UNIX — универсальная среда программирования

Керниган Б. В., **Пайк Р.**

1992

К36 : Пер. с англ.;
Предисл. М. И. Белякова. - М.: Финансы и статистика, 1992. - 304 с.: ил.

ISBN 5-279-00253-4.

В книге американских авторов — разработчиков операционной системы UNIX — блестяще решена проблема автоматизации деятельности программиста, системной поддержки его творчества, выходящей за рамки языков программирования. Профессионалам открыт богатый “встроенный” арсенал системы UNIX. Многочисленными примерами иллюстрировано использование языка управления заданиями `shell`. Для программистов–пользователей операционной системы UNIX.

ISBN 0-13-937699-2 (США)

ISBN 5-279-00253-4 (СССР)

Предисловия

Предисловие к русскому изданию

Операционная система UNIX пользуется в нашей стране заслуженной известностью. Начиная с 1985 г. вышло порядка 10 книг, рассказывающих об этой системе или ее отечественных аналогах. Специалисты в области вычислительной техники и программисты в других отраслях давно уже применяют систему в своей профессиональной деятельности. Развиваются советские операционные системы ДЕМОС, ИНМОС, МОС и другие, представляющие собой различные вариации на тему UNIX.

Жизнь UNIX оказалась большой и насыщенной событиями. Появившись 20 лет тому назад как инструментальная среда для профессиональных программистов, система благодаря своему изяществу быстро приобрела поклонников и стала использоваться широко. Одно из основных свойств UNIX — мобильность — позволило переносить ее на компьютеры с различной архитектурой, что существенно повысило интерес к ней. Нельзя сказать, что шествие UNIX было триумфальным, поскольку она оказалась не свободной от недостатков (слабость межпроцессного взаимодействия, низкая надежность, недружественность человеко–машинного интерфейса, неразвитость прикладного багажа и др.) и обладала не слишком выдающимися техническими характеристиками, но всем этим пришлось поступиться ради концептуального изящества, универсальности и мобильности. Хотя система и была реализована на большинстве 16-разрядных компьютеров, коммерческого распространения она там не получила и, как правило, побивалась любой операционной системой, создававшейся специально для данной архитектуры. Постановка на 16–разрядные компьютеры скорее была данью моде и рекламе, чем серьезной коммерции.

Начавшаяся эпоха 32–разрядных микропроцессоров постепенно подтверждает прогнозы о том, что родным для UNIX станет именно этот класс компьютеров. Неважные технические характеристики системы компенсируются высокими техническими параметрами компьютера, а мобильность делает ее перенос более предпочтительным, чем разработка новой операционной системы для 32–разрядной архитектуры. В еще большей степени система оказалась пригодной для активно развиваемых в последнее время компьютеров с RISC–архитектурой.

Утвердившись на 32–разрядных компьютерах, UNIX, по-видимому, вступила в период зрелости. “Изюминки” системы стали классикой системного программирования. Ведущие компьютерные компании, имеющие устойчивую репутацию производителей собственных операционных систем, делают серьезную ставку на UNIX для своих 32-разрядных компьютеров. Ассоциации UNIX International и Open Software Foundation под эгидой соответствующих фирм (AT&T и IBM) усиливают борьбу за лидерство в сфере UNIX. Рабочие группы института IEEE под эгидой международной организации стандартов ISO разрабатывают стандарты на интерфейсы мобильной операционной среды, беря за

основу интерфейсы UNIX.

Книга Б. В. Кернигана и Р. Пайка — это не просто еще одно издание по модной системе. Написанная превосходными специалистами, один из которых (Б. В. Керниган) стоял у истоков UNIX и языка Си, книга заметно выделяется в общем потоке литературы по UNIX, в том числе и переведенной у нас. При сравнительно небольшом объеме рассматриваются не только общие вопросы строения, функционирования и использования системы, программирования основных операций, но и инструментарий обработки текстов, а также средства создания программ. Соотношение глубины изложения и широты охвата материала выбрано близким к оптимальному, и у читателя по прочтении очередной главы не возникает ощущения, что он либо получил весьма поверхностные сведения, либо, наоборот, чрезмерно углубился в нечто не столь существенное.

Книгу могут изучать начинающие, не знакомые с системой программисты. Таким читателям адресована первая глава, в общих чертах описывающая круг основополагающих идей UNIX. Затем последовательно рассматриваются свойства системы на разных уровнях интерфейса пользователя (программы), начиная от командного языка и кончая примитивами нижнего уровня (и даже внутренними структурами данных системы). Изложение сопровождается большим количеством примеров, иллюстрирующих как вопросы частного характера, так и технологию решения типовых задач, в частности создание программы с помощью инструментальных средств построения программ. Подробно разбираются файловая система, командный интерпретатор `shell`, методология программирования на командном языке н применения фильтров, стандартные средства буферизованного ввода-вывода и ввод-вывод нижнего уровня, генераторы лексических и синтаксических анализаторов и построитель программ `make`, форматирование текстов В результате читатель получает цельное представление о возможностях системы и методах их использования для решения практических задач программирования.

Несомненно, педагогическое мастерство авторов и хорошо продуманные примеры позволяют рекомендовать эту книгу всем, кто хочет основательно изучить одну из самых знаменитых операционных систем нашего времени.

М. И. Беляков

Предисловие

Число работающих систем UNIX достигло уже 10,
и в дальнейшем возрастет еще больше.
(Справочное руководство по системе UNIX*, 2-е изд., июнь, 1972)*

Операционная система UNIX начала свой жизненный путь на "заброшенной" машине DEC PDP-7 BELL Laboratories в 1969г. (Система UNIX представляет собой продукт BELL Laboratories. Слово "UNIX" не акроним: оно перекликается со словом "MULTICS" — названием операционной системы, с которой работали К. Томпсон и Д. Ритчи ранее.) К. Томпсон при поддержке Р. Канадея, Д. МакИлроя, Д. Осанна и Д. Ритчи написал небольшую систему разделения времени общего назначения, оказавшуюся достаточно удобной, что привлекло энтузиастов–пользователей и послужило основанием для приобретения более мощной машины — PDP-11/20. Одним из первых пользователей системы считается Д. Ритчи, который помог в 1970 г. перенести ее на PDP-11. Д. Ритчи также разработал и написал компилятор с языка программирования Си. В 1973 г. Д. Ритчи и К. Томпсон переписали ядро системы UNIX на языке Си, отойдя от традиции создания таких программ на языке Ассемблера. В этом последнем варианте система в основном сохранилась и по сей день.

В 1974 г. UNIX была передана университетам "для образовательных целей", а несколько лет спустя нашла и коммерческое применение. В течение всего этого периода система UNIX продолжала совершенствоваться в BELL Laboratories, получив признание в лабораториях, при создании программного обеспечения, в центрах обработки информации, при поддержке операций телефонных компаний. В настоящее время диапазон ее использования весьма широк — от микрокомпьютеров до самых больших универсальных машин.

В чем причины успеха системы UNIX? Их несколько. Во-первых, поскольку UNIX написана на языке Си, она переносима и, как уже отмечалось, имеет широкий диапазон применения. Во-вторых, доступны исходные тексты программ системы, написанные на языке высокого уровня, что позволяет легко адаптировать ее в соответствии с конкретными требованиями. Наконец, самое главное — это хорошая операционная система, особенно для программистов. Программная среда UNIX необычайно богата и результативна.

Вместе с UNIX появилось много новых программ и методов программирования, но никакая отдельная программа или идея не дает гарантии в отношении качества системы. Эффективность системы достигается благодаря определенному подходу к программированию, своего рода философии использования вычислительной машины. Основной смысл ее состоит в том, что мощь системы обусловливается взаимодействием программ, а не мощью самих программ. Многие программы UNIX могут решать простейшие задачи, но при объединении с другими программами они превращаются в универсальные и полезные средства. Поэтому в своей книге мы уделяем большое внимание вопросам, связанным с взаимодействием программ и с нх использованием в качестве инструментария для создания других программ. Чтобы успешно работать с UNIX, необходимо не только знать, как применять ту или иную программу, но и понимать ее роль в системе.

По мере распространения UNIX число специалистов, хорошо владеющих системой, к сожалению, уменьшается. Помочь здесь может только время (когда-то и опытные пользователи, включая авторов, могли найти лишь "неуклюжее" решение задачи или написать программы для выполнения задач, с которыми вполне можно справиться уже существующими средствами). Конечно, найти элегантные решения без определенного опыта и знаний нелегко. Мы надеемся, что, изучив материал книги, вы достигнете такой

степени компетентности, когда работа в системе, независимо от того, новичок вы или профессионал, становится эффективной и доставляет удовольствие.

Если наша книга поможет программистам-индивидуалам сделать свой труд более производительным, то тем самым повысится производительность и больших коллективов. Хотя книга в основном рассчитана на программистов, для понимания первых четырех или даже пяти глав программистский опыт не требуется, так что они могут оказаться полезными всем без исключения. Всюду, где было возможно, в качестве примеров мы приводили настоящие программы. Многие из них уже вошли в наш повседневный программный обиход. Все примеры проверялись на машине прямо из текста, который представлен в форме, пригодной для ввода.

Книга построена следующим образом. В гл. 1 излагаются основные возможности UNIX (вход в систему, почта, система файлов), рассматриваются наиболее употребительные команды и даются начальные сведения об интерпретаторе команд. Опытные пользователи могут опустить эту главу.

В гл. 2 обсуждается файловая система UNIX. Файловая система является центральным звеном в обеспечении успешной работы, и ее нужно хорошо понимать. Здесь описываются файлы и каталоги, права доступа и режимы файлов, а также индексные дескрипторы. Приводятся краткое описание иерархии файловой системы и пояснения относительно файлов устройств.

Интерпретатору команд (`shell`), этому основополагающему средству не только для выполнения программ, но и для их написания, посвящена гл. 3. Вы узнаете, как использовать `shell` для своих целей (новые команды, аргументы команд, переменные `shell`, простые структуры управления и переключение ввода-вывода).

Глава 4 познакомит вас с фильтрами, т. е. программами, которые производят какие-то простые преобразования данных по мере их получения. В первой части главы рассматривается команда контекстного поиска `grep` и родственные ей команды. Затем обсуждаются более общие фильтры, такие, как `sort`. В заключение описываются две программы преобразования данных общего назначения `sed` и `awk`. Потоковый редактор `sed` редактирует поток данных по мере их получения, `awk` — язык программирования для простых операций поиска информации и задач генерации отчетов. Эти программы, иногда в сочетании с интерпретатором `shell`, позволяют в ряде ситуаций обойтись без традиционных языков программирования.

В гл. 5 обсуждается, как с помощью языка `shell` можно создавать программы, которые будут использоваться другими людьми. Тема включает более развитые структуры управления и переменные, перехват и обработку прерываний. Приводятся примеры с привлечением программ `sed` и `awk`, а также интерпретатора `shell`.

О том, как написать новые программы, используя стандартную библиотеку ввода-вывода, рассказывается в гл. 6. Программы написаны на языке Си, который предполагается известным, или по крайней мере должен быть изучен параллельно. Мы пытаемся показать разумную стратегию проектирования и организации новых программ: как создавать их путем осмысленных приближений и как применять уже существующие средства.

Далее (гл. 7) мы переходим к системным обращениям, являющимся фундаментом для всех остальных слоев программного обеспечения. Здесь обсуждаются вопросы ввода-вывода, создания файла, обработки ошибки, описываются каталоги, индексные дескрипторы, процессы и сигналы.

В гл. 8 рассматриваются развитые программные средства: программа `yacc`, создающая программы грамматического разбора; `make`, управляющая процессом трансляции больших программ; `lex`, создающая лексические анализаторы. Изложение строится на

примере создания большой программы — программного калькулятора в стиле языка Си.

В гл. 9 вашему вниманию предлагаются средства подготовки документации, которые сопровождаются описаниями на уровне пользователя и страницей руководства по калькулятору из гл. 8. Данную главу можно читать независимо от остальных глав.

В конце книги приводятся приложения. В приложении 1 суммируются возможности стандартного редактора `ed`. Хотя многие читатели предпочтут для повседневного пользования иной редактор, `ed` представляется общедоступным, действенным и эффективным средством. Регулярные выражения редактора являются ключевым понятием других программ, таких, как `grep` и `sed`, и даже только по этой причине его стоит изучить. Приложение 2 содержит справочное руководство по языку калькулятора из гл. 8, а приложение 3 — распечатку окончательной версии программы калькулятора, где для удобства чтения собраны воедино все программные фрагменты.

Некоторые практические вопросы. Во-первых, система UNIX стала очень популярной, и несколько ее версий нашли широкое применение. Например, седьмая версия происходит от первоначальной системы UNIX вычислительного и научно-исследовательского центра фирмы BELL. System III и System V — версии, официально поддерживаемые фирмой. Университет в Беркли (штат Калифорния) распространяет системы UCB 4.xBSD, производные от шестой версии. Помимо названных существуют многочисленные версии, особенно на малых машинах, которые созданы на базе седьмой версии. Мы пытались справиться с этим многообразием, стараясь максимально придерживаться каких-то общих концепций, одинаковых для всех систем. Для большей точности мы выбрали представление, базирующееся на седьмой версии, поскольку она является основой почти всех широко используемых систем UNIX. Программы прогонялись на System V фирмы BELL и на 4.1BSD из Беркли, причем потребовались лишь тривиальные изменения в нескольких примерах. Независимо от вашей версии системы найденные вами различия будут минимальны.

Далее, несмотря на обширность материала книги, есть вопросы, которых мы не касались. Основным источником информации для вас является справочное руководство по системе UNIX. Вы можете обращаться к нему, чтобы разрешить свои сомнения или определить, насколько ваша система отличается от нашей.

Наконец, мы убеждены в том, что лучший способ изучения чего-либо — это эксперимент. Книгу следует читать за терминалом, находя подтверждение либо опровергая прочитанное, исследуя ограничения и варианты. Читайте, проверяйте и снова читайте. Система UNIX — хотя и не идеальный, но удивительный программный мир, и мы надеемся, что вы откроете его для себя.

Всем, кто так или иначе помогал нам в процессе работы над книгой, мы выражаем искреннюю признательность.

Б. В. Керниган,
Р. Пайк

Глава 1

UNIX для начинающих

Что такое UNIX? В узком смысле слова это ядро операционной системы разделения времени, т. е. программа, которая распоряжается ресурсами вычислительной машины и предоставляет их пользователям. Она дает пользователям возможность запускать свои программы, управляет периферийными устройствами машины (дисками, терминалами, печатающими устройствами и т. п.) и обеспечивает работу файловой системы, предназначенной для длительного хранения информации, в частности программ, данных или документов.

В более широком смысле под UNIX часто понимают не только ядро, но и такие важные компоненты, как компиляторы, редакторы, программы копирования и печати файлов. Сюда даже относят языки управления заданиями (командные языки). UNIX может также включать созданные вами или другими пользователями программы, работающие в вашей системе и предназначенные для подготовки документации, статистического анализа, или, скажем, "графические" средства. Значение слова "UNIX" в каждом конкретном случае зависит от того, на каком уровне вы рассматриваете систему. В дальнейшем изложении смысл этого термина вам будет ясен из контекста.

Иногда система UNIX кажется более сложной, чем она есть на самом деле. Новичку нелегко понять, как наилучшим образом воспользоваться доступными средствами. Но, к счастью, начинать работать в системе не так уж и трудно, а познакомившись с несколькими программами, вы получите достаточное стартовое ускорение. Цель настоящей главы помочь вам по возможности скорее освоить систему. Это не справочное руководство, а лишь краткий обзор приводимого в книге материала, причем многие вопросы будут рассмотрены более детально в последующих главах. Мы обсудим здесь следующие темы:

- основные понятия: вход и выход из системы, простые команды, исправление ошибок при вводе с терминала, связь между терминалами;
- повседневная работа: файлы и файловая система, печать файлов, каталоги, часто используемые команды;
- интерпретатор команд shell: сокращенная форма имен файлов, переключение ввода–вывода, программные каналы, установка символов удаления и стирания, задание вашего собственного пути поиска команд.

Если вы ранее работали в системе UNIX и большая часть материала главы покажется вам знакомой, то можете перейти сразу к гл. 2. Но, даже прочитав эту главу, вы будете вынуждены обращаться к справочному руководству по системе, поскольку данная книги

не заменяет его, а лишь дает рекомендации по применению описанных в нем команд. Более того, вы можете обнаружить расхождение в тексте книги и описании вашей системы. В начале справочного руководства приводится предметный указатель (индекс), который поможет вам ориентироваться в программах. Научитесь им пользоваться.

В заключение хотелось бы дать вам совет: не бойтесь экспериментировать. Если вы новичок, то существует очень мало ситуаций, в которых вы можете повредить себе или другим пользователям, так что изучайте свою систему методом "проб и ошибок". Первая глава довольно большая, и лучший способ ознакомиться с ней читать за один прием всего несколько страниц, постоянно экспериментируя.

1.1 Итак, приступаем

Некоторые предварительные замечания о терминалах и вводе символов. Мы не собираемся здесь затрагивать вопросы, связанные с применением вычислительных машин, поскольку предполагается, что вы имеете представление о терминалах и о том, как ими пользоваться. Если по ходу изложения вам что-нибудь покажется непонятным, обратитесь за помощью к местному специалисту.

Система UNIX полностью дуплексная: символы, которые вы вводите с терминала, попадают в систему, а она посылает их назад на терминал, с тем чтобы они появились на экране. Обычно такой процесс подобен эху: символы просто копируются на экран, и вы можете видеть, что вводите, но иногда, например при вводе секретного пароля, эхо пропадает, и символы не появляются на экране.

Большинство символов на клавиатуре не имеет специального назначения, но есть символы, подсказывающие машине, как интерпретировать то, что вы вводите. Наиболее важным из них является символ, вводимый при нажатии клавиши *RETURN*. Он обозначает конец вводимой строки: в ответ на его ввод система передвигает курсор на начало следующей строки на экране. Прежде чем система начнет интерпретировать введенные символы, вы должны нажать клавишу *RETURN*.

RETURN это пример управляющего символа, т. е. невидимого символа, определяющего некоторые аспекты действий, выполняемых при вводе и выводе на терминал. На обычном терминале символ *RETURN* вводится с помощью отдельной клавиши, хотя большинство управляющих символов не связано с "персональными" клавишами. Для их ввода требуется нажать клавишу *CONTROL*, иногда обозначаемую как *CTRL*, *CTL* или *CNTL*, одновременно нажав еще одну клавишу, обычно соответствующую букве. Например, конец строки можно получить, нажав клавишу *RETURN*, либо (эквивалентный способ) при нажатой клавише *CONTROL* ввести символ 'm' Поэтому *RETURN* можно называть "управляющее m". или *ctl-m*. Среди других управляющих символов необходимо выделить *ctl-d*, сообщающий программе, что ввод окончен: "*ctl-g*", вызывающий звонок на терминале: *ctl-h*, который часто называется "шаг назад" и может использоваться при коррекции ошибок ввода, и *ctl-i* (или *tab*), перемещающий курсор на следующую позицию табуляции почти так же, как и на стандартном телетайпе. Позиции табуляции в системе UNIX отстоят друг от друга на восемь пробелов. На большинстве терминалов предусмотрены специальные клавиши для символов "шаг назад" и *tab*.

