs = (int) *pc++;
        if (nargs > fp->nargs)
            execerror(fp->sp->name, "not enough arguments");
        return &fp->argn[nargs - fp->nargs].val;
}

arg()         /* push argument onto stack */
{
        Datum d;
        d.val = *getarg();
        push(d);
}

argassign()         /* store top of stack in argument */
{
        Datum d;
        d = pop();
        push(d);          /* leave value on stack */
        *getarg() = d.val;
}

bltin()
{

        Datum d;
        d = pop();
        d.val = (*(double (*)())*pc++)(d.val);
```

```
        push(d);
}

eval()                  /* evaluate variable on stack */
{
        Datum d;
        d = pop();
        if (d.sym->type != VAR && d.sym->type != UNDEF)
                execerror("attempt to evaluate non-variable", d.sym->name);
        if (d.sym->type == UNDEF)
                execerror("undefined variable", d.sym->name);
        d.val = d.sym->u.val;
        push(d);
}

add()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val += d2.val;
        push(d1);
}

sub()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val -= d2.val;
        push(d1);
}

mul()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val *= d2.val;
        push(d1);
}

Div()
{
        Datum d1, d2;
        d2 = pop();
        if (d2.val == 0.0)
                execerror("division by zero", (char *)0);
        d1 = pop();
        d1.val /= d2.val;
        push(d1);
}

negate()
```

```
{
        Datum d;
        d = pop();
        d.val = -d.val;
        push(d);
}

gt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val > d2.val);
        push(d1);
}

lt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val < d2.val);
        push(d1);
}

ge()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val >= d2.val);
        push(d1);
}

le()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val <= d2.val);
        push(d1);
}

eq()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val == d2.val);
        push(d1);
}

ne()
{
```

```
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != d2.val);
        push(d1);
}

and()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
        push(d1);
}

or()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
        push(d1);
}

not()
{
        Datum d;
        d = pop();
        d.val = (double)(d.val == 0.0);
        push(d);
}

power()
{
        Datum d1, d2;
        extern double Pow();
        d2 = pop();
        d1 = pop();
        d1.val = Pow(d1.val, d2.val);
        push(d1);
}

assign()
{
        Datum d1, d2;
        d1 = pop();
        d2 = pop();
        if (d1.sym->type != VAR && d1.sym->type != UNDEF)
                execerror("assignment to non-variable",
                        d1.sym->name);
        d1.sym->u.val = d2.val;
        d1.sym->type = VAR;
```

```
        push(d2);
}

print()        /* pop top value from stack, print it */
{
        Datum d;
        d = pop();
        printf("\t%.8g\n", d.val);
}

prexpr()        /* print numeric value */
{
        Datum d;
        d = pop();
        printf("%.8g ", d.val);
}

prstr()                /* print string value */
{
        printf("%s", (char *) *pc++);
}

varread()        /* read into variable */
{
        Datum d;
        extern FILE *fin;
        Symbol *var = (Symbol *) *pc++;
  Again:
        switch (fscanf(fin, "%lf", &var->u.val)) {
        case EOF:
                if (moreinput())
                        goto Again;
                d.val = var->u.val = 0.0;
                break;
        case 0:
                execerror("non-number read into", var->name);
                break;
        default:
                d.val = 1.0;
                break;
        }
        var->type = VAR;
        push(d);
}

Inst *code(f)        /* install one instruction or operand */
        Inst f;
{
        Inst *oprogp = progp;
        if (progp >= &prog[NPROG])
                execerror("program too big", (char *)0);
        *progp++ = f;
        return oprogp;
}
```

```
execute(p)
        Inst *p;
{
        for (pc = p; *pc != STOP && !returning; )
                (*((++pc)[-1]))();
}

*****         init.c   ************************************************

#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double        Log(), Log10(), Sqrt(), Exp(), integer();

static struct {                 /* Keywords */
        char        *name;
        int         kval;
} keywords[] = {
        "proc",         PROC,
        "func",         FUNC,
        "return",       RETURN,
        "if",           IF,
        "else",         ELSE,
        "while",        WHILE,
        "print",        PRINT,
        "read",         READ,
        0,              0,
};

static struct {                 /* Constants */
        char *name;
        double cval;
} consts[] = {
        "PI",     3.14159265358979323846,
        "E",      2.71828182845904523536,
        "GAMMA",  0.57721566490153286060,  /* Euler */
        "DEG",   57.29577951308232087680,  /* deg/radian */
        "PHI",    1.61803398874989484820,  /* golden ratio */
        0,        0
};

static struct {                 /* Built-ins */
        char *name;
        double        (*func)();
} builtins[] = {
        "sin",          sin,
        "cos",          cos,
        "atan",         atan,
        "log",          Log,         /* checks range */
        "log10",        Log10,       /* checks range */
        "exp",          Exp,         /* checks range */
        "sqrt",         Sqrt,       /* checks range */
```

```
        "int",          integer,
        "abs",          fabs,
        0,              0
};

init()        /* install constants and built-ins in table */
{
        int i;
        Symbol *s;
        for (i = 0; keywords[i].name; i++)
                install(keywords[i].name, keywords[i].kval, 0.0);
        for (i = 0; consts[i].name; i++)
                install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
                s = install(builtins[i].name, BLTIN, 0.0);
                s->u.ptr = builtins[i].func;
        }
}

*****         math.c   ************************************************

#include <math.h>
#include <errno.h>
extern   int        errno;
double   errcheck();

double Log(x)
        double x;
{
        return errcheck(log(x), "log");
}
double Log10(x)
        double x;
{
        return errcheck(log10(x), "log10");
}

double Sqrt(x)
        double x;
{
        return errcheck(sqrt(x), "sqrt");
}

double Exp(x)
        double x;
{
        return errcheck(exp(x), "exp");
}

double Pow(x, y)
        double x, y;
{
        return errcheck(pow(x,y), "exponentiation");
}
```

```
double integer(x)
        double x;
{
        return (double)(long)x;
}

double errcheck(d, s)        /* check result of library call */
        double d;
        char *s;
{
        if (errno == EDOM) {
                errno = 0;
                execerror(s, "argument out of domain");
        } else if (errno == ERANGE) {
                errno = 0;
                execerror(s, "result out of range");
        }
        return d;
}

*****         makefile   ************************************************

CC = lcc
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc6:        $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -lm -o hoc6

hoc.o code.o init.o symbol.o:        hoc.h

code.o init.o symbol.o:        x.tab.h

x.tab.h:        y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr:        hoc.y hoc.h code.c init.c math.c symbol.c
        @pr $?
        @touch pr

clean:
        rm -f $(OBJS) [xy].tab.[ch]
```

Алфавитный указатель

Специальные символы

Числа

A

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

Y

Г

Д

З

И

К

Л

М

Н

О

П

P

С

T

У

Ф

Х

Ц

Ч

Ш

Э

Я