Существуют еще две особые клавиши: *DELETE* (*DEL*), иногда обозначаемая как *RUBOUT* (могут использоваться разные сокращения), и *BREAK* или *INTERRUPT*. В большинстве систем UNIX ввод символа *DELETE* немедленно останавливает программу, даже если она еще не завершилась. В некоторых системах для этого употребляется

символ *ctl-c*. В ряде систем в зависимости от способа подключения терминала синонимом *DELETE* или *ctl-c* может служить *BREAK*.

Сеанс работы с UNIX. Начнем с диалога между вами и системой UNIX. Установить связь: позвонить по телефону или включить питание, если необходимо. Ваша система должна ответить

```
login: you          Введите ваше имя, затем нажмите RETURN
Password:           Ваш пароль, если вы его вводите,
                    не появится на экране

You have mail.      Есть почта, ее можно прочесть после входа в систему

$                   Система готова к приему ваших команд
$                   Можно нажать RETURN несколько раз
$ date              Узнаем дату и время
Sun Sep 25 23: 02: 57 EDT 1983
$ who               Узнаем,кто работает на машине?
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              Прочтем почту
From doug Sun Sep 25 20:53 EDT 1983
зайди ко мне когда-нибудь в понедельник
?                   При нажатии RETURN выдается следующее сообщение
From mary Sun Sep 25 19:07 EDT 1983
Пообедаем завтра в полдень?
? d                 Удалим это сообщение
$                   Почты больше нет
$ mail mary         Пошлем почту mary
обед в 12
ctl-d               Конец сообщения
$                   Позвоним по телефону или выключим терминал
                    Вот и все
```

Иногда к такому диалогу сводится вся ваша работа с системой, однако бывает, что при этом и "дело делается". Далее мы разберем приведенный пример сеанса, а также некоторые полезные программы.

Вход в систему. Вы должны иметь входное имя и пароль, которые можно получить от администратора системы. Система UNIX может работать с самыми разными терминалами, но в основном она ориентирована на устройства со строчными буквами. Разница между строчными и прописными буквами существенна! Если на вашем терминале есть только прописные буквы (как на некоторых портативных терминалах), это настолько усложнит вам жизнь, что вскоре вы начнете искать другой терминал.

Перед началом работы проверьте, все ли переключатели на терминале установлены в правильное положение: верхний или нижний регистр, дуплексная ли связь и т. д. с

учетом рекомендаций местного специалиста, например в отношении скорости передачи. Затем, используя любые "заключения", установите связь с системой: иногда для этого достаточно лишь позвонить по телефону или просто включить тумблер. В любом случае система должна выдать:

```
login:
```

Если она выдала чепуху, то, может быть, у вас установлена не та скорость передачи: проверьте положение соответствующего (а заодно и других) переключателя. Когда это не помогает, нужно медленно несколько раз нажать клавишу *BREAK* или *INTERRUPT*. Если приглашение `login:` и теперь не появляется, вам следует обратиться за помощью к специалисту.

Получив приглашение на вход, введите ваше входное имя строчными буквами, после чего нажмите клавишу *RETURN*. При необходимости система запросит у вас пароль, и пока вы вводите его, выдача на терминал будет отключена. В конце концов, войдя в систему, вы получите приглашение (как правило, один символ, означающий, что система готова принимать ваши команды). Приглашением, вероятнее всего, окажется символ $ (знак доллара) или % (процент), но можно заменить его любым другим; позднее мы покажем, как это сделать. Приглашение обычно выдается программой, называемой интерпретатором команд, или `shell`, с помощью которой вы общаетесь с системой. Перед приглашением на экране может появиться сообщение о том, что для вас есть почта, или вопрос о типе применяемого терминала. Ваш ответ поможет системе воспользоваться какими–либо особыми свойствами данного терминала.

Ввод команд. Получив приглашение, вы можете ввести команды, т. е. требования к системе что-либо сделать. Мы часто будем употреблять слово программа как синоним команды. Когда на экране появится приглашение (пусть это будет символ '$'), введите `date` и нажмите клавишу *RETURN*. Система в ответ выдаст дату и время и выведет следующее приглашение, так что весь диалог на вашем терминале примет вид:

```
$ date
Mon Sep 26 12 : 20 : 57 EDT 1983
$
```

Не забывайте нажимать клавишу *RETURN* и не вводите символ '$'. Если вам покажется, что о вас забыли, нажмите *RETURN*, возможно, что-нибудь и произойдет. В дальнейшем мы не будем напоминать вам о том, что символ *RETURN* должен завершать каждую строку.

Затем попробуйте ввести команду `who`, которая сообщает, кто в данный момент работает в системе:

```
$ who
rim     tty0     Sep 26 11:37
pjw     tty4     Sep 26 11:30
gerard  tty7     Sep 26 10:27'
mark    tty9     Sep 26 07:59
you     ttya     Sep 26 12:20
$
```

В первом столбце указывается имя пользователя, во втором - системное имя используемого устройства связи (`tty` — сокращение от teletype, “архаическое” обозначение терминала). В третьем столбце отмечаются дата и время входа в систему. Вы можете поэкспериментировать и с такой командой:

```
$ who am i
you tty a Sep 26 12 : 20
$
```

Если вы ошибетесь и укажете при вводе несуществующую команду, система ответит, что команда с этим именем не найдена:

```
$ whom                  Опечатка в имени команды...
whom:not found          поэтому система не знает, как запустить ее
$
```

Конечно, если вы неумышленно введете имя существующей команды, то она начнет выполняться и, возможно, даст неожиданный результат.

Странное поведение терминала. Иногда ваш терминал может повести себя странно, например, каждая буква будет выдаваться дважды, или при нажатии клавиши *RETURN* курсор не переместится в первую позицию следующей строки. Обычно это можно устранить, выключив и включив терминал или выйдя из системы и вновь войдя в нее. Можно также прочитать описание команды `stty` (“set terminal options” — установка режима терминала) в разделе 1 справочного руководства. Для разумной интерпретации символа tab, если на вашем терминале не предусмотрен режим автоматической табуляции, введите команду

```
$ stty -tabs
```

и система будет преобразовывать символ *tab* в определенное число пробелов. Если же на терминале допускается возможность установки позиций табуляции от машины, то команда `tabs` поставит их должным образом. (Вы можете дать и такую команду:

```
$ tabs terminal-type
```

см. описание команды `tabs` в справочном руководстве.)

Ошибки при вводе. Если вы ошиблись при вводе и заметили это прежде, чем нажали клавишу *RETURN*, то поправить дело можно двумя способами: или стирать символы (но одному), или уничтожить всю строку и ввести ее заново. При нажатии клавиши, служащей для уничтожения строки (обычно это символ ‘@’), вся строка будет удалена, как будто бы вы ее никогда и не вводили, а вам придется выполнить ввод с начала новой строки:

```
$ ddate                                  Уничтожается вся строка:
date                                     ввод с новой строки
Mon Sep 26 12 : 23 : 39 EDT 1983
$
```

Символ '#' стирает последний введенный символ: каждый раз при вводе # стирается только один символ слева от курсора. Поэтому если вы не уверены в своем вводе, можно делать поправки по мере работы:

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

Представление символов стирания (#) и уничтожения (@) зависит от системы. Во многих системах (включая используемую здесь для примеров) в качестве символа стирания употребляется "шаг назад", что хорошо смотрится на видеотерминалах. С помощью своей системы вы можете легко проверить, так ли это:

```
$ datee<-                                Пробуем <-
datee<- : not found                      Не подходит
$ datee#                                 Пробуем #
Mon Sep 26 12 : 26 : 08 EDT 1983         # подходит
$
```

(Мы изображаем "шаг назад" как <-, чтобы символ был "виден".) Обычно принимается, что символ *ctl-u* уничтожает всю строку.

Далее в качестве символа стирания мы будем использовать #, по-скольку он "виден" на терминале. Если в вашей системе его нет, то сделайте соответствующие изменения. Ниже при описании настройки окружения мы покажем, как задать удобное для вас представление символов стирания и уничтожения.

А что нужно, чтобы ввести сами символы стирания и уничтожения как часть текста? Если перед символами # и @ поставить обратную дробную черту (\), то они утрачивают свое специальное назначение. Поэтому для ввода # или @ необходимо набрать на клавиатуре \# или \@ . При вводе @ система может передвинуть курсор в начало следующей строки, даже если ему предшествовала обратная дробная черта. Не волнуйтесь, символ @ в текст попадет.

Обратную дробную черту иногда называют символом экранирования. Она указывает на то, что следующий за ней символ в некотором смысле специальный. Для удаления обратной дробной черты необходимо ввести два символа стирания: ##. Понятно почему?

Вводимые вами символы, прежде чем попадут к адресату, анализируются и интерпретируются последовательным рядом программ. Поэтому их интерпретация зависит не только от программы, в которую они попали, но и от того, откуда они туда попали.

Каждый вводимый символ немедленно, как эхо, появляется на терминале, если, конечно, эхо "не отключено", что бывает редко. До тех пор пока вы не нажали клавишу *RETURN*, все символы временно сохраняются в ядре системы, так что с помощью символов стирания и уничтожения можно исправить ошибки ввода. Если символ стирания

или уничтожения предваряется обратной дробной чертой, то ядро отбрасывает обратную черту и хранит соответствующий символ, не интерпретируя его специальным образом.

При нажатии клавиши *RETURN* хранимые символы пересылаются в программу, читающую сейчас символы с терминала. Считывающая программа в свою очередь может интерпретировать символы особым образом: например, интерпретатор `shell` исключает любую специальную интерпретацию символа, если ему предшествует обратная дробная черта. Мы вернемся к этому вопросу в гл. refsec:3. Пока же следует помнить о том, что ядро системы реагирует лишь на символы уничтожения и стирания и на обратную дробную черту, если она предшествует одному из них. Любые символы, которые остаются после такой обработки, могут интерпретироваться другими программами.

■ Упражнение: Объясните, что произойдет в случае ввода команды

```
$ date @
```

■ Упражнение: Большинство интерпретаторов `shell` (кроме версии 7) интерпретирует символ # как начало примечания и игнорирует весь текст от символа # до конца строки. Учитывая это, объясните приведенный ниже диалог. Предполагается, что для стирания также используется символ #:

```
$ 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. Ядро системы воспринимает ваш ввод по мере того, как вы набираете текст, даже если оно занято чем-то другим, так что можно вводить данные сколь угодно быстро, даже в тот момент, когда какая-либо программа выполняет вывод на терминал. Вводимые в это время символы будут “перемежаться” на экране с выводимыми, но будут сохраняться и интерпретироваться в должном порядке. Вы можете вводить команды одну за другой, не дожидаясь не только конца, но и начала их выполнения.

Остановка программы. Выполнение большинства команд можно остановить, нажав клавишу *DELETE* (УДАЛИТЬ). Чаще всего так же действует и клавиша *BREAK* (ПРЕРВАТЬ), которая имеется почти на всех терминалах. При работе некоторых программ, таких, как текстовые редакторы, *DELETE* прекращает выполнение программы, но без выхода из нее. При выключении терминала или отключении от сети выполнение многих программ завершается.

Если вы хотите приостановить вывод на терминал, например, для того, чтобы важная информация не исчезла с экрана, введите *ctl-s*. Вывод прекратится практически немед-

ленно: выполнение программы приостановится до тех пор, пока вы не возобновите его, для чего достаточно ввести *ctl-q*.

Выход из системы. Для правильного выхода из системы нужно вместо очередной команды ввести *ctl-d*. `Shell` воспримет это как сообщение о конце ввода. (Что произойдет на самом деле, будет объяснено в следующей главе.) Обычно вы можете просто выключить терминал или отключить его от сети, но произойдет ли при этом фактический выход из системы, зависит от самой системы.

Почта. Система предоставляет почтовые услуги для обмена сообщениями с другими пользователями, так что, войдя однажды в систему, вы увидите на экране до появления приглашения сообщение:

```
You have mail
```

Для чтения почты введите:

```
$ mail
```

Сообщения будут выведены одно за другим, начиная с самых последних. После каждого сообщения программа `mail` ожидает вашего указания, что делать с сообщением. Возможны два основных ответа: ввод символа 'd', означающего удаление сообщения, и ввод *RETURN*, оставляющего его (т. е. оно вновь появится при следующем просмотре почты). Другими ответами могут быть 'p', что означает распечатку сообщения, '*s filename*' — сохранение сообщения в поименованном файле и 'q' — выход из программы `mail`. (Если вы не знаете, что такое файл, то представьте его себе как место, где можно хранить информацию под выбранным вами именем, а затем получать ее оттуда. Файлы рассматриваются в разд. 1.2, как, впрочем, и в большей части этой книги.)

`mail` — именно та программа, которая, вероятно, будет отличаться от описываемой здесь, поскольку существует много вариантов такой программы. Более детально вы можете познакомиться с ней по своему справочному руководству.

Послать почту кому-нибудь весьма просто. Допустим, она предназначена для пользователя с входным именем `nico`. Легче всего это сделать так:

```
$ mail nico
Теперь вводите любой текст письма
из любого числа строк...
После последней строки письма
введите ctl-d
ctl-d
$
```

Ввод *ctl-d* означает, что письмо окончено. Если в процессе составления письма вы передумаете и решите его не отправлять, нажмите клавишу *DELETE* вместо *ctl-d*. Незаконченное письмо будет сохранено в файле `dead.letter`.

Для проверки пошлите письмо самому себе, а затем введите `mail`, чтобы прочитать его. (Это не так странно, как может показаться, и представляется удобным механизмом напоминания.) Существуют и иные способы посылки почты: можно послать заранее

подготовленное письмо, направить почту нескольким адресатам одновременно и даже переслать почту пользователям, работающим на других машинах (подробнее об этом см. в описании команды `mail` в разд. 1 справочного руководства по UNIX.) В дальнейшем мы будем применять обозначение `mail(1)` для страницы, описывающей команду `mail` в разд. 1 справочного руководства.

Имеется также служебная программа `calendar` для печати календаря (см. `calendar(1)`); в гл. 4 мы покажем, как создать такую программу, если она отсутствует.

Сообщение для других пользователей. Если ваша система UNIX многопользовательская, то как-нибудь однажды на вашем терминале может появиться сообщение типа

```
Message from mary tty 7...
```

сопровождаемое пугающим жужжанием. Пользователь Mary хочет что-то сообщить вам, но, если вы не совершите определенных действий, то не сможете ей ответить. Поэтому введите

```
$ write mary
```

чтобы установить двустороннюю связь. Теперь вы с Mary сможете обмениваться сообщениями, хотя эта линия связи очень медленная, словно ваш абонент находится на Луне.

У вас может появиться желание во время выполнения программы задать ту или иную команду для `shell`. Обычно, какая бы программа ни выполнялась, она должна быть приостановлена либо остановлена но некоторые программы, такие, как редактор или сама команда `write`, имеют специальную команду 'T' для временного выхода в интерпретатор `shell` (см. табл. 2 приложения 1).

Команда `write` не накладывает никаких ограничений, поэтому необходим протокол общения, чтобы ваш ввод не перемешивался с тем, что вводит Mary. Существует соглашение, согласно которому ввод следует осуществлять порциями, оканчивающимися символами (o), что означает конец ввода ("over"), а для сигнализации о прекращении связи использовать (oo) ("over" и "out" — конец и выход).

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

Выполнение команды `write` также можно прекратить, нажав клавишу *DELETE*. Заметьте, что ваши ошибки при вводе не появляются на терминале у Mary.

Если вы попытаетесь послать сообщение на терминал тому, кто пока еще не вошел в систему или не хочет, чтобы его беспокоили, вас известят об этом. В том случае, когда адресат находится в системе, но не отвечает за разумный промежуток времени (возможно, он занят или отошел от терминала), просто введите *ctl-d* или *DELETE*. Если вы не хотите, чтобы вас беспокоили, используйте команду `mesg(1)`.

Служба новостей. Многие системы UNIX имеют службу новостей, чтобы держать пользователей в курсе интересных и не очень интересных событий. Попробуйте ввести

```
$ news
```

Существует также большая сеть систем UNIX, которые связаны но телефонному каналу: справьтесь у местного специалиста о командах `netnews` или **USENET**.

Справочное руководство. В справочном руководстве по UNIX вы найдете большую часть того, что нужно знать о системе. В разд. 1 включены и те команды, которые мы обсуждали в этой главе. В разд. 2 описываются системные обращения, которые будут темой гл. 7, а в разд. 6 приводится информация об играх. В остальных разделах описываются функции, применяемые пользователями, программирующими на языке Си, а также структура файлов и средства поддержания работоспособности системы. (Число разделов варьируется от системы к системе.) Не забывайте о предметном указателе в начале руководства: вам достаточно бегло просмотреть его, чтобы найти команды, необходимые для вашей работы. Кроме того, существует введение в систему, где дается обзор ее функциональных возможностей. Часто справочное руководство хранится в системе и его можно читать с терминала. Если вам требуется помощь и не к кому обратиться, можно вывести на терминал любую страницу руководства по системе, введя команду `man command-name`. Так, чтобы получить информацию о команде `who`, введите

```
$ man who
```

и, конечно,

```
$ man man
```

выдаст вам сведения о самой команде `man`.

Автоматизированный справочник. В вашей системе может быть команда с именем `learn`, которая реализует автоматизированный справочник по системе файлов, основным командам, редактору, программам подготовки документации и даже языку программирования Си. Введите

```
$ learn
```

Если эта команда существует в вашей системе, то она подскажет вам, что делать дальше. Если же ее нет, попробуйте ввести еще `teach`.

Игры. Они не всегда признаются официально, но это один из лучших способов обрести уверенность в общении с машиной и терминалом. Сама система UNIX имеет достаточно скромный набор игр, но он часто пополняется программистами на местах. Поспрашивайте вокруг или обратитесь к разд. 6 своего справочного руководства.

1.2 Повседневная работа: файлы и основные команды

Информация в системе UNIX хранится в файлах, которые весьма похожи на обычные учрежденческие "дела". Каждый файл имеет имя, содержание, место, где он хранится, и некоторую служебную информацию, в частности о владельце файла, размере последнего и т. д. Файл может содержать письмо, список имен и адресов, операторы текста программы, данные, которые будет использовать программа, или даже программы в форме, готовой к выполнению, а также другую нетекстовую информацию.

Файловая система UNIX устроена таким образом, что вы можете хранить ваши личные файлы, не смешивая их с файлами других пользователей, причем те в свою очередь не смогут повлиять на ваши файлы. Существует бесчисленное множество программ, оперирующих с файлами, но пока мы остановимся на наиболее часто используемых. В гл. 2 дается последовательное изложение файловой системы и вводится много команд, с которыми приходится иметь дело при работе с файлами.

Создание файлов. Редактор. Если вы хотите ввести статью, письмо или программу, как заставить машину хранить информацию? Большинство таких задач решается с помощью текстового редактора, т. е. программы для ввода и обработки информации в машине. Практически в любой системе UNIX есть экранный редактор — редактор, который использует возможности современных терминалов отображать результат редактирования но мере корректирования текста. Наиболее популярны редакторы `vi` и `emacs`. Однако мы не будем описывать здесь какой-либо конкретный экранный редактор частично из-за сложностей, связанных с типографским набором, частично из-за отсутствия стандартного редактора.

Но существует старый редактор `ed`, который, без сомнения, имеется в вашей системе. Его можно использовать на любом терминале. Он содержит основу, на базе которой строятся другие важные программы (включая и некоторые экранные редакторы), поэтому все–таки следует его изучить. Подробное описание `ed` приводится в приложении 1.

Какой бы редактор вы ни предпочли, вам придется изучить его настолько, чтобы с его помощью уметь создавать файлы. Мы будем работать с редактором `ed`, что позволит нам сделать изложение более конкретным, а вам — выполнить приводимые здесь примеры в своей системе, хотя в принципе вы можете использовать любой редактор.

Попытайтесь создать с помощью `ed` файл под именем `junk` следующим образом:

```
$ ed                          Вызов текстового редактора
a                             команда редактора для добавления текста (add)
now type in
whatever text you want...
.                             Ввод только '.' прекращает добавление
w junk                        Запись текста в файл с именем junk
39                            ed сообщает число введенных символов
q                             Выход из ed (quit)
$
```

Команда 'a' (append добавить) сообщает редактору, что нужно принять текст. Сигналом окончания текста служит один символ '.' , который должен быть введен в начале строки. Не забывайте об этом, поскольку пока он не введен, не распознаются никакие

команды редактора, т. е. все, что вы вводите, будет трактоваться как продолжение вводимого текста.

Команда редактора 'w' (write писать) сохранит введенную информацию: 'w junk' запишет ее в файл с именем `junk`. Именем файла может быть любое слово. Мы выбрали `junk`, чтобы показать, что этот файл не очень важен ("junk" — мусор).

Редактор сообщает системе число символов, записанных им в файл. До ввода команды 'w' ничего не отправляется на постоянное хранение, поэтому, если вы отключите свой компьютер от сети и пойдете домой, информация не попадет в файл. (Если вы это сделаете во время редактирования, информация, с которой вы работаете, будет сохранена в файле `ed.hup`, и в дальнейшем можно будет продолжить работу.) В случае аварии системы в процессе редактирования (т. е. неожиданного останова из-за неисправности аппаратуры или ошибок в программном обеспечении) ваш файл сохранит только то, что в него записала последняя команда write. Но после выполнения команды w информация хранится постоянно. Она может стать доступной, если вы введете

```
$ ed junk
```

Конечно, можно редактировать введенный текст, чтобы исправить опечатки, заменить слова, переставить части текста и т. д. Когда вы завершите редактирование, команда 'q' ("quit" — выход) осуществит выход из редактора.

Что за файлы здесь? Чтобы знать, с чем приходится иметь дело, создадим два файла с именами `junk` и `temp`:

```
$ ed
a
To be or not to be
.
w junk
19
q
$ ed
a
What is a question.
.
w temp
22
q
$
```

Число символов, сообщаемое редактором `ed`, включает и специальный символ в конце каждой строки, называемый перевод строки или конец строки, — так система представляет символ *RETURN*.

Команда `ls` перечисляет имена (но не содержание) файлов:

```
$ ls
junk
temp
$
```

Они и являются, действительно, именами двух только что созданных файлов (могут быть также и другие, которые вы не создавали сами). Имена автоматически сортируются в алфавитном порядке.

Как и большинство команд, `ls` имеет возможные аргументы, которые позволяют изменить ее действие. Возможные аргументы следуют за именем команды в командной строке и обычно состоят из знака минус '-' и одной буквы, несущей смысловую нагрузку. Например, команда `ls -t` требует перечисления файлов во временном порядке, т. е. в зависимости от времени последнего изменения файла, причем файлы, измененные последними, перечисляются вначале:

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

Возможный аргумент `-l` означает "длинный" список (long — длинный), который содержит больший объем информации о каждом файле:

```
$ ls -l
total 2
-rw-r--r-- 1 you 17 Sep 26 16:25 junk
-rw-r--r-- 1 you 18 Sep 26 16:26 temp
$
```

Строка total 2 указывает число занятых блоков на диске: блок обычно содержит 512 или 1024 символа. Строка `-rw-r-r-` показывает, кто имеет право читать из файла и писать в него: в данном случае владелец (you) может и читать, и писать, но другие могут только читать. За ней следует 1 — число связей файла; забудем о нем до гл. 2. Строка you содержит имя владельца файла, т. е. пользователя, создавшего его. Число символов в соответствующих файлах (17 и 18) совпадает с тем, что сообщил редактор `ed`. Дата и время соответствуют последнему изменению файла.

Возможные аргументы (в дальнейшем будем именовать их флагами) могут быть сгруппированы: `ls -lt` дает ту же информацию, но отсортированную в определенном порядке, начиная с файлов, измененных последними. Флаг `-r` показывает дату и время последнего обращения к файлу; `ls -lut` представляет список файлов по порядку их использования, начиная с наиболее позднего. Флаг `-r` меняет порядок в списке на обратный, так что `ls -rt` перечисляет файлы, начиная с самых старых. Можно также указать имена интересующих вас файлов, тогда команда `ls` выдаст информацию только о них:

```
$ ls -l junk
-rw-r--r-- 1 you 17 Sep 26 16 : 25
$
```

Строки, следующие за именем команды в командной строке, такие, как `-l` и `junk` в приведенном примере, называются аргументами команды. Аргументы обычно бывают флагами или именами файлов, используемыми в команде. Обозначение флага с помощью знака "дефис" и одной буквы, например `-l`, является весьма распространенным. В общем случае, если команда имеет возможные аргументы, то они должны предшествовать аргументам–именам файлов, но появляться могут в любом порядке. Однако система UNIX "капризна" при разборе многочисленных флагов. Например, в стандартной седьмой версии системы команда `ls` не принимает строку

```
$ ls -l-t                                   Не работает в седьмой версии
```

в качестве синонима для `ls -lt`, тогда как другие команды требуют, чтобы флаги были разделены.

По мере изучения системы вы обнаружите, что здесь нет регулярного или систематического подхода к флагам. У каждой команды есть свои "причуды" и свой набор букв для флагов (часто отличающийся от применяемых для выполнения той же функции в другой команде). Такое непредсказуемое поведение системы несколько смущает, и на него часто указывают как на основной недостаток, хотя в новых версиях системы предусмотрено большее единообразие. Пока же мы можем посоветовать вам лучше продумывать флаги в своих программах и на всякий случай держать под рукой копию справочного руководства.

Печать файлов. Команды `cat` и `pr`. Теперь, когда у нас есть файлы, как посмотреть их содержимое? Существует множество программ, решающих эту задачу (возможно, даже слишком много). Один из вариантов — использование редактора:

```
$ ed junk
19                              ed сообщает, что в файле 17 символов
1,$ p                           Печать от первой до последней строки
To be or not to be              В файле только одна строка
q                               Все сделано
```

Редактор начинает работу с сообщения числа символов в файле `junk`: команда '1,$ p' инициирует вывод всех строк файла. После того как вы научитесь пользоваться редактором, вы сможете выбирать части файла, предназначенные для печати. Но бывают ситуации, когда невозможно использовать редактор для печати. Например, размер файла, с которым может работать редактор, имеет определенный предел (несколько тысяч строк). Далее, он может вывести на печать только один файл в данный момент, а нужно печатать несколько, один за другим без перерыва. В таких ситуациях существует несколько способов просмотра файлов.

Прежде всего есть программа `cat` (самая простая из программ печати), которая выдает содержимое всех файлов, указанных как аргументы:

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

Поименованный файл или файлы "катенируются" (отсюда и имя `cat`), т.е. выводятся на терминал последовательно один за другим без промежутков.

С короткими файлами никаких проблем нет, но в случае длинных файлов, если ваш терминал соединен с машиной высокоскоростной линией, вы должны быть достаточно проворны, чтобы с помощью *ctl-s* остановить вывод прежде, чем он исчезнет с экрана. Стандартной команды для выдачи файла на видеотерминал порциями размеров в экран не существует, хотя в каждой системе UNIX такая команда имеется. В вашей системе это может быть команда pg или more. Здесь она называется pr в гл. 6 будет показана ее реализация.

Подобно команде `cat`, `pr` выдает содержимое всех файлов, перечисленных в списке, но в виде, подходящем для устройства печати: каждая страница длиной в 11 дюймов содержит 66 строк, включая заголовок, где указываются номер страницы, имя файла, дата и время его последнего изменения. В месте сгиба бумаги строки пропускаются. Итак, для того чтобы красиво напечатать файл `junk`, затем перейти на следующую страницу и так же красиво напечатать файл `temp`, задайте:

```
$ pr junk temp
Sep 26 16:25 1983 junk Page 1

To be or not to be

(еще 60 пустых строк)

Sep 26 16:26 1983 temp Page 1


That is a question.

(еще 60 пустых строк)

$
```

Команда `pr` может также инициировать печать в несколько столбцов. Так,

```
$ pr -3 filenames
```

печатает каждый файл в три столбца. Можно заменить число 3 любым разумным числом, и команда `pr` "постарается" исполнить задание. (Под `filename` подразумевается список имен файлов.) Команда `pr -m` напечатает набор файлов параллельными столбцами, см. `pr(1)`.

Следует отметить, что `pr` — это не программа форматирования текста: она не разбивает текст на строки и не выравнивает поля. Настоящими программами форматирования являются `troff` и `nroff`, которые обсуждаются в гл. 9.

Существуют также команды, которые производят вывод на высокоскоростное печатающее устройство. Поищите в вашем руководстве команду с именем `lp` или `lpr` или посмотрите в предметном указателе (индексе) слово "`printer`". Выбирайте команду в зависимости от того, какое печатающее устройство подключено к вашей машине. Часто команды pr и lpr используются совместно. После того как `pr` отформатирует информацию должным образом, `lpr` будет управлять процессом передачи па печатающее устройство. Мы вернемся к этому вопросу позднее.

Пересылка, копирование и удаление файлов. Команды `mv`, `cp`, `rm`. Рассмотрим другие команды. Вначале попробуем изменить имя файла. Переименование файла производится "пересылкой" (moving) его от одного имени к другому следующим образом:

```
$ mv junk special
```

Это означает, что файл с именем `junk` будет называться теперь `special`, содержимое его не меняется. Если теперь выполнить команду `ls`, то вы увидите другой список, в котором нет файла `junk`, но есть файл `special`:

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

Будьте осторожны: если вы перешлете файл на место уже существующего файла, то последний будет замещен.

Чтобы иметь копию файла (т. е. две его версии), воспользуйтесь командой `cp`:

```
$ cp special special.save
```

которая продублирует файл `special` в `special.save`. Наконец, когда вы устанете создавать и пересылать файлы, команда rm уберет все указанные файлы:

```
$ rm temp junk
rm: junk nonexistent
$
$ cp special special.save
```

Вы получите предупреждение, если один из удаляемых файлов не существует, но в противном случае `rm`, как и большинство команд UNIX, отработает без лишних слов. Системе не свойственна "болтовня". Приглашения, сообщения об ошибках кратки и не всегда помогают. Краткость может огорчать новичков, но опытных пользователей раздражают "разговорчивые" команды.

Чем может быть имя файла? До сих пор мы употребляли имена файлов, даже не упоминая о том, что является законным именем файла. Теперь пора ввести несколько правил. Во-первых, имя файла ограничено 14 символами[1]. Во-вторых, хотя и можно использовать практически любой символ в имени файла, здравый смысл подсказывает, что следует употреблять только видимые символы и избегать применения символов, несущих определенную смысловую нагрузку. Например, как вы уже видели, в команде `ls` флаг `-t` означает список, упорядоченный по времени, так что если у вас есть файл с именем `-t`, вам придется очень постараться, чтобы он попал в список. (Как, действительно, это сделать?) Кроме знака "минус", есть и другие символы, имеющие специальный смысл в первой позиции, однако пока вы не освоите систему, лучше использовать на этом месте только буквы, цифры, точку и символ подчеркивания. (Точка и символ подчеркивания по традиции употребляются для разбития имени файла на части, как в случае `special.save`). Наконец, не забывайте о различии прописных и строчных букв: `junk`, `JUNK` и `Junk` — разные имена файлов.

Группа полезных команд. Поскольку у нас уже есть основные средства создания файлов, выдачи списка имен файлов, печати их содержимого, мы можем рассмотреть и другие команды обработки файлов. Чтобы изложение было конкретным, будем использовать файл poem, который содержит известное стихотворение Августа Де Моргана. Создадим его с помощью редактора `ed`:

```
$ ed
a
Greate 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 a greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.
w poem
263
q
$
```

Начнем с первой команды, которая подсчитывает число строк, слов и символов в одном или нескольких файлах и называется `wc` по одной из ее функций — подсчета слов ("word counting"):

[1] В современных системах такого ограничения нет, длина файла там ограничена в большинстве случаев 255 символами (прим. редактора)

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

т. е. в файле `poem` восемь строк, 46 слов и 263 символа. Определение "слова" весьма просто — любая последовательность символов, не содержащая пробела, символа табуляции или перевода строки. Команда `wc` произведет подсчет более чем в одном файле (и сообщит итог) и при необходимости "умолчит" о любом счетчике, см. `wc(1)`.

Вторая команда, `grep`, отыскивает в файлах строки, которые подходят под шаблон (ее имя происходит от имени команды редактора `ed` *g/regular-expression/p*, которая объясняется в приложении 1). С помощью этой команды можно найти слово "fleas" в файле `poem`:

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

Команда `grep` может также отыскивать строки, которые не соответствуют шаблону, если используется флаг `-v`. (Флаг назван по имени команды редактора `ed`; действие флага можно представить как инвертирование условия соответствия шаблону.)

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

Команду `grep` можно использовать для поиска в нескольких файлах: в таком случае она будет выдавать имя файла перед каждой строкой, подходящей под шаблон, что позволяет найти место, где произошло сопоставление. Существуют также флаги для нумерации строк и т. д. Команда может применяться и для сопоставления с более сложными шаблонами, чем "fleas", но об этом речь пойдет в гл. 4.

Далее рассмотрим команду `sort`, которая сортирует/входные данные в алфавитном порядке последовательно строку за строкой. Выполним сортировку для файла `poem`, что не очень интересно, но зато наглядно:

```
$ 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,
$
```

Сортируются все строки, и по умолчанию вначале следуют строки, начинающиеся с пробела, за ними — начинающиеся с прописных букв, затем — со строчных букв, так что здесь не выдерживается строго алфавитный порядок.

У команды `sort` есть множество флагов для управления порядком сортировки: обратным, числовым, словарным, с игнорированием начальных пробелов, с сортировкой полей внутри строки и т. п., но, конечно, нужно изучить эти флаги, чтобы уверенно пользоваться ими. Приведем несколько наиболее употребительных флагов:

`sort -r`	Обратный порядок
`sort -n`	Числовой порядок
`sort -nr`	Обратный числовой порядок
`sort -f`	Не учитывать различие прописных и строчных букв
`sort +n`	Начать сортировку с поля $n+1$

В гл. 4 приводится дополнительная информация о команде `sort`.

Следующая команда для работы с файлом — `tail`; она выдает 10 последних строк файла. Этого более чем достаточно для файла poem и полезно для больших файлов. В команде есть флаг, указывающий число выдаваемых строк, так что для печати последней строки файла `poem` можно задать:

```
$ tail -1 poem
  and greter still, and so on
$
```

Команду `tail` можно использовать и для вывода файла, начиная с указанной строки:

```
$ tail +3 filename
```

файл будет печататься с третьей строки (обратите внимание на естественное изменение знака у аргумента).

Последняя пара команд предназначена для сравнения файлов. Допустим, имеется вариант файла `poem` с именем `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.
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.
$
```

Между этими двумя файлами немного различий; на самом деле, нужно постараться, чтобы найти их. Здесь помогут команды сравнения файлов. Команда `cmp` находит первое место, где файлы различаются:

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

Как видите, файлы различаются во второй строке, но неизвестно, в чем состоит их различие, и, кроме того, не отмечены другие различия. Вторая команда сравнения файлов `diff` сообщает обо всех строках, которые изменены, добавлены или удалены:

```
$ 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.
$
```

Итак, вторая строка первого файла `poem` изменена и отличается от второй строки второго файла `new_poem`. То же самое мы наблюдаем и в отношении четвертой строки.

Вообще, команда `cmp` применяется в тех случаях, когда вы хотите убедиться, что два файла действительно имеют одинаковое содержимое. Это быстродействующая команда,

которая работает с любыми (не только с текстовыми) файлами. Командой `diff` следует пользоваться, если вы предполагаете, что файлы различны, и хотите узнать, в каких именно строках они различаются. Команда работает только с текстовыми файлами.

Сводка команд файловой системы. В табл. 1.1 дана краткая сводка описания команд, которые были рассмотрены выше.

`ls`	Вывод списка имен файлов текущего каталога
`ls filenames`	Вывод списка только поименованных файлов
`ls -t`	Вывод списка, упорядоченного по времени создания файла (сначала более новые)
`ls -l`	Вывод данного списка, содержащего большую информацию; допустимо также `ls -lt`
`ls -u`	Вывод списка, упорядоченного по времени последнего использования; допустимо также `ls -lu`, `ls -lut`
`ls -r`	Вывод списка с обратным порядком; допустимо также `ls -rt`, `ls -rit` и т.п.
`ed filename`	Редактирование поименованного файла
`cp file1 file2`	Копирование `file1` в `file2`, старое содержимое `file2` пропадает, если оно было
`mv file1 file2`	Переименование `file1` в `file2`; старый `file2` исчезает, если он был
`rm filenames`	Удаление поименованных файлов безвозвратно
`cat filenames`	Вывод содержимого поименованных файлов
`pr filenames`	Печать содержимого файлов с заголовком, по 66 строк на странице
`pr -n filenames`	Печать в n столбцов
`pr -m filenames`	Печать поименованных файлов в несколько столбцов
`wc filenames`	Подсчет числа строк, слов и символов для каждого файла
`ws -I filenames`	Подсчет числа строк для каждого файла
`grep pattern filenames`	Вывод строк, соответствующих шаблону
`qrep -v pattern files`	Вывод строк, не соответствующих шаблону
`sort filenames`	Сортировка файлов по строкам в алфавитном порядке
`tail filename`	Вывод 10 последних строк файла
`tail -n filename`	Вывод n последних строк файла
`tail +n filename`	Вывод файла, начиная со строки n
`cmp file1 file2`	Вывод места первого расхождения
`diff file1 file2`	Вывод всех расхождений между файлами

Таблица 1.1: Сводка команд файловой системы

1.3 Продолжаем изучать файлы: каталоги

Система отличит ваш файл с именем `junk` от “нужного” файла с тем же именем. Это обеспечивается за счет группировки файлов в каталоги подобно тому, как книги помещаются на полках в библиотеке, так что файлы могут иметь одинаковые имена в разных каталогах без конфликтов.

В общем случае каждый пользователь имеет свой личный каталог, иногда называемый

начальным каталогом, который содержит только принадлежащие ему файлы. Входя в систему, вы оказываетесь в вашем личном (начальном) каталоге. Можно сменить каталог, с которым вы работаете (его часто называют рабочим или текущим каталогом), но ваш личный каталог останется тем же. Если не предпринять специальных действий, новый файл, создаваемый вами, попадает в текущий каталог. Так как вначале текущим является личный каталог, ваш файл никак не связан с файлом, имеющим то же имя, в чужом каталоге.

Каталог может содержать и другие каталоги как обычные файлы. Естественным способом представления такой организации каталогов служит дерево файлов и каталогов. В процессе обхода дерева — от корня вдоль нужных ветвей — можно найти любой файл системы. Можно поступить и наоборот: начать в произвольном месте и двигаться по направлению к корню.

Рассмотрим первый способ. Основным нашим средством будет команда `pwd` ("print working directory" — печать рабочего каталога), которая выведет имена файлов каталога, с которым вы работаете:

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

Команда выведет сообщение о том, что вы находитесь в каталоге you, а сам каталог — в каталоге usr, который в свою очередь находится в корневом каталоге, традиционно обозначаемом как '/'. Символ '/' разделяет компоненты имени: каждый компонент ограничен по длине 14 символами. Во многих системах каталог `/usr` содержит имена каталогов всех пользователей. (Даже если ваш личный каталог не `/usr/you`, команда `pwd` выдаст нечто аналогичное, так что вы сможете следить за последующими примерами.) Введя

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

вы получите тот же самый список файлов, который выдает только `ls`. Если в команде `ls` нет аргументов, то она выводит содержимое текущего каталога; если же ей присвоить имя каталога, то она выдает содержимое указанного каталога.

Далее вводите

```
$ ls /usr
```

Это приведет к появлению длинного списка имен, среди которых есть и ваш начальный каталог `you`.

На следующем шаге попытайтесь распечатать сам корневой каталог. В результате получите ответ, подобный следующему:

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

(Пусть вас не смущает то, что символ '/' имеет два назначения одновременно: имя корневого каталога и разделитель в именах файлов.) Большую часть приведенного списка составляют имена каталогов, но `unix` на самом деле является файлом, содержащим в готовом к выполнению виде ядро системы UNIX (более подробно об этом см. в гл. 2).

Теперь попробуйте ввести

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

(если файл `junk` все еще хранится в вашем каталоге). Имя `/usr/you/jnuk` называется путевым, или абсолютным, именем файла. Путевое имя имеет интуитивный смысл: оно представляет путь по дереву каталогов от корня к отдельному файлу. В системе UNIX есть универсальное правило: всюду, где можно использовать обычное имя файла, можно использовать и абсолютное имя.

Файловая система имеет структуру, подобную генеалогическому дереву:

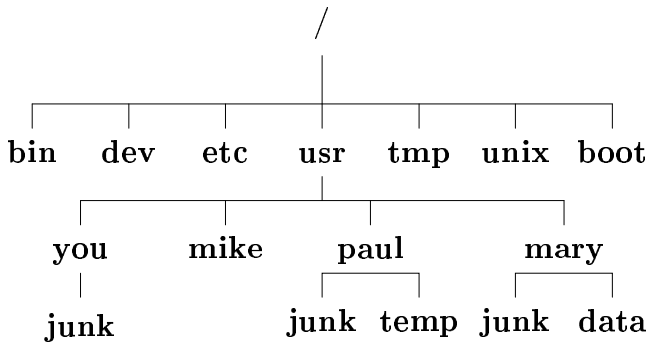


Рис. 1.1: Карта файловой системы UNIX

Ваш файл с именем `junk` никак не связан с файлами пользователей paul или mary. Абсолютные имена не слишком впечатляют, если все интересующие вас файлы находятся в вашем каталоге, но если вы работаете совместно с кем-либо или над несколькими проектами одновременно, они становятся весьма удобными. Например, ваши коллеги могут распечатать файл `junk` с помощью команды

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

Соответственно вы можете выяснить, какие файлы есть у mary:

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

или скопировать один из ее файлов:

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

Вы можете редактировать ее файл:

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

Если Мэри не хочет, чтобы вы имели доступ к ее файлам, или того же не хотите вы, можно обеспечить защищенность каждого файла и каталога. С этой целью предусмотрены права доступа на чтение-запись-выполнение для владельца, группы и всех остальных пользователей. Право доступа используется для контроля обращения к файлу или каталогу. (Вспомните результат `ls -l`.) В нашей системе большинство пользователей, как правило, считают, что открытая система более полезна, чем защищенная, но ваша политика может быть иной; мы вернемся к этому вопросу в гл. 2.

Завершая серию экспериментов с абсолютными именами, попробуйте ввести

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

Не кажутся ли имена вам знакомыми? Когда вы запускаете команду, задавая ее после приглашения, система ищет файл с указанным именем. Вначале поиск ведется в вашем рабочем каталоге (где его, вероятно, найти не удается), затем в каталоге `/bin` и, наконец, в `/usr/bin`. Нет ничего особенного в командах, подобных `cat` или `ls`, за исключением того, что для удобства поиска и управления они находятся в нескольких каталогах. Чтобы убедиться в этом, попытайтесь выполнить некоторые из них, используя абсолютные имена:

```
$ /bin/date
Mon Sep 26 23:39:32 EDT 1983
$ /bin/who
srm tty1 Sep 26 22:20
cvw tty4 Sep 26 22:40
you tty5 Sep 26 23:04
$
```

■ Упражнение: Попробуйте выполнить команду

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

а затем что-либо из предложенного ею. Большее удовольствие это доставит вам в нерабочее время.

Смена каталога. Команда `cd`. Если вы постоянно работаете с информацией, хранящейся в каталоге mary, у вас может возникнуть желание работать с файлами Мэри, а не со своими. Для этого вам достаточно сменить каталог с помощью команды `cd`:

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

Теперь, если использовать имя файла (без /) в качестве аргумента для команд `cat` или `pr`, это будет файл из каталога mary. Смена каталога не влияет на права доступа к файлу: если файл был недоступен из вашего каталога, то таким он и останется.

Обычно бывает удобно сгруппировать свои файлы так, чтобы все файлы, относящиеся к одному проекту, попали в отдельный каталог. Например, если вы надумаете писать книгу, то весь текст вы, естественно, захотите хранить в каталоге с именем `book` (книга). Команда `mkdir` создает новый каталог:

```
$ mkdir book                Создать каталог
$ cd book                   Перейти в него
$ pwd                       Убедиться, что вы попали куда надо
/usr/you/book
...                         Работа над книгой (прошло несколько минут)
$ cd ..                     Подняться на один уровень в файловой системе
$ pwd
/usr/you
$
```

Обозначение '.' относится к "родителю" того каталога, с которым вы работаете в данный момент, а именно к каталогу, расположенному на один уровень ближе к корню. Обозначение '.' является синонимом текущего каталога.

```
$ cd                        Возврат в личный каталог
```

Команда вернет вас в ваш начальный каталог, т. е. в каталог, в который вы попадаете при входе в систему.

После того как книга опубликована, можно почистить каталог. Чтобы удалить каталог `book`, удалите все содержащиеся в нем файлы (быстрый способ выполнения этой операции мы вскоре покажем), затем перейдите в родительский каталог для `book` и задайте команду

```
$ rmdir book                Команда rmdir удаляет только пустые каталоги.
```

1.4 Интерпретатор shell

Когда система выдает приглашение $ и вы вводите команды для выполнения, вы имеете дело не с ядром самой системы, а с неким посредником, называемым интерпретатором команд, или `shell`. Это обычная программа, подобная `date` или `who`, хотя она может делать удивительные вещи. Тот факт, что программа `shell` находится между вами и ядром, дает реальные выгоды, и некоторые из них мы вам укажем. Применение программы–посредника обеспечивает три главных преимущества:

- сокращенные имена файлов: можно задать целое множество файлов в качестве аргументов команде, указав шаблон для имен: `shell` будет искать файлы, имена которых соответствуют заданному шаблону;
- переключение ввода–вывода: вывод любой программы можно направить в файл, а не на терминал, ввод можно получать из файла, а не с терминала. Ввод и вывод можно даже передать другим программам;
- создание собственной среды: можно определить свои собственные команды и правила сокращений.

Сокращенное имя файла. Начнем с шаблонов имен файлов. Допустим, вы вводите обширный документ, наподобие книги. Логически он разбивается на множество частей, аналогично главам и разделам. И физически его следует разбить на части, поскольку затруднительно редактировать большие файлы. В этом случае для печати всего текста нужно указать ряд файлов. У вас могут быть отдельные файлы для каждой главы с именами `ch1`, `ch2` и т. д. Если каждая глава разбита на разделы, вы можете создать файлы с именами

```
ch.1
ch.2
ch.3
...
ch2.1
ch2.2
...
```

что и используется в нашей книге. При систематизированном именовании можно указать с первого взгляда, в какой ряд файлов попадает данный файл.

Как быть, если вы захотите напечатать книгу? Можно задать

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

но вы быстро устанете вводить имена файлов и начнете делать ошибки. Именно здесь приходит на помощь сокращенное имя файла. Если задать

```
$ pr ch*
```

интерпретатор shell воспримет * как любую последовательность символов, поэтому ch* является шаблоном, под который подходят все имена файлов из текущего каталога, начинающиеся на ch. Интерпретатор shell создаст список в алфавитном порядке (Порядок, конечно, не строго алфавитный, прописные буквы предшествуют строчным. Чтобы узнать порядок, используемый в программе sort, см. ascii(7).) и передаст его программе pr. Команда pr никогда "не узнает" * ; выбор по шаблону, который shell производит в текущем каталоге, порождает список строк, передаваемых pr.

Ключевой момент состоит в том, что способ сокращения имени файла — это не свойство программы pr, а возможность, реализуемая интерпретатором shell. Поэтому вы можете использовать ее для создания последовательности имен файлов в любой команде, например для подсчета числа слов первой главы:

```
$ 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
  75   323  2030 ch1.6
3801 16210 88930 total
$
```

Существует программа с именем echo ("эхо"), которая особенно ценна для экспериментов со "смыслом" сокращенных имен. Как вы смогли догадаться, echo лишь повторяет свои аргументы.

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

Но аргументы могут формироваться путем выбора по шаблону. Так, команда

```
$ echo ch1.*
```

перечисляет имена всех файлов в гл. 1,

```
$ echo *
```

перечисляет имена всех файлов текущего каталога в алфавитном порядке,

```
$ pr *
```

выводит на печать содержимое всех ваших файлов (в алфавитном порядке), а

```
$ rm *
```

удаляет все файлы текущего каталога. (Лучше быть абсолютно уверенным, что вы действительно этого хотите!)

Символ * может встречаться не только в конце имени файла. Его можно использовать всюду и даже по нескольку раз. Поэтому

```
$ rm *.save
```

удалит все файлы, оканчивающиеся на .save.

Заметьте, что все имена файлов выбираются в алфавитном порядке, который отличается от числового. Если в вашей книге 10 глав, порядок может быть не тем, на который вы рассчитываете, поскольку ch10 идет перед ch2:

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

Символ * — не единственный способ задания шаблона для интерпретатора shell, хотя и наиболее часто используемый. Шаблон [...] задает любые символы из перечисленных внутри скобок. Несколько подряд следующих букв или цифр можно задать в сокращенном виде:

```
$ pr ch[12346789]*          Печать глав 1,2,3,4,6,7,8,9, но не 5
$ pr ch[1-46-9]*            То же самое
$ rm temp[a-z]              Удалить все tempa, ... , tempz
```

Шаблон ? задает любой одиночный символ:

```
$ ls ?                      Список файлов с именем из одного символа
$ ls -l ch?.1               Список ch1.1 ch2.1 ch3.1 и т. д., но не ch10.1
$ rm temp?                  Удалить все файлы tempi, ..., tempa и т. д.
```

Отметим, что шаблоны сопоставляются только с именами существующих файлов. В частности, нельзя создать новые имена файлов с помощью шаблонов. Например, если вы захотите расширить ch до chapter в каждом имени файла, то такой вариант вам не поможет:

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

поскольку chapter.* не соответствует ни одному из существующих имен файлов.

Символы шаблонов, подобные * , могут использоваться в абсолютных именах наравне с обычными именами файлов; сопоставление происходит для каждого компонента абсолютного имени, содержащего специальный символ. Так, /usr/mary/* инициирует поиск

файлов в /usr/mary/, а /usr/*/calendar порождает список абсолютных имен всех пользователей, работающих с каталогом calendar.

Если вам когда-нибудь придется отказаться от специального назначения символов *, ? и др., заключите весь аргумент в апострофы, например:

```
$ ls '?'
```

Можно также предварить специальный символ обратной дробной чертой:

```
$ ls \?
```

(Вспомните, что, поскольку ? не является символом стирания или уничтожения, обратная дробная черта перед ним будет обрабатываться не ядром, а интерпретатором shell.) Использование кавычек подробно рассматривается в гл. 3.

■ Упражнение: В чем состоит различие между следующими командами:

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

Переключение ввода–вывода. Большинство команд, которые мы рассматривали, производят вывод на терминал, некоторые из них, подобно редактору, осуществляют ввод с терминала. А теперь приведем почти универсальное правило: терминал может быть заменен для ввода, вывода или обеих операций на файл.

Например,

```
$ ls
```

выдает список файлов на ваш терминал. Но если задать

```
$ ls > filelist
```

то тот же список файлов помещается вместо этого в файл filelist. Символ > означает, что выходной поток должен быть помещен в указанный далее файл, а не выведен на терминал. Файл будет создан, если он ранее не существовал, или будет заменено содержимое старого. На своем терминале вы ничего не получите. В качестве другого примера можно слить несколько файлов, "перехватив" выходной поток команды cat и направив его в файл:

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

Символ >> действует подобно >, но указывает на необходимость добавить выходной поток к концу файла. Значит, команда

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

сольет содержимое f1, f2, f3 и добавит результат в конец temp, вместо того чтобы затереть его старое содержимое. Так же как и для операции >, если файл temp не существует, то он будет создан первоначально пустым.

Аналогично символ < означает, что входной поток программы берется из последующего файла, а не с терминала. Так, можно заготовить письмо в файле let, а затем послать его нескольким адресатам:

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

Во всех этих примерах наличие пробелов по обе стороны символа > или < не обязательно, но такое представление традиционно.

Имея возможность переключать выходной поток с помощью <, мы можем комбинировать команды, получая эффект, недостижимый другим способом. Например, можно выдать список пользователей в алфавитном порядке

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

Поскольку команда who выдает по одной строке на каждого пользователя, работающего в системе, а wc -l производит подсчет строк (подавляя вывод числа слов и символов), можно подсчитать число пользователей с помощью команд:

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

и число файлов в текущем каталоге:

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

хотя в это число войдет и сам файл temp. Можно выдать список имен файлов в три столбца, задав

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

Наконец, можно убедиться в том, что некий пользователь вошел в систему, комбинируя команды who и grep:

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

Во всех перечисленных выше примерах, как и в случае имен файлов, содержащих образы типа * , важно понимать, что символы < и > обрабатываются самим интерпретатором shell, а не отдельной программой. Благодаря этому переключение входного и выходного потоков возможно для любой программы, причем сама программа даже "не подозревает", что происходит что-то необычное.

Изложенное подводит нас к важному выводу. Команда

```
$ sort < temp
```

сортирует содержимое файла temp так же, как

```
$ sort temp
```

но в их действиях есть различие. Поскольку строка < temp обрабатывается интерпретатором shell, первая команда sort не воспринимает файл temp как свой аргумент; она просто сортирует собственный стандартный входной поток, который переключен интерпретатором на файл temp. В то же время в последнем случае имя temp передается команде sort в качестве аргумента, она читает его и сортирует файл. Команде sort можно передать список файлов:

```
$ sort temp1 temp2 temp3
```

но, если имена файлов отсутствуют, она всегда будет сортировать стандартный входной поток. Это существенная особенность большинства команд: если не указаны имена файлов, то обрабатывается стандартный входной поток. Следовательно, достаточно ввести имя команды, чтобы посмотреть, как она выполняется. Например,

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

В дальнейшем мы покажем, как реализуется этот принцип.

■ Упражнение: Объясните, почему команда

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

включает `ls.out` в список имен.

■ Упражнение: Объясните результат выполнения команды

```
$ wc temp > temp
```

Что произойдет, если вы ошибетесь в имени команды, задав

```
$ woh > temp
```

Программные каналы. Все примеры, приведенные в конце предыдущего раздела, основаны на одном и том же приеме: выходной поток одной программы передается в качестве входного потока для другой программы через временный файл. Сам временный файл больше не имеет никакого смысла; в самом деле, неудобно использовать такой файл. Это соображение привело к возникновению одной из фундаментальных концепций системы UNIX, идеи программного канала. Программный канал представляет собой средство связи выходного потока одной программы с входным потоком другой без всяких временных файлов; соединение программным каналом двух или более программ называется конвейером.

Пересмотрим теперь некоторые из предыдущих примеров с точки зрения использования программных каналов вместо временных файлов. Вертикальная черта служит указанием интерпретатору `shell` для создания конвейера:

```
$ who | sort          Печать отсортированного списка пользователей
$ who | wc -l         Подсчет числа пользователей
$ ls | wc -1          Подсчет числа файлов
$ ls | pr -3          Вывод списка имен файлов в три столбца
$ who | grep mary     Поиск определенного пользователя
```

Всякая программа, вводящая информацию с терминала, может вводить ее и по программному каналу; всякая программа, производящая вывод на терминал, может выдавать информацию в программный канал. Это тот случай, когда приносит плоды решение читать стандартный входной поток, если не заданы никакие файлы. Любая программа, выполняющая данное соглашение, может быть включена в конвейер. В рассмотренных выше примерах команды `pr`, `grep`, `sort` и `wc` используются именно таким способом.

Можно связать конвейером сколь угодно много программ. Например,

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

создает список имен файлов в три столбца и выдает его на печатающее устройство, а

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

подсчитывает, сколько раз пользователь Мэри входила в систему.

Программы, связанные конвейером, выполняются одновременно, а не последовательно одна за другой. Это означает, что программы в конвейере могут вступать в диалог; ядро выполняет необходимые операции переключения и синхронизации, чтобы такая схема работала. Большинство команд следует определенному образцу, поэтому они хорошо вписываются в конвейер и могут выполняться в нем на любом месте. Обычный вызов команды имеет вид:

```
команда флаги возможные имена файлов
```

Если имена файлов не указаны, то команда читает стандартный входной поток, который по умолчанию поступает с терминала (что удобно для экспериментирования), однако возможно его переключение на файл или программный канал. Кроме того, во многих командах выдача идет в стандартный выходной поток, который по умолчанию направлен на терминал, но его также можно переключить на файл или программный канал.

Сообщения же об ошибках, выдаваемые командами, следует обрабатывать по-другому, иначе они затеряются в файле или программном канале. Поэтому каждая команда имеет еще один стандартный файл, называемый файлом диагностики, который обычно связан с вашим терминалом:

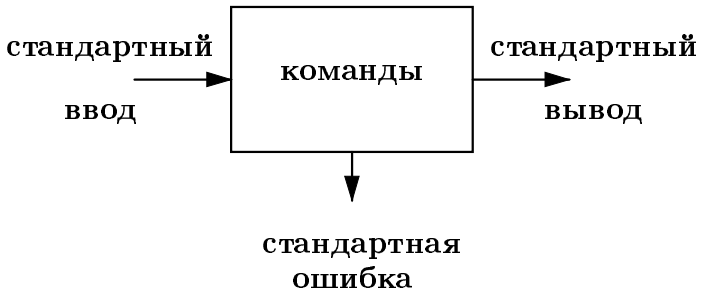


Рис. 1.2: Схема потоков в UNIX

Почти все рассматривавшиеся выше команды укладываются в эту схему; исключение составляют `who` и `date`, не имеющие входной информации, а также те, например `cmp` или `diff`, которые имеют определенное число входных файлов. (Посмотрите их флаг '-'.)

■ Упражнение: Объясните разницу между командами

```
$ who | sort
```

и

```
$ who > sort
```

Процессы. Интерпретатор `shell` выполняет и некоторые другие операции, помимо связывания через программный канал. Рассмотрим кратко вопрос одновременного выполнения нескольких программ, о чем уже упоминалось при обсуждении программных каналов. Например, можно запустить две команды с помощью одной командной строки, разделив их точкой с запятой; интерпретатор `shell` распознает этот символ и разобьет 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
her tty7 Sep 26 23:34
```

Обе команды будут выполнены (подряд) прежде, чем интерпретатор вновь вернется с приглашением.

Можно также при желании запустить несколько команд одновременно. Предположим, что вы собираетесь заняться длительными вычислениями, например, подсчитать число слов в вашей книге, но не хотите ждать окончания команды wc для перехода к другой работе. Тогда можно задать:

```
$ wc ch* > wc.out &
6944                    Shell дает номер процесса
$
```

Амперсанд (&) в конце командной строки указывает интерпретатору, что нужно запустить данную команду, а затем сразу перейти к получению последующих команд с терминала, т. е. не ждать ее завершения. Итак, команда будет выполняться, а вы можете отвлечься на что-нибудь другое. Переключение выходного потока на файл `wc.out` предотвращает возможность его смешивания с той информацией, которая появится на терминале в процессе дальнейшей работы.

Каждый экземпляр запущенной программы называется процессом. Число, выдаваемое `shell` в ответ на команду, введенную с &, является номером процесса. Его можно

использовать в других командах в качестве ссылки на данный экземпляр выполняемой программы.

Важно понимать различие между программами и процессами. Скажем, `wc` — это программа, но каждый запуск программы `wc` создает новый процесс. Если одновременно выполняется несколько экземпляров одной программы, то любой из них считается отдельным процессом с отличным от других номером.

Если конвейер завершается операцией `&`

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

то все процессы этого конвейера начинают выполняться сразу, и & относится ко всем программам, участвующим в конвейере. Однако выдается только номер процесса, относящийся к последней программе в конвейере. Команда

```
$ wait
```

ожидает, пока не завершатся все процессы, запущенные с помощью &. Если она не возвращается сразу, значит, у вас есть незавершенные команды. Прервать выполнение команд можно, нажав клавишу *DELETE*.

Можно использовать номер процесса, сообщаемый интерпретатором, для остановки процесса, инициированного операцией &:

```
$ kill 6944
```

Если вы забыли номер процесса, команда `ps` выведет сообщение обо всех ваших процессах. В том случае, когда вам некогда, команда `kill 0` уничтожит все ваши процессы, за исключением начального процесса–интерпретатора. Если же вам интересно, что делают другие пользователи, команда `ps -ag` сообщит обо всех выполняемых процессах. Приведем пример 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
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
$
```

Здесь PID — номер процесса; TTY — терминал, связанный с процессом (как в команде who); TIME — затраченное время процессора в минутах и секундах, а в конце строки — выполняемая команда. Команда ps — одна из тех команд, которые выполняются поразному в различных версиях системы, так что вывод в вашей системе может иметь другой формат. Даже аргументы могут отличаться — см. в своем справочном руководстве страницу ps(1).

Процессы, подобно файлам, имеют иерархическую структуру: у каждого процесса есть родитель и могут быть потомки. Ваша копия интерпретатора shell была создана процессом, обслуживающим связь через терминал с системой. Когда вы запускаете команды, их процессы становятся прямыми потомками вашей копии shell. Если вы запускаете программу "внутри" одной из этих команд, например команду '!' для выхода из редактора ed, то создается новый процесс–потомок, который является, таким образом, уже внуком для shell.

Иногда процесс выполняется столь долго, что вы уже жалеете, что запустили его. Выключите терминал и идите домой, не дожидаясь его окончания. Но если вы выключите терминал или отсоедините его от сети, то процесс будет уничтожен, даже если применен &. Специально для такого случая предусмотрена команда nohup ("no hangup" — без отбоя).

Введите

```
$ nohup команда &
```

и команда будет продолжать выполняться, даже если выйти из системы. Любой результат выполнения команды будет сохранен в файле, называемом nohup.out. После запуска программы никакая команда nohup уже не поможет.

Если ваш процесс требует много процессорного времени, вы можете облегчить участь тех, кто работает вместе с вами, запустив его с приоритетом ниже обычного. Это можно сделать с помощью программы nice:

```
$ nice большая-команда &
```

Команда nohup автоматически вызывает nice, поскольку раз уж вы собираетесь выйти из системы, то можете позволить, чтобы ваша команда выполнялась дольше.

Наконец, вы можете дать указание системе запустить ваш процесс в необычное время, скажем, утром, когда все нормальные люди спят, а не работают на машине. Команда называется at(1):

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

Это пример типичного использования команды at, но, конечно, команды можно брать и из файла:

```
$ at 3am << файл
$
```

Время можно задавать исходя из 24–часового цикла как 2130 или 12–часового как 930pm.

Создание среды. Одним из достоинств системы UNIX является то, что вы можете легко адаптировать ее по своему вкусу либо в соответствии с местными традициями программистского мира. Например, как отмечалось выше, существуют разные стандарты для символов стирания и удаления; по умолчанию используются # и @ . Вы можете изменить их в любой момент с помощью команды

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

где 'e' обозначает нужный вам символ стирания, а 'k' — символ удаления. Однако задавать эти символы при каждом входе в систему — довольно нудное занятие.

На помощь здесь приходит shell. Если в вашем начальном каталоге есть файл .profile, интерпретатор будет запускать команды из него при каждом входе в систему перед выдачей первого приглашения. Поэтому можно поместить команды в .profile для установки требуемой среды, и они будут выполняться всякий раз при входе в систему.

Большинство пользователей первым делом помещают в свой файл .profile команду

```
$ stty erase <-
```

Мы использовали <- , чтобы сделать символ стирания видимым, но вы должны поместить в .profile настоящий символ "шаг назад". Команда stty воспринимает также обозначение $\hat{}x$ в качестве *ctl-x*, поэтому тот же результат можно получить, вводя:

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

поскольку *ctl-h* и есть шаг назад. (Символ '^' ранее применялся для операции программного канала |, поэтому его следует экранировать с помощью кавычек.) Если на вашем терминале нет возможности задать интервалы табуляции, можно добавить к строке с `stty` аргумент `-tabs`:

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

Если у вас есть желание посмотреть, как загружена система, при входе в нее добавьте для подсчета пользователей конвейер

```
who | wc -l
```

Если имеется служба новостей, можно добавить команду `news`. Те, кому нравится игра `fortune`, могут добавить

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

Спустя некоторое время вы обнаружите, что период входа в систему существенно возрос, и выберете для себя оптимальный вариант.

Некоторыми возможностями системы можно управлять с помощью так называемых `shell`-переменных, значения которых пользователь может и посмотреть, и установить. Например, строка-приглашение, обозначаемая ранее как $, на самом деле хранится в `shell`-переменной, называемой `PS1`, и можно присвоить ей любое значение:

```
PS1='Yes, dear ?'              Да, дорогой?
```

Кавычки необходимы, поскольку в строке-приглашении есть пробелы, а в этой операции пробелы перед и после "=" не допускаются.

Интерпретатор также выделяет переменные `HOME` и `MAIL`. `HOME` представляет собой имя вашего начального каталога; переменная обычно имеет правильное значение даже без установки ее в `.profile`. Переменная `MAIL` содержит имя стандартного файла, в котором хранится ваша почта. Если вы переопределите ее для интерпретатора, то будете в случае появления новой почты получать извещение после ввода каждой команды (Эта возможность плохо реализована в `shell`. Просмотр файла после ввода каждой команды сказывается на времени работы системы. Кроме того, если вы достаточно долго работаете в редакторе, то не узнаете о новой почте, поскольку не задаете новых команд в начальном интерпретаторе `shell`. Лучшим решением была бы проверка через каждые несколько минут, а не после очередной команды. В гл. 5 и 7 будет показано, как реализовать такую проверку получения почты. Третьим решением, хотя и не для всех доступным, может быть извещение, посылаемое самой программой `mail`, — она точно "знает", когда появится почта.)

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

(В вашей системе файл для почты может быть другим; распространенным является и имя /usr/mail/you.)

Наиболее полезной переменной интерпретатора shell, вероятно, считается та, которая определяет, где проводится поиск команд. Вспомните, что, когда вы вводите имя команды, интерпретатор обычно вначале ищет его в текущем каталоге, затем в /bin и далее в /usr/bin. Эта последовательность каталогов называется путем поиска и хранится в переменной интерпретатора с именем PATH. Если определенный по умолчанию путь поиска вас не устраивает, то его можно изменить (опять в файле .profile). Например, строкой ниже к стандартному пути поиска добавляется /usr/games:

```
PATH=.:/bin:/usr/bin:/usr/games/                    Один способ...
```

Синтаксис может показаться вам несколько странным: последовательность имен каталогов разделяется двоеточием. Напоминаем, что '.' обозначает текущий каталог. Можно опустить имя '.', пустой компонент в PATH обозначает текущий каталог.

Другой способ установить значение PATH — просто добавить к предыдущему значению

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

Можно получить значение любой переменной интерпретатора, предварив ее имя символом $. В приведенном примере выражение $PATH выбирает текущее значение, к которому добавляется новый компонент, и результат вновь присваивается PATH. Можно проверить это с помощью команды echo:

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

Если у вас есть свои собственные команды, то, возможно, вы захотите собрать их в свой каталог и добавить его к пути поиска. В таком случае переменная PATH может принять подобное значение:

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

Вопрос создания своих собственных команд мы обсудим в гл. 3.

Существует еще одна переменная, часто используемая текстовыми редакторами, более популярными, чем ed, — TERM, которая указывает тип используемого терминала. Эта информация позволяет программам более эффективно работать с экраном. Поэтому можно в .profile добавить, например, следующее:

```
TERM=adm3
```

Можно применять переменные и для сокращения записи. Если вам часто приходится ссылаться на каталог с длинным именем, имеет смысл добавить строку типа

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

к файлу `.profile`, чтобы использовать:

```
$ cd $d
```

Ваши собственные переменные, скажем `d`, по традиции обозначаются строчными буквами, что позволяет отличить их от тех, которые, как `PATH`, использует сам интерпретатор.

Наконец, вы должны сообщить интерпретатору, что будете использовать некоторые переменные в других программах; для этого служит команда `export`, к которой мы вернемся в гл. 3:

```
export MAIL PATH TERM
```

Подводя итоги, покажем, как может выглядеть типичный файл `.profile`:

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

Мы еще далеко не полностью перечислили возможности интерпретатора. В частности, можно создать собственные команды, собрав в файле уже существующие команды, и в данном случае уже будут интерпретироваться команды из этого файла. Удивительно, как многого можно добиться с помощью такого в основе своей простого средства. Мы рассмотрим его в гл. 3.

1.5 Другие средства UNIX

С системой UNIX связано значительно больше проблем, чем те, которые мы обсудили в настоящей главе. Многие из них еще будут рассматриваться в этой книге. Теперь

вы должны чувствовать себя увереннее в общении с системой и, в частности, уметь ориентироваться в ее справочном руководстве. Если у вас появятся вопросы, в каких случаях и как использовать команды, всегда обращайтесь к руководству.

Имеет смысл также периодически заглядывать в руководство, чтобы освежить свои знания об известных вам командах и познакомиться с новыми. В руководстве описывается множество программ, которые мы не обсуждали, включая компиляторы языков программирования, подобные Фортран 77, программы–калькуляторы типа `bc(1)`, `cu(1)` и `uucp(1)` — программы для межмашинного взаимодействия, графические пакеты, статистические программы и даже такая программа, как `units(1)`.

Как уже отмечалось ранее, эта книга не заменяет справочное руководство, а дополняет его. В последующих главах мы будем изучать фрагменты и программы системы UNIX, используя информацию, приведенную в руководстве, и следуя логике взаимодействия компонентов системы. Хотя это взаимодействие явно не описано в руководстве, именно оно лежит в основе приемов программирования в системе.

Историческая и библиографическая справка. Первой публикацией по системе UNIX является статья Д. М. Ритчи и К. Л. Томпсона "The UNIX Time-sharing System" (Communications of the ACM, July, 1974). Она была перепечатана там же в январе 1983г. (стр. 89 из перепечатки есть в мартовском выпуске 1983г.). Это обзор системы для специалистов по операционным системам, но мы рекомендуем познакомиться с ним всем программистам.

Специальный июльский выпуск журнала The Bell System Technical Journal (BSTJ) 1978г. содержит ряд статей, посвященных дальнейшему развитию системы и некоторым историческим вопросам, включая переработанный вариант статьи Ритчи и Томпсона. Следующий специальный выпуск BSTJ, содержащий новые статьи по системе UNIX, вышел в свет в 1984 г.

В статье В. Кернигана и Д. Мэши "The UNIX Programming Environment" (IEEE Computer Magazine, April, 1981) делается попытка выделить наиболее существенные свойства системы с точки зрения программистов.

В справочном руководстве по системе UNIX, какой бы ни была ваша версия системы, вы найдете команды, системные функции и правила взаимодействия с ними, форматы файлов и процедуры поддержания системы. Вы не сможете обойтись без этого руководства, хотя на первых порах, пока вы не начнете программировать, вам, вероятно, будет достаточно прочесть только часть первого тома. Том 1 справочного руководства по седьмой версии системы опубликован издательством Холта, Райнхарта и Уинстона. Том 2 "Documents for Use with the UNIX Timesharing System" справочного руководства содержит рекомендации по использованию и описания большинства команд. В частности, здесь описываются достаточно подробно средства подготовки документации и разработки программ. В конечном счете, мы уверены, вас заинтересует этот материал.

Хорошим введением для совсем "зеленых" новичков и непрограммистов представляется книга Э. и Н. Ломато "A UNIX Primer" (Prentice-Hall, 1983).

Глава 2

Файловая система

Все, с чем работает система UNIX, она воспринимает в виде файла. Это не такое уж упрощение, как может показаться на первый взгляд. Когда разрабатывалась первая версия системы, даже прежде, чем ей дали имя, все усилия сосредоточились на создании структуры файловой системы, которая должна была быть простой и удобной в использовании. Файловая система — ключевое звено, обеспечившее успешное применение UNIX. Это наилучший пример философии "прекрасное в малом", показывающий, какой мощи можно достичь реализацией нескольких хорошо продуманных идей.

Для описания команд и их взаимодействия нужно хорошо знать структуру и внешние связи файловой системы. В этой главе излагается большинство вопросов, связанных с файловой системой, — понятие файла и его представление, каталоги и иерархия файловой системы, права доступа, индексный дескриптор (внутреннее представление файла в системе) и файлы устройств. Поскольку основная работа в системе связана с манипулированием файлами, существует множество команд для анализа и модификации файла; здесь вводятся наиболее употребительные команды.

2.1 Основные сведения о файлах

Файл представляет собой последовательность байтов. (Байт — небольшая порция информации, обычно размером в восемь бит. Для наших целей можно считать байт синонимом слова "символ".) Никаких ограничений по структуре системой на файл не накладывается, и никакого смысла не приписывается его содержимому: смысл байтов зависит исключительно от программ, обрабатывающих файл. Более того, как мы увидим позднее, это верно не только для файлов, хранящихся на дисках, но и для файлов, представляющих периферийные устройства. Записи на магнитных лентах, почта, символы, вводимые с клавиатуры, вывод на печатающее устройство, данные, передаваемые по конвейеру — каждый из этих файлов система и входящие в нее программы воспринимают просто как последовательность байтов.

Лучше всего познакомиться с файлами экспериментальным путем, так что начнем с создания небольшого файла:

```
$ ed
a
now is the time.
for all good people
.
```

```
w junk
36
q
$ls -l
-rw-r--r-- 1 you 26 Sep 27 06:11 junk
$
```

Здесь `junk` — это файл из 36 байт, т. е. 36 символов, которые вы ввели (не считая, конечно, символов, введенных при коррекции ошибок). Команда `cat` показывает содержимое файла в следующем виде:

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

Команда `od` ("octal dump" — восьмиричный дамп) выдает "изображение" 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
$
```

Флаг `-c` означает, что следует интерпретировать байты как символы. Если добавить флаг `-b`, то можно, кроме того, показать байты как восьмиричные числа. (В каждом байте файла находится число, достаточно большое, чтобы закодировать изображаемый символ. В большинстве систем UNIX кодировка называется ASCII ("American Standard Code for Information Interchange" — американский стандартный код для обмена информацией), но на некоторых машинах, особенно произведенных фирмой IBM, используется кодировка, называемая EBCDIC ("Extended Binary Coded Decimal Interchange Code" — расширенная двоично закодированная десятичная общая кодировка). Здесь и далее в книге мы будем применять множество символов ASCII; воспользуйтесь командой `cat /usr/pub/ascii` или прочтите `ascii(7)`, чтобы узнать восьмиричные значения 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  d   l   e   \n
        160 154 145 012
0000044
$
```

Семизначные числа в колонке слева показывают место в файле, т. е. порядковый номер следующего изображаемого символа в восьмиричной форме. Между прочим, приоритет

восьмиричных чисел — это пережиток времен PDP-11, когда восьмиричной нотации отдавалось предпочтение. Для других машин больше подходит шестнадцатиричная нотация; флаг -x предписывает команде od печатать информацию в шестнадцатиричной форме.

Обратите внимание на то, что после каждой строки идет символ с восьмиричным значением 012. Это символ перевода строки для ASCII; система помещает его во входной поток, когда вы нажимаете клавишу *RETURN*. По соглашению, заимствованному из языка Си, символ перевода строки изображается как \n, что лишь облегчает чтение. Такого соглашения придерживаются только программы типа od; в файле же хранится единственный байт 012.

Перевод строки — наиболее типичный пример специального символа. Другими специальными символами, связанными с некоторыми операциями управления терминалом, являются символы: шаг назад (восьмиричное значение 010 изображается как \b), табуляция (011, \t), возврат каретки (015, \r).

Важно в каждом случае различать, в каком виде символ хранится в файле и как он интерпретируется в той или иной ситуации. Например, когда вы вводите с клавиатуры символ "шаг назад" (предполагая, что это ваш символ стирания), система воспринимает его как требование уничтожить символ, введенный перед ним. Оба символа — и стираемый, и "шаг назад" — на терминале исчезают, а курсор возвращается на одну позицию назад.

Если ввести последовательность

```
\<-
```

(т. е. символ \и вслед за ним "шаг назад"), то ядро в этом случае "считает", что вы действительно хотите ввести символ <-, поэтому \исчезает, а в вашем файле появляется байт 010. Когда "шаг назад" отражается на терминале, происходит возврат курсора, так что он указывает на символ \.

При выводе файла, содержащего символ <- , он передается на терминал без обработки, что опять приводит к передвижке курсора на одну позицию назад. Если воспользоваться командой od, чтобы вывести файл, содержащий символ <-, он появится как байт со значением 010 или, если указан флаг -c, как \b.

Аналогичную ситуацию мы имеем и с символом табуляции: при вводе он отражается на терминале и посылается программе, осуществляющей ввод; при выводе символ табуляции просто передается на терминал и интерпретируется. Однако в отличие от предыдущего случая здесь можно указать ядру, что вы хотите получить интерпретацию табуляции при выводе; тогда вместо изображения каждого символа табуляции будет выдаваться нужное число пробелов, чтобы перейти к следующей позиции табуляции. Позиции табуляции установлены в столбцах 9, 17, 25 и т. д. Команда

```
$ stty = tabs
```

приводит к замене символов табуляции пробелами при выводе на терминал — см. описание stty(1).

Обработка символа *RETURN* аналогична рассмотренной выше. Ядро отображает *RETURN* на терминале как "возврат каретки" и "конец строки", но во входной поток попадает только "перевод строки". При выводе этот символ вновь заменяется символами возврата каретки и конца строки.

Подход системы UNIX к представлению управляющей информации нетрадиционен, особенно использование символа перевода строки для завершения строки (в качестве

конца посылки). Многие системы вместо этого трактуют каждую строку как "запись", содержащую не только введенные данные, но и счетчик числа символов в строке (специального символа конца строки нет). В других системах каждая строка завершается символами возврата каретки и перевода строки, поскольку такая последовательность необходима для вывода на большинство терминалов. (Слово "linefeed" — завершение строки, синоним перевода строки, поэтому такую последовательность часто называют "CRLF", что невозможно произнести.)

Система UNIX не делает ни того, ни другого: нет записей и счетчиков, к тому же ни в одном файле нет никаких байтов, которые бы вы или ваша программа не поместили туда. Символ перевода строки преобразуется в два символа — возврата каретки и перевода строки при выводе на терминал, но программы должны иметь дело с одним символом перевода строки, поскольку это все, что они могут "увидеть". В большинстве случаев подобная простая схема является оптимальной. Если необходима более сложная структура, ее легко построить на базе этой, тогда как получить простое из сложного значительно трудней.

Поскольку завершение строки обозначается символом перевода строки, можно ожидать, что и файл завершается другим специальным символом, скажем \e как сокращение "end of file" — конец файла. Но, посмотрев на вывод программы `od`, вы не увидите никакого специального символа в конце файла — он просто кончается. Вместо того чтобы использовать специальный символ, система отмечает конец файла сообщением о том, что данных в файле больше нет. Ядро запоминает длину файла, поэтому программа встречает конец файла после обработки всех составляющих файл байтов.

Программы выбирают данные из файла с помощью системного обращения с именем `read` (подпрограмма в ядре). При каждом обращении к `read` читается следующая часть файла, например очередная введенная строка. Подпрограмма `read` также сообщает число прочитанных байтов файла, поэтому конец файла обнаруживается, когда она сообщает: "прочитано 0 байт". Если какие-либо байты оставались в файле, то подпрограмма `read` выдала хотя бы часть их. На самом деле, отказ от ввода байта со специальным значением "конец файла" вполне оправдан, поскольку, как отмечалось ранее, смысл содержимого байта зависит от интерпретации файла. Но все файлы имеют конец, и поэтому их следует читать с помощью подпрограммы `read`, а возврат нуля — это зависящий от интерпретации способ представления конца файла без использования специальных символов.

Когда программа читает с вашего терминала, каждая введенная строка передается программе ядром только после ввода символа перевода строки (т. е. нажатия *RETURN*). Поэтому если вы сделаете ошибки и заметите это до ввода *RETURN*, можно вернуться и исправить их. Если символ перевода строки введен до того, как вы заметили ошибку, то строка уже прочитана системой и исправить ее нельзя.

Можно посмотреть ввод по строкам на примере команды `cat`. Эта команда обычно накапливает или буферизует свой выходной поток, чтобы для повышения эффективности писать большими порциями, но флаг `-u` отключает буферизацию, так что она выдает строку сразу по получении:

```
$ cat                                Выдача команды cat с буферизацией
123
456
789
ctl-d
123
```

```
456
789

$ cat -u                              Выдача команды cat без буферизации
123
123
456
456
789
789
ctl-d
$
```

Команда `cat` получает каждую строку, когда вы нажимаете клавишу *RETURN*; без буферизации она выдает данные, как только их получит.

Теперь попробуем сделать нечто другое: введите несколько символов, а затем вместо *RETURN* наберите на клавиатуре *ctl-d*:

```
$ cat -u
123ctl-dl23
```

Команда cat выдает символы мгновенно. Символ *ctl-d* означает, что нужно немедленно послать символы, введенные с терминала, программе, которая производит ввод с терминала. В отличие от символа перевода строки *ctl-d* не передается программе. Теперь введите второй раз *ctl-d* без каких-либо символов:

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

Интерпретатор отвечает на это выводом приглашения, поскольку команда `cat`, не получив символов, считает, что файл кончился, и прекращает работу. Символ *ctl-d* передает все, что вы ввели, программе, производящей ввод с терминала. Если вы ничего не ввели, программа не получит никаких символов, что соответствует концу файла. Именно поэтому ввод *ctl-d* приводит к выходу из системы — интерпретатор не получает больше входной информации. Конечно, символ *ctl-d* в основном используется как сигнал о конце файла, но он имеет и более общее назначение.

■ Упражнение: Что произойдет, если ввести *ctl-d* редактору ed? Сравните этот случай с вводом команды

```
$ ed < файл
```

2.2 Что хранится в файле?

Формат файла зависит от программ, которые используют его. Типы файла весьма разнообразны, возможно, потому, что существует большое разнообразие программ. Но, поскольку типы файла не определяются файловой системой, ядро не может указать вам тип файла — оно не знает его. Команда `file` делает обоснованную "догадку" (мы вскоре объясним, как это происходит):

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

Здесь показаны четыре типичных файла. Все они связаны с редактором: каталог (`/bin`), в котором находится редактор, двоичный файл или сама программа, готовая к выполнению (`/bin/ed`), входной текст, т. е. операторы языка Си, составляющие программу (`/usr/src/cmd/ed.c`), и страница справочного руководства (`/usr/man/man1/ed.1`).

При определении типа файла команда `file` не обращает внимания на имена (хотя могла бы), поскольку соглашения об именах — это всего лишь соглашения и поэтому на них нельзя полагаться полностью. Например, файлы, оканчивающиеся на `.c`, почти всегда содержат текст программы на языке Си, но ничто не мешает вам создать файл, оканчивающийся на `.c`, с произвольным содержанием. Команда `file` читает первые несколько сотен байтов файла и пытается по ним определить тип последнего. (Как мы покажем позднее, файлы специального системного назначения, такие, как каталоги, могут быть идентифицированы путем запроса системы, но эта команда может определить каталог, читая его.)

Иногда установить тип файла нетрудно. Выполняемая программа помечается вначале двоичным "магическим" числом. Команда `od`, запущенная без всяких флагов, выдает содержимое файла по словам в 16-разрядном или двухбайтовом представлении, и магическое число становится видимым:

```
$ od /bin/ed
0000000 000410 025000 000462 011444 0000000 000000 000000 000001
0000020 170011 016600 000002 005060 1777776 010600 162706 000004
0000040 016616 000004 005720 010066 0000002 005720 001376 020076
...
$
```

Восьмиричное число 410 отмечает готовую к выполнению программу, которую могут разделять несколько процессов. (Конкретные магические числа зависят от системы.) Набор разрядов, представляющий 410, не является символом из множества ASCII, поэтому он не может быть создан непреднамеренно такими программами, как редактор. Но, конечно, вы можете создать подобный файл, запустив свою собственную программу, а система в соответствии с соглашением сочтет его выполняемой программой.

В случае текстовых файлов указание может быть скрыто более глубоко в файле, поэтому команда `file` отыскивает строки, подобные `#include`, чтобы распознать текст программы на Си, или строки, начинающиеся с точки, чтобы распознать входные данные для программ `nroff` или `troff`.

У вас может возникнуть вопрос: почему система не следит за типами файлов более внимательно, ведь тогда, например, программе `sort` в качестве входного потока никогда не попадал бы файл `/bin/ed`. Одна из причин сосстоит в том, чтобы не потерять какие-нибудь полезные для программиста свойства. Хотя команда

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

не имеет особого смысла, существуют команды, которые могут выполняться с любыми файлами, и нет причин ограничивать их возможности. Команды `od`, `cp`, `wc`, `cmp`, `file` и многие другие обрабатывают файлы независимо от их содержания. Но идея безтиповых файлов этим не ограничивается. Если, скажем, для программы `nroff` входные данные отличаются от текста программы на Си, редактор будет вынужден различать их, создавая файл, и, вероятно, считывая файл для редактирования. Тогда, без сомнения, авторам этой книги было бы трудно подготавливать примеры на языке Си для глав 6, 7 и 8.

Система UNIX пытается сгладить все различия. Любой текст состоит из строк, заканчивающихся символом перевода строки, и большинство программ воспринимает такой простой формат. В процессе работы над книгой мы много раз запускали команды для создания текстовых файлов, затем обрабатывали эти файлы с использованием команд, подобных уже рассмотренным, и применяли редактор, чтобы слить файлы во входной поток для форматирующей программы `troff`, с помощью которой подготовлен текст книги. Практически любая страница здесь получена подобным образом:

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

Команда `od` передает текст в стандартный выходной поток, который можно использовать там же, где и сам текст. Такая универсальность непривычна; в большинстве систем имеется несколько форматов файла, даже для текста, и требуется диалог с системой со стороны программы или пользователя, чтобы создать файл определенного типа. В системе UNIX есть только один вид файла, и для доступа к такому файлу достаточно лишь знать его имя. (Существует хороший тест на единообразие системы, предложенный Д. МакИлроем. UNIX легко выдерживает его. Можно ли результат компиляции с Фортрана использовать как входной поток для компилятора с Фортрана? Очень больше число систем не позволяет этого сделать.)

Отсутствие форматов файла дает выигрыш во всем — программисты не должны заботиться о типе файла, а все стандартные программы будут работать с любым файлом. Однако необходимо отметить и недостатки. Программы, которые сортируют, редактируют, проводят поиск, действительно ожидают текст в качестве входного потока: команда grep не может правильно анализировать двоичные файлы, команда `sort` не может сортировать их, и никакой стандартный редактор не может работать с ними.

В большинстве программ, которые ожидают текст в качестве входного потока, существуют ограничения реализации. Мы проверили несколько программ на тексте размером 30 тыс. байт, не содержащем ни одного символа перевода строки, и только некоторые из них работали правильно, поскольку многие программы делают явно неоговариваемые допущения о максимальном размере строки текста (исключительные ситуации см. в разделе **BUGS** (ошибки) описания `sort(1)`).

Нетекстовые файлы, несомненно, имеют свою специфику. Например, для очень больших баз данных обычно нужна дополнительная адресная информация, обеспечивающая быстрый доступ; для повышения эффективности она должна быть представлена в двоичном виде. Но каждому формату файла, не являющемуся текстовым, должно соответствовать свое семейство программ, выполняющее те операции, которые для текстовых файлов производят стандартные средства. Машинная обработка текстовых файлов, возможно,

несколько менее эффективна, но это уравновешивается стоимостью дополнительного программного обеспечения, поддерживающего более сложные форматы. Если вы выбираете формат файла, следует тщательно все продумать, прежде чем остановиться на нетекстовом представлении. (Желательно также предусмотреть вариант, при котором поведение вашей программы будет "осмысленным" в случае длинных входных строк.)

2.3 Каталоги и имена файлов

Все ваши файлы имеют вполне определенные имена, начиная с `/usr/you`, но если у вас есть только файл `junk`, то при задании команды `ls` не выдается `/usr/you/junk`; имя файла выводится без всякого префикса:

```
$ ls
junk
$
```

Это происходит потому, что любая выполняемая программа, т. е. каждый процесс, имеет текущий каталог и неявно предполагается, что все имена файлов начинаются с имени этого каталога, если они явно не начинаются с дробной черты. Таким образом, у интерпретатора `shell`, в который вы вошли, и команды `ls` есть текущий каталог. Команда `pwd` ("print working directory" — печать текущего каталога) выдает имя текущего каталога:

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

Текущий каталог представляет собой атрибут процесса, а не пользователя или программы: пользователям соответствуют начальные каталоги, а процессам — текущие. Если процесс порождает процесс–потомка, последний наследует текущий каталог родителя. Но, если затем потомок сменит текущий каталог, родителя это не затронет: его текущий каталог останется тем же, независимо от действий потомка.

Понятие текущего каталога, конечно, обеспечивает удобство обозначения, поскольку освобождает от излишнего ввода, но настоящее его назначение — организационное. Связанные друг с другом файлы находятся в одном каталоге. Каталог `/usr` обычно является начальным каталогом файловой системы пользователей (`usr` — сокращение от `user`, подобно `cmp`, `ls` и т. д.). Ваш начальный каталог `/usr/you` — это ваш текущий каталог при первом вхождении в систему. Каталог `/usr/src` содержит исходные тексты системных программ, каталог `/usr/src/cmd` — исходные тексты команд UNIX, `/usr/src/cmd/sh` — исходные тексты интерпретатора `shell` и т. д. Всякий раз, когда вы приступаете к новому проекту или когда у вас появляется группа связанных файлов, скажем, набор рецептов, вы можете создать новый каталог с помощью команды `mkdir` и поместить в него файлы.

```
$ pwd
/usr/you
$ mkdir recipes
$ cd recipes
$ pwd
/usr/you/recipes
```

```
$ mkdir pie cookie
$ ed pie/apple
...
$ ed
...
$
```

Заметьте, как легко ссылаться на вложенные каталоги. Файл `pie/apple` имеет очевидный смысл: рецепт яблочного пирога из каталога `/usr/you/recipes/pie`. Вместо этого вы могли бы поместить рецепт, например, в каталог `recipes/ apple.pie`, а не во вложенный каталог в каталоге `recipes`, но лучшее решение — собрать все рецепты пирогов вместе. Так, рецепт крема мог бы храниться в `recipes/pie/crust`, чтобы не дублировать его в рецепте для каждого пирога. Хотя файловая система предоставляет мощное средство организации данных, вы можете забыть, куда помещен файл, и даже какие файлы у вас есть. Естественным решением было бы иметь одну или несколько команд, позволяющих "порыться" в каталогах. Конечно, команда `ls` помогает искать файлы, но не дает возможности исследовать вложенные каталоги:

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

Эту часть файловой системы можно изобразить графически:

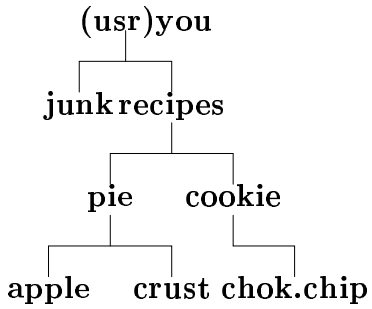


Рис. 2.1: Часть файловой системы

С помощью команды `du` ("disk usage" — использование диска) вы можете выяснить, какое пространство на диске занято файлами каталога, включая все вложенные каталоги:

```
$ du
6   ./recipes/pie
```

```
4   ./recipes/cookie
11  ./recipes
13  .
$
```

Смысл имен файлов понятен; числа соответствуют количеству блоков на диске (обычно размер блока составляет 512 или 1024 байта) для хранения каждого файла. При использовании каталога число показывает, сколько блоков задействовано всеми файлами этого каталога, включая вложенные каталоги и сам каталог.

Команда `du` имеет флаг `-a` ("all" — все), который означает, что требуется выдавать все файлы в каталоге. Если один из них является каталогом, команда `du` сообщает и о нем:

```
$ 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  .
$
```

Выходной поток команды `du -a` можно направить по программному каналу через команду `grep` для поиска каких-либо файлов:

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

Напомним (см. гл. 1), что имя '.' — это запись в каталоге, обозначающая сам каталог; оно обеспечивает доступ к каталогу в тех случаях, когда не известно его полное имя. Команда `du` просматривает файлы в каталоге, причем если вы не указали, в каком именно каталоге следует производить поиск, то она выберет '.', т. е. каталог, с которым вы работаете в данный момент. Значит, `junk` и `./junk` — имена одного и того же файла.

Несмотря на то, что каталоги играют в системе важную роль, они представляются в файловой системе как обычные файлы. Эти каталоги можно читать, но в отличие от традиционных файлов их нельзя создавать и в них нельзя писать. Для сохранения целостности системы и файлов пользователей ядро берет на себя контроль за содержимым каталогов.

Теперь представим содержимое каталога в байтовой форме:

```
$ od -cb
000000  4   ;   .  \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
000020 273  (   .   .  \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
000040 252  ,   р   е   ц   е   п   т   ы  \0  \0  \0  \0  \0  \0
```

```
      252 073 256 243 263 243 255 260 273 000 000 000 000 000 000
000060 230  =   j   u   n   k  \0  \0  \0  \0  \0  \0  \0  \0  \0
      230 075 152 165 156 153 000 000 000 000 000 000 000 000 000
000100
$
```

Видите имена файлов, "спрятанные" здесь? Формат каталога — это комбинация двоичного и текстового представлений. Каталог строится из фрагментов по 16 байт, причем последние 14 байт здесь содержат имя файла, дополненное символом NUL из ASCII (нулевой код, имеющий значение 0), а первые два байта указывают системе, где находится служебная информация, относящаяся к файлу (мы вернемся к этому вопросу позднее). Каждый каталог начинается двумя записями: '.' (точка) и '..' (точка–точка).

```
$ cd                                    Начальный каталог
$ cd recipes
$ pwd
/usr/you/recipes
$ cd ..; pwd                            На один уровень выше
/usr/you
$ cd ..; pwd                            Еще на один уровень выше
/usr
$ cd ..; pwd                            Еще на один уровень выше
/
$ cd ..; pwd                            Еще на один уровень выше
/                                       Выше некуда
$
```

Каталог / называется корнем файловой системы. Каждый файл системы находится в корневом каталоге или в одном из вложенных в него каталогов, и корневой каталог является родителем самому себе.

■ Упражнение: На основании изложенного выше представьте приблизительно действие команды `ls`. Подсказка: `cat . > foo; ls -f foo`.

■ Упражнение: (*Более сложное*). Как действует команда `pwd`?

■ Упражнение: Команда `du` предназначена для учета использования дискового пространства. Осуществлять с ее помощью поиск файлов в иерархии каталогов — довольно странное решение, возможно, даже неподходящее. За альтернативой обратитесь к странице справочного руководства `find(1)` и сравните две команды, в частности `du -a | grep ...` и `find`. Какая из них сработает быстрее? Что лучше: создать новую команду или воспользоваться побочным эффектом уже существующей?

2.4 Права доступа

В отношении каждого файла существуют права доступа, которые определяют, кто и что может делать с файлом. Если вы, допустим, храните свои любовные письма в системе и даже собрали их в отдельный каталог в иерархии каталогов, то, вероятно, вряд ли захотите, чтобы кто-либо мог их прочитать. Поэтому вы можете изменить права доступа к каждому письму во избежание появления слухов (или только к части писем, возбудив тем самым чье-то любопытство). Вы можете, наконец, просто изменить права доступа к каталогу, содержащему письма, и пресечь любые попытки вмешаться в ваши дела.

Однако мы должны предупредить вас: в каждой системе UNIX есть особый пользователь, называемый суперпользователем, который может читать или изменять любой файл в системе. Привилегии входа в систему суперпользователю обеспечивает специальное имя root (корень). Это имя используется администраторами системы для выполнения работ по ее поддержанию. Существует команда `su`, которая гарантирует вам статус суперпользователя при условии, что вы знаете пароль при входе под именем root. Таким образом, всякий, кто знает пароль суперпользователя, может читать ваши любовные письма, поэтому не стоит хранить в системе частную информацию.

Если нужна большая защищенность, то можно изменить данные в файле так, что даже суперпользователь не сможет прочесть (или, по крайней мере, понять) их, для чего служит команда `crypt` (см. `crypt(1)`). Конечно, даже эта команда не дает абсолютной защиты. Суперпользователь может изменить команду `crypt`, или кто-то попытается расшифровать алгоритм `crypt`. Правда, и то и другое является трудно выполнимым, так что практически команда `crypt` достаточно надежна.

Большинство нарушений защиты происходит из-за паролей, которые выдаются или легко отгадываются. Изредка по недосмотру администратора системы пользователь-злоумышленник получает права суперпользователя. Вопросы безопасности более подробно обсуждаются в статьях, которые указаны в списке литературы в конце главы.

При входе в систему вы вводите имя и подтверждаете, что это вы, задавая пароль. Имя представляет собой ваш входной идентификатор, или `login-id`. На самом деле, система распознает вас по числу, называемому идентификатором пользователя, или `uid`. В действительности различным `login-id` может соответствовать один `uid`, что делает их неразличимыми для системы, хотя такое бывает относительно редко, и, по всей видимости, является нежелательным по соображениям безопасности. Кроме `uid`, вам приписывается индентификатор группы, или `group-id`, который относит вас к определенной группе пользователей. Во многих системах обычных пользователей (в отличие от тех, кто имеет `login-id` типа root) объединяют в одну группу под именем other (другие), но в вашей системе может быть иначе. Файловая система, а значит, и вся система UNIX в целом определяет ваши возможности исходя из прав доступа, предоставляемых вашему `uid` и `group-id`.

Файл `/etc/passwd` — это файл паролей; он содержит всю информацию, связанную со входом каждого пользователя в систему. Подобно системе, вы можете определить свой `uid` и `group-id`, если найдете свое имя в `/etc/passwd`:

```
$ grep you /etc/passwd
you:gkmbCTrJ04COM:604:1:Y.O.A.People:/usr/you:
$
```

Поля в файле паролей разделяются двоеточием и расположены следующим образом (как видим из `passwd(5)`):

```
login-id:зашифрованный_пароль:uid:group-id:разное:начальный_каталог:shell
```

Файл паролей представляет собой обычный текстовый файл, но назначение и разделитель полей определяются по соглашению между программами, работающими с информацией этого файла. Поле `shell` обычно пустое; значит, по умолчанию используется стандартный интерпретатор `/bin/sh`. Поле "разное" может содержать что угодно (как правило, ваше имя, адрес или телефон).

Заметьте, что ваш пароль присутствует здесь во втором поле, но в зашифрованном виде. Файл паролей могут прочесть все (вы только что это сделали), и если ваш пароль бы там, то любой, кто пожелает, может выдать себя за вас. Когда вы вводите свой пароль при входе в систему, он шифруется, и результат сравнивается с зашифрованным паролем из `/etc/passwd`. Если они совпадают, то вам разрешают войти. Этот механизм работоспособен, потому что алгоритм шифрации таков, что позволяет легко перейти от раскрытой формы к зашифрованной, тогда как обратный переход очень труден. Например, если ваш пароль `ka-boom`, он может быть зашифрован как `gkmbCTrJ04COM`, но, получив последний, вам будет нелегко вернуться к оригиналу.

Ядро решает, что вам можно позволить читать файл `/etc/passwd` исходя из прав доступа, связанных с файлом. Для каждого файла предусмотрены три вида прав доступа: чтение (т. е. исследование его содержимого), запись (т. е. изменение его содержимого) и выполнение (т. е. запуск его как программы). Далее, разным пользователям могут быть предоставлены различные права доступа. Как владелец файла вы имеете один набор прав на чтение, запись и выполнение. У "вашей" группы — другой набор прав доступа, у всех остальных — третий набор.

Команда `ls` с флагом `-l` сообщает среди прочего права доступа:

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

Информацию, содержащуюся в двух строках вывода команды `ls`, можно интерпретировать так: владельцем файла `/etc/passwd` является пользователь с `login-id`, равным root; его группа называется adm; размер файла 5115 байт; последний раз изменен был 30 августа в 10:40; файл имеет единственную связь, т. е. одно имя в файловой системе (вопрос о связях мы обсудим в следующем разделе). Некоторые варианты команды `ls` выдают имена владельца и группы сразу при однократном вызове.

Строка `-rw-r-r-` показывает, как представляет права доступа к файлу команда `ls`. Первый дефис (`-`) означает, что это обычный файл. В случае каталога на его месте стояла бы буква `d`. Следующие три символа обозначают права владельца файла на чтение, запись и выполнение (исходя из `uid`). Строка `rw-` свидетельствует о том, что владелец (root) может читать, писать, но не выполнять файл. В случае выполняемого файла дефис был бы заменен символом `x`.

Три символа (`r-`) обозначают права доступа группы, в данном случае пользователей из группы adm — по-видимому, системных администраторов, которые могут читать файл, но не писать и не выполнять его. Следующие три символа (также `r-`) определяют права доступа для всех остальных пользователей системы. Таким образом, на данной машине только root может изменить информацию по входу в систему для пользователя, но прочесть файл и узнать эту информацию может любой. Разумным был бы вариант, при котором группа adm также имела бы право на запись в файл `/etc/passwd`.

Файл /etc/group хранит в зашифрованном виде имена групп и их group-id и определяет, какие пользователи входят в какие группы. В файле /etc/passwd определяется только ваша группа при входе в систему; команда newgrp изменяет ее права доступа на права другой группы.

Кто угодно может задать:

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

и редактировать файл паролей, но только root может записать измененный файл. Поэтому вполне правомочен вопрос: как изменить свой пароль, если это требует редактирования.файла паролей. Программа, изменяющая пароли, называется passwd, вероятно, вы найдете ее в /bin:

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

(Обратите внимание на то, что /etc/passwd — текстовый файл, содержащий информацию по входу в систему, тогда как /bin/passwd находится в другом каталоге, содержит программу, готовую к выполнению, и позволяет изменить данные, связанные с паролем). Права доступа к этому файлу показывают, что выполнить команду может кто угодно, но изменить команду passwd — только root. Буква s вместо x в поле прав на выполнение для владельца файла означает, что при выполнении команды ей предоставляются права, соответствующие праву владельца файла, в данном случае root. Поскольку файл /bin/passwd имеет такой признак установки uid и при выполнении получает права root, любой пользователь, выполняя команду passwd, может редактировать файл /etc/passwd.

Введение признака установки uid — простое элегантное решение целого ряда проблем безопасности (Признак установки uid введен Д. Ритчи). Например, автор игровой программы может установить свой uid для программы, поэтому она сможет изменять файл с результатами игр, который защищен от доступа со стороны других пользователей. Но идея введения признака установки uid потенциально опасна. Программа /bin/passwd должна быть правильной, иначе она может уничтожить системную информацию под прикрытием суперпользователя root. При наличии прав доступа -rwsrwxrwx ее мог бы переписать любой пользователь, и, таким образом, заменить файл на неработоспособную программу. Это особенно опасно для программ, обладающих признаком установки uid, поскольку root имеет доступ к каждому файлу, системы. (В некоторых системах UNIX происходит отключение признака установки uid всякий раз, когда файл изменяется, что уменьшает вероятность нарушения защиты).

Признак установки uid — мощное средство, но оно используется в основном для нескольких системных программ, таких, как passwd. Рассмотрим более типичный файл:

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

Этот файл доступен для выполнения всем, а писать в него могут только root и пользователь той же группы. Слова “доступен для выполнения” означают, что при вводе

```
$ who
```

интерпретатор shell просматривает ряд каталогов, в том числе /bin, отыскивая файл с именем who. Если такой файл найден и он имеет право доступа на выполнение, то shell обращается к ядру для его запуска. Ядро проверяет права доступа, и, если они действительны, запускает программу. Отметим, что программа — это просто файл с правом доступа на выполнение. В следующей главе вы познакомитесь с программами, являющимися обычными текстовыми файлами, но они могут выполняться как команды, поскольку имеют право доступа на выполнение.

Права доступа к каталогам действуют несколько иначе, но основной принцип остается тем же:

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

Команда ls с флагом -d сообщает скорее о самом каталоге, чем о его содержимом, и первая буква d в выводе означает, что '.' в действительности является каталогом. Поле r показывает, что можно читать каталог, поэтому с помощью команды ls (или od для данного случая) можно выяснить, какие файлы хранятся в нем. Буква w свидетельствует о том, что можно создавать и исключать файлы из каталога, поскольку это требует изменения, а значит, записи в файл каталога.

На самом деле нельзя просто писать в каталог, даже суперпользователю root это запрещено:

```
$ who > .                          Попытка затереть '.'
.: cannot create                   Нельзя
$
```

Существуют системные обращения, которые создают и удаляют файлы, и только с их помощью можно изменить содержимое каталога. Но принцип прав доступа применим и для них: поле w показывает, кто может использовать системные функции для изменения каталога.

Право на удаление файла не зависит от самого файла. Если у вас есть право записи в каталог, вы можете удалять файлы из него, причем даже те, которые защищены от записи. Команда rm все-таки запрашивает подтверждение, прежде чем удалить защищенный файл, чтобы убедиться, что вы действительно хотите это сделать, — редкий для команд системы UNIX случай двойной проверки намерений пользователя. (Флаг -f команды rm обеспечивает удаление файлов без запроса.)

Поле x в случае каталога означает не выполнение, а поиск. Право на выполнение определяет возможность поиска файла в каталоге. Поэтому возможно создать каталог с правом доступа "x" для других пользователей, предполагая, что пользователи будут иметь доступ к любому известному им файлу в каталоге, но не смогут выполнять команду ls или читать каталог, чтобы узнать, какие файлы в нем находятся. Аналогично каталог с правом доступа r- можно читать (с помощью ls), но нельзя работать с его файлами. В некоторых системах используют это свойство, чтобы закрыть каталог /usr/games в рабочее время.

Команда chmod ("change mode" — изменить режим) меняет права доступа к файлам:

```
$ chmod права_доступа имена файлов...
```

Синтаксис конструкции “права_доступа”, к сожалению, громоздкий. Она может определяться двумя способами: с помощью восьмиричных чисел и последовательностью символов. Проще использовать восьмиричные числа, хотя иногда более удобными оказываются символьные обозначения, так как с их помощью можно показать, какие изменения произошли в правах доступа. Лучше, конечно, было бы задать

```
$ chmod rw-rw-rw- junk                    Так нельзя!
```

чем вводить

```
$ chmod 666 junk
```

но так не получается. Восьмиричное значение режима складывается из значений прав доступа: 4 — для чтения, 2 — для записи и 1 — для выполнения. Три цифры, как и в выводе команды `ls`, показывают права доступа для владельца, группы и всех остальных. Символьные обозначения объяснить труднее; их точное описание приводится в справочном руководстве `chmod(1)`. Для наших же целей достаточно указать, что “+” устанавливает право доступа, а “-” лишает его. Например,

```
$ chmod +x command
```

позволяет всем выполнять команду, а

```
$ chmod -w file
```

лишает всех права записи в файл, включая и владельца файла. Если не принимать во внимание существование суперпользователя, то только владелец файла может изменить права доступа к файлу, независимо от текущих прав доступа. В том случае, когда кто-то еще предоставил вам право записи в файл, система не позволит изменить код прав доступа к файлу:

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

Но, если каталог не защищен от записи, пользователи могут удалять файлы, несмотря на отсутствие права доступа к файлу. Если вы хотите быть уверенным в том, что ни вы, ни ваши друзья никогда не удалят файлы из каталога, отмените право на запись для каталога:

```
$ cd
$ date > temp
$ chmod -w .                              Закрыть каталог по записи
$ ls -ld .
dr-xr-xr-x 3 you 80 Sep 27 11:48 .
$ rm temp
rm: temp not removed                      Нельзя удалить файл
$ chmod 775 .                             Восстановление прав доступа
$ ls -ld .
drwxrwxr-x 3 you 80 Sep 27 11:48 .
$ rm temp                                 Теперь можно
$
```

Файл `temp` теперь удален. Обратите внимание на то, что изменение прав доступа к каталогу не меняет дату последней модификации файла. Дата модификации отражает изменение содержимого файла, а не прав доступа к нему. Права доступа и даты хранятся не в самом файле, а в системной структуре данных, называемой индексным дескриптором (речь о нем пойдет в следующем разделе).

■ УПРАЖНЕНИЕ: Поэкспериментируйте с командой `chmod`. Попробуйте разные простые варианты типа 0 или 1. Будьте осторожны, чтобы не испортить свой начальный каталог.

2.5 Индексные дескрипторы

Файл имеет несколько атрибутов: имя, содержимое и служебную информацию (права доступа и даты модификации). Служебная информация размещается в индексном дескрипторе вместе с важной системной информацией, такой, как размер файла, место хранения его на диске и т. д.(Индексный дескриптор обозначается как `inode` ("index node") или `i-node`. — Прим. перев.) В индексном дескрипторе хранятся три даты: время последнего изменения файла (записи в него), время последнего использования файла (чтение или выполнение), время последнего изменения самого индексного дескриптора, например изменения 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 видно из результата действия команды `ls -lu`, изменение содержимого файла не влияет на дату последнего использования, а с изменением прав доступа связана только дата изменения индексного дескриптора, о чем выдается сообщение командой `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
$
```

Можно использовать флаг `-t` команды `ls`, который применяется для сортировки файлов по времени (по умолчанию принимается время последней модификации), совместно с флагами `-c` или `-r`, чтобы узнать порядок, в котором изменились индексные дескрипторы или читались файлы:

```
$ ls recipes
apple
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
```

У каталога `recipes`, как вы видите, более позднее время использования, поскольку мы только что просмотрели его содержимое.

Очень важное понять значение индексного дескриптора, причем не для того, чтобы оценить действие флагов команды `ls`. По существу, индексные дескрипторы и есть файлы. Иерархия каталогов предоставляет только удобный способ именования файлов. Внутреннее системное имя файла или индекс файла — это номер индексного дескриптора, содержащего информацию о файле. Команда `ls -i` выдает индекс файла в десятичной форме:

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

Именно индекс файла хранится в первых двух байтах каталога, предшествующих имени. Команда `od -d` выдает информацию не в восьмиричной форме по байтам, а в десятичной, объединив по два байта в одно целое, и поэтому мы увидим на экране индекс файла:

```
$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   ;   p   e   ц   п   т   ы  \0  \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 в каждой строке каталога являются единственной связью между именем файла и его содержимым. Именно поэтому имя файла в каталоге называется связью: оно связывает имя в иерархии каталогов с индексным дескриптором и, тем самым, с информацией. Один и тот же индекс файла может появиться в нескольких

каталогах. Команда rm в действительности удаляет не индексный дескриптор, а строку каталога или связь. Только когда последняя связь файла исчезает, система удаляет индексный дескриптор, а значит, и сам файл.

Если индекс файла в строке каталога равен нулю, это означает, что связь удалена, но сам файл не обязательно удален — могут существовать связи где–нибудь еще. Можно убедиться в том, что индекс файла становится равным нулю 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 файл, создаваемый в этом каталоге, займет освободившуюся позицию, хотя у него, вероятно, будет другой индекс. Команда `ln`, имеющая приведенный ниже синтаксис, устанавливает связь с уже существующим файлом:

```
$ ln old-file new-file
```

Назначение связи состоит в том, чтобы дать два имени одному и тому же файлу, поэтому он часто оказывается в двух разных каталогах. Во многих системах есть связь с редактором `/bin/ed` под названием `/bin/e`, так что пользователи могут вызывать редактор как `e`. Две связи одного файла указывают на одну и ту же запись, а значит, имеют один и тот же индекс файла:

```
$ 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
$
```

Целое число, выдаваемое между правом доступа и именем владельца файла, является числом связей файла. Поскольку каждая связь ссылается на индексный дескриптор, все связи одинаково, важны — нет разницы между первой связью и последующими. (Заметим, что общий объем занимаемого на диске пространства, сообщаемый командой `ls`, вычисляется неверно из-за двойного подсчета).

Если изменить файл, то изменение обнаружится при обращении к файлу под любым из его имен, так как все связи ссылаются на один файл:

```
$ echo x > junk
$ ls -l
total 3
-rw-rw-rw- 2 you  2 Sep 27 12:37 iunk
```

```
-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 файла `linktojunk` число связей опять становится равным единице. Как уже отмечалось, при удалении файла уничтожается лишь связь. Сам же файл сохраняется до тех пор, пока не будет удалена последняя связь. На практике, конечно, большинство файлов имеет единственную связь, но тем не менее вы убедились в том, что можно простым способом повысить гибкость системы.

Предостережение тем, кто слишком торопится: после удаления последней связи данные становятся недоступными. Удаленные файлы попадают скорее в топку, чем в мусорную корзину, и нет средства возродить их из пепла. (Слабая надежда на возрождение все-таки есть. В больших системах UNIX есть рутинная функция дублирования, которая периодически копирует изменяемые файлы в какое–нибудь надежное место типа магнитной ленты, откуда их можно извлечь. Для собственного успокоения вам следует знать, какой объем дублирования обеспечивает ваша система. Если нет ничего, будьте бдительны — любые неполадки с дисками могут обернуться катастрофой.)

Связи файла удобны, если два пользователя совместно работают с ним, но иногда нужна на самом деле отдельная копия — другой файл с той же информацией. Например, вы можете скопировать документ до внесения в него существенных изменений, так что можно будет воссстановить оригинал, если вас не устроят эти изменения. Здесь не поможет установление связи, так как при изменении данных обе связи будут ссылаться на измененный файл. Копирует файлы команда `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
$
```

Индексы файлов `junk` и `copyofjunk` различны, поскольку это различные файлы, хотя в данный момент они имеют одинаковое содержимое. Существует полезный прием: можно изменить права доступа к копии файла, чтобы ее труднее было случайно удалить.

```
$ chmod -w copyofjunk                          Убрать право записи
$ 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                      Нельзя! Он нужен
```

```
$ 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                          А может быть, и не так нужен
$ 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
$
```

Изменение копии файла или ее удаление не оказывает действия на оригинал. Обратите внимание на то, что, поскольку у файла `copyofjunk` отменено право на запись, команда `rm` запрашивает подтверждение, прежде чем удалить файл.

Есть еще одна команда общего назначения, управляющая файлами, — `mv`, которая переносит или переименовывает файлы, просто преобразуя связи. Синтаксис ее такой же, как у команд `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
$
```

`sameoldjunk` — это тот же самый файл, что и наш старый файл `junk`, вплоть до индекса файла, который связан с записью каталога с номером 15768; изменилось только его имя.

Все описанные выше манипуляции с файлами происходили в одном каталоге, однако команды применяются и в других каталогах. Команда `ln` часто используется для того, чтобы установить связь с одним именем в разных каталогах; это бывает в тех случаях, когда несколько пользователей работают с одной программой или над одним документом. Команда `mv` может переслать файл или каталог из одного каталога в другой. На самом деле, это довольно стандартный прием, так что у команд `mv` и `cp` есть специальный синтаксис для такой ситуации:

```
$ mv (или cp) file1 file2 ... directory
```

Здесь показана пересылка (или копирование) одного или нескольких файлов в каталог, который является последним аргументом. Связи или копии файлов получаются под теми же именами. Например, если вы хотите “набить руку” в работе с редактором, то можете начать с

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

получив свою копию исходного текста редактора для экспериментов. Осваивая интерпретатор `shell`, исходные тексты которого находятся в нескольких каталогах, задайте

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

и команда `cp` скопирует все исходные тексты `shell` в ваш вложенный каталог `sh` (мы считаем, что в `/usr/src/cmd/sh` нет вложенных каталогов, так как команда `cp` не слишком "умна"). В некоторых случаях команду `ln` допустимо применять с несколькими именами файлов в качестве аргументов, но имя каталога по-прежнему является последним аргументом. В ряде систем команды `mv`, `cp` и `ln` сами служат связями, ссылающимися на один файл, который анализирует имя команды, чтобы узнать, какое задание выполнить.

■ Упражнение: Почему команда `ls -l` выдает четыре связи у каталога `recipes`?
Подсказка: попробуйте ввести

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

Чем эта информация полезна?

■ Упражнение: В чем состоит разница между

```
$ mv junk junk1 и
$ cp junk junk1
$ rm junk
```

Подсказка: установите связь с `junk` и затем используйте ее.

■ Упражнение: Команда `cp` не производит копирования во вложенных каталогах, а ограничивается файлами первого уровня вложенности. Каковы будут ее действия, если один из аргументов окажется каталогом? Насколько это хорошо и осмысленно? Обсудите возможные преимущества трех вариантов: включить еще один флаг в `cp`, чтобы работать с вложенными каталогами, ввести отдельную команду `rcp` (рекурсивную cp) для данного случая или просто предъявить к `cp` требование копировать все файлы из каталога, если он встретится среди аргументов (см. гл. 7). Что получат другие программы, если они смогут перемещаться по дереву каталогов?

2.6 Иерархия каталогов

В первой главе рассмотрение иерархии файловой системы, начиная с каталога `/usr/you`, носило несколько неформальный характер. Теперь мы хотим изучить ее последовательно, начиная от корня дерева. Корневой каталог называется `/`:

```
$ ls /
bin
boot
```

```
dev
etc
lib
tmp
unix
usr
$
```

Программа `/unix` — это программа ядра UNIX: когда система начинает работу, `/unix` считывается с диска в память и начинает выполняться. Все происходит за два шага: вначале считывается файл `/boot`, а затем он считывает `/unix`. Более подробно о таком процессе раскрутки можно узнать в справочном руководстве по `boot(8)`. Остальные файлы каталога `/`, по крайней мере в нашей версии, являются каталогами, каждый из которых представляет законченный раздел файловой системы. После дальнейшего краткого обзора иерархии читателю будет предоставлена возможность поэкспериментировать с упоминаемыми здесь каталогами. Чем лучше вы разберетесь в устройстве файловой системы, тем более эффективно сможете ею пользоваться. В табл. 2.1 указаны подходящие места для поиска, хотя некоторые имена каталогов зависят от системы.

Каталог `/bin` вам уже известен: в нем находятся основные программы типа `who` или `ed`.

Каталог `/dev` (device — устройства) мы обсудим в следующем разделе.

Каталог `/etc` (et cetera — и т. д.) также уже вам встречался ранее. В нем находится различная служебная информация, например файл паролей, и некоторые системные программы, такие, как `/etc/getty`, которая инициирует связь с терминалом для команды `/bin/login`, `/etc/rc` — это файл команд, выполняющихся после раскрутки системы. В файле `/etc/group` содержатся сведения о составе всех групп.

Каталог `/lib` (library — библиотека) включает основные части компилятора языка Си, такие, как `/lib/cpp` — препроцессор Си, `/lib/libc.a` — библиотека стандартных функций Си.

Каталог `/tmp` (temporaries — временное) представляет собой хранилище для временных файлов, создаваемых при выполнении программы.

Например, когда вы вызываете редактор, он создает файл с именем типа `/tmp/e00512`, что позволяет иметь свою копию редактируемого файла, а не работать с оригиналом. Редактор мог бы, конечно, создать копию в вашем текущем каталоге, но есть причина для преимущественного использования `/tmp`: хотя это и маловероятно, в вашем каталоге уже мог присутствовать файл `e00512`. Далее каталог `/tmp` автоматически очищается при запуске системы, так что в случае системной аварии в вашем каталоге не появится ненужный файл. Часто каталог `/tmp` организуется на диске для обеспечения быстрого доступа к нему. Однако здесь возникает проблема: если сразу несколько программ создают файлы в каталоге `/tmp`, их файлы могут перепутаться. Именно поэтому редактор `ed` выбирает особое имя; оно построено таким образом, чтобы никакая другая программа не могла выбрать то же имя для временного файла. В гл. 5 и 6 будет показан способ достижения этого.

Каталог `/usr` называется файловой системой пользователей, хотя он может быть мало связан с файлами настоящих пользователей системы. На своей машине мы используем исходные каталоги `/usr/bwk` и `/usr/rob`, но у вас часть иерархии, начинающаяся с `/usr`, может быть другой. Независимо от того, находятся ли ваши файлы в каталоге, вложенном в `/usr`, вы всегда найдете в нем что-нибудь интересное (если нет местной специфики). Так

/	Корень файловой системы
/bin	Основные программы, готовые к выполнению (двоичные)
/dev	Файлы устройств
/etc	“Разное” системы
/etc/motd	Сегодняшнее сообщение при входе в систему
/etc/passwd	Файл паролей
/lib	Основные библиотеки и т. п.
/tmp	Временные файлы; обновляется при запуске системы
/unix	Операционная система в форме, готовой к выполнению
/usr	Файловая система пользователей
/usr/adm	Системная служба: справочная информация и т. п.
/usr/bin	Команды для пользователей: troff и т. п.
/usr/games	Игровые программы
/usr/include	Файлы определений Си-программ, например math.h
/usr/include/sys	Системные файлы определений Си-программ, например inode.h
/usr/lib	Библиотеки для Си, Фортрана и т. п.
/usr/man	Диалоговое справочное руководство
/usr/man/man1	Страницы справочного руководства раздела 1
/usr/mdec	Диагностика ошибок аппаратуры, программы раскрутки и т.п.
/usr/news	Служба сообщений пользователей
/usr/pub	“Всякая всячина”: см. ascii(7) и eqnchar(7)
/usr/src	Исходные тексты служебных функций и библиотек
/usr/src/cmd	Исходные тексты команд из /bin и /usr/bin
/usr/src/lib	Исходные тексты библиотечных функций
/usr/spool	Рабочий каталог для взаимодействующих программ
/usr/spool/lpd	Временный каталог для печатающего устройства
/usr/spool/mail	Почтовые ящики
/usr/spool/uucp	Рабочий каталог программ uucp
/usr/sys	Исходный текст ядра операционной системы
/usr/tmp	Альтернативный временный каталог (редко используется)
/usr/you	Ваш начальный каталог
/usr/you/bin	Ваши собственные программы

Таблица 2.1: Интересные каталоги (см. также hier(7))

же, как и в каталоге /, здесь есть каталоги с именами /usr/bin, /usr/lib и /usr/tmp. Эти каталоги имеют назначение, сходное со своими тезками в каталоге /, но содержат программы, менее критичные для системы. Например, программа nroff обычно находится в /usr/bin, а не в /bin, библиотеки компилятора с Фортрана располагаются в /usr/lib. Правда, “критичными” для разных систем считаются разные программы. Некоторые системы, такие, как широко распространенная седьмая версия, все программы хранят в /bin, не имея дела с /usr/bin. В других системах каталог /usr/bin разбивается на два каталога в зависимости от частоты использования.

Кроме того, в /usr есть каталог /usr/adm со справочной информацией и /usr/dict, содержащий небольшой словарь (см. spell(1)). Диалоговое справочное руководство хранится в /usr/man (см. в качестве примера /usr/man/man1/spell.1). Если в вашей системе имеются исходные тексты, вы, вероятно, найдете их в /usr/src.

Вероятно, целесообразно потратить немного времени на исследование файловой системы, особенно каталога `/usr`, чтобы вам было понятно, как она устроена и где что можно найти.

2.7 Файлы устройств

В нашем кратком обзоре мы пропустили каталог `/dev` по той причине, что файлы в нем дают хорошее общее представление о файлах вообще. Как можно догадаться по его названию, этот каталог содержит файлы устройств ("device" — устройство).

К привлекательным чертам системы UNIX относится форма ее работы с периферийными устройствами: дисками, магнитными лентами, принтерами, терминалами и т. п. Вместо того чтобы иметь специальные системные программы, например программу чтения с магнитной ленты, достаточно создать файл с именем `/dev/mt0` (опять-таки местные соглашения могут различаться). В ядре обращения к этому файлу преобразуются в машинные команды обращения к магнитной ленте, как если бы программа читала `/dev/mt0`, выдавая содержимое магнитной ленты, подключенной к устройству. Например, команда

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

копирует содержимое магнитной ленты в файл `junk`. Команда `cp` не имеет понятия о специфике файла `/dev/mt0`; для нее он является обычным файлом, т. е. просто последовательностью байтов.

Файлы устройств в чем-то подобны зверинцу, где каждая особь чем-нибудь отличается от остальных, но основные характеристики применимы ко всем. Ниже приведен сокращенный список нашего 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 fcmem
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 root  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
$
```

Первое, что здесь бросается в глаза, это то, что вместо количества байтов указывается пара небольших целых чисел, а в первой позиции прав доступа используется 'b' или 'c'. В таком виде команда `ls` выдает информацию из индексного дескриптора для файла устройств, но не для обычного файла. Обычному файлу предназначен хранимый в

индексном дескрипторе список блоков памяти диска, в которых находится содержимое файла. В случае же файла устройств индексный дескриптор содержит внутреннее имя устройства, включающее его тип (символьное `c` или блочное `b`) и пару чисел, называемых верхним и нижним числами устройства. К блочным устройствам относятся диски и магнитные ленты, а все остальное: терминалы, принтеры, линии сетевой связи и т. п. — к символьным. Верхнее число устройства обозначает его тип, а нижнее характеризует различные экземпляры устройств одного типа. Например, `/dev/tty0` и `/dev/tty1` — это два порта одного контроллера терминала, поэтому они имеют одно и то же верхнее число и различные нижние числа.

Файлы для дисков обычно именуются в соответствии с тем вариантом оборудования, которое представлено в системе. Файлы `/dev/rp00` и `/dev/rp01` названы так потому, что в системе используются дисковые накопители DEC RP06. Есть только один дисковый накопитель, логически поделенный на две файловые системы. Если бы существовал еще один накопитель, связанные с ним файлы имели бы имена `/dev/rp10` и `/dev/rp11`. Первая цифра обозначает номер накопителя, а вторая показывает, какая его часть используется.

У вас может возникнуть вопрос: почему существует несколько дисковых файлов устройств, а не одно? Исторически так сложилось (и для удобства поддержания), что файловая система была разделена на подсистемы. Файлы в подсистеме доступны через каталог главной системы. Программа `/etc/mount` показывает соответствие между файлами устройств и каталогами:

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

В нашем случае каталог `root` находится на `/dev/rp00` (хотя команда `/etc/mount` об этом не сообщает), а файловая система пользователей, т. е. файлы из каталога `/usr` и вложенных каталогов, находится на `/dev/rp01`.

Каталог `/root` должен быть доступен системе для выполнения команд. Каталоги `/bin`, `/dev` и `/etc` всегда находятся в корневом каталоге, поскольку при запуске системы доступны только файлы корневого каталога, а такие, как `/bin/sh`, необходимы для работы. Во время раскрутки системы все файловые системы проверяются на целостность (см. `icheck(8)` или `fsck(8)`) и подключаются к корню иерархии файлов. Эта операция подключения называется присоединением и является программистским эквивалентом операции установки пакета дисков на накопитель; обычно она выполняется только суперпользователем. После присоединения `/dev/rp01` в качестве `/usr` файлы пользователей становятся доступными, как если бы они были частью корневого каталога.

Для обычного пользователя детали операции присоединения подсистемы файлов представляют мало интереса, но здесь есть несколько моментов, относящихся к нашей теме. Во-первых, поскольку подсистемы файлов могут быть присоединены и отсоединены, недопустимо устанавливать связь с файлом из другой подсистемы. Например, нельзя связать программы из общего каталога `/bin` с какими-то файлами из каталогов `/bin` пользователей, поскольку `/usr` находится в иной подсистеме файлов, чем `/bin`:

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

Проблема могла возникнуть и потому, что значения индексных дескрипторов иногда совпадают в различных файловых системах.

Далее, каждая подсистема ограничена по размеру (числу доступных блоков для файлов) и числу индексных дескрипторов. Если подсистема заполнена, то невозможно расширять файлы в такой системе, пока не будет добавлено какое-то пространство. Команда `df` ("disc free space" — свободное пространство диска) выдает сообщение о доступном пространстве в присоединенной подсистеме файлов:

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

В каталоге `/usr` имеется 21257 свободных блоков. Достаточно ли этого пространства или наступил кризис, зависит от того, как система используется; в одних случаях требуется больше свободного пространства, в других — меньше. Кстати, из всех команд `df`, вероятно, обеспечивает наибольшее разнообразие в формате вывода. Результат действия вашей команды `df` может выглядеть совершенно иначе.

Рассмотрим теперь некоторые более общие понятия. При входе в систему вы устанавливаете связь вашего терминала с системой и, значит, получаете в каталоге `/dev` файл, через который передаются вводимые и принимаемые вами символы. Команда `tty` сообщает, какой терминал вы используете:

```
$ 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
$
```

Заметьте, что вы владелец устройства и только у вас есть право на чтение с него. Иными словами, никто не может непосредственно читать вводимые вами символы, но выводить на ваш терминал может любой. Во избежание этого можно изменить права доступа к устройству, запретив тем самым другим использовать программу `write` для прямой записи или просто воспользоваться командой `mesg`.

```
$ mesg n                                    Запретим сообщения
$ ls -l /dev/tty0
crw--w---- 1 you 1, 12 Sep 28 02:41 /dev/tty0
$ mesg y                                    Разрешим
$ ls -l /dev/tty0
crw--w--w- 1 you 1, 12 Sep 28 02:42 /dev/tty0
$
```

Часто бывает удобно использовать имя для ссылки на применяемый терминал, но трудно определить, каково имя вашего терминала. Имя устройства `/dev/tty` является синонимом имени терминала, с которого вы вошли в систему, с каким бы терминалом вы ни работали на самом деле:

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

Имя `/dev/tty` особенно полезно, если программе необходимо начать диалог с пользователем, в то время когда ее стандартный входной и выходной потоки связаны с файлами, а не с терминалом. Команда `crypt` является одной из команд, использующих имя `/dev/tty`. "Открытый" текст поступает из стандартного входного потока, а зашифрованная информация направляется в стандартный выходной поток, поэтому команда `crypt` читает ключ для шифрования с `/dev/tty`:

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

В данном примере имя `/dev/tty` используется неявно, но все-таки используется. Если бы команда `crypt` читала ключ из стандартного входного потока, она бы прочла первую строку из файла `cleartext`. Вместо этого она открывает файл `/dev/tty`, отключает автоматическое эхо вводимых символов, чтобы ваш ключ не появился на экране, и читает ключ. В гл. 5 и 6 приводится несколько других примеров использования `/dev/tty`.

Иногда вы хотите запустить программу, но вам не важен результат ее выполнения. Например, вы могли уже ознакомиться с сегодняшними новостями и не желаете читать их еще раз. Переключение вывода команды `news` в файл `/dev/null` приведет к игнорированию выходного потока:

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

Информация, направляемая в `/dev/null`, просто пропадает, а программы, читающие из этого файла, сразу получают символ конца файла, поскольку программа чтения всегда возвращает 0 прочитанных байтов.

Обычно файл `/dev/null` используют, чтобы отказаться от стандартного выходного потока и сделать видимыми диагностические сообщения. Например, команда `time` (time(1)) сообщает об использованном программой процессорном времени. Результат выдается в стандартный поток диагностики, так что можно хронометрировать команды, производящие преобразование входного потока в выходной, переключая стандартный выходной поток в файл `/dev/null`:

```
$ ls -l /usr/diet/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
$
```

Команда `time` выдает прошедшее календарное время, время процессора, затраченное программой, и время процессора, затраченное ядром системы для выполнения запросов программы. Команда `egrep` — это мощный вариант команды `grep`, который мы обсудим в гл. 4; она выполняется почти в два раза быстрее команды `grep` при просмотре больших файлов. Если бы выдача команд `egrep` или `grep` не была переключена в `/dev/null` или текущий файл, пришлось бы ждать, пока сотни тысяч символов "пробегут" на экране, прежде чем появятся нужные нам временные характеристики.

■ Упражнение: Познакомьтесь с другими файлами каталога `/dev`, прочитав разд. 4 справочного руководства. В чем состоит разница между `/dev/mt0` и `/dev/rmt0`? Прокомментируйте возможную пользу применения вложенных каталогов в `/dev` для дисков, магнитных лент и т. п.

■ Упражнение: Магнитные ленты, записанные в других системах, обычно имеют другие размеры блоков, такие, как 800 байт — десятикратный образ перфокарты из 80 символов, но устройство `/dev/mt0` предполагает блоки из 512 байт. Обратитесь к команде `dd` (`dd(1)`), чтобы узнать, как читать такую ленту.

■ Упражнение: Почему `/dev/tty` не является просто связью с терминалом, с которого вы вошли в систему? Что бы произошло, если бы права доступа для него были `rw-w-w-`, как на вашем терминале?

■ Упражнение: Как работает `write(1)`?
Подсказка: см. в `utmp(5)`.

■ Упражнение: Как узнать, использует ли человек терминал в данный момент?

Историческая и библиографическая справка. Файловой системе посвящена часть статьи К. Томпсона "UNIX implementation" (BSTJ, July, 1978). Статья Д. Ритчи ""The evolution of the UNIX time-sharing system" (Symposium on Language Desing and Programming Methodology", Sydney, Australia, Sept., 1979) содержит завораживающее описание того, как разрабатывалась и была реализована на исходной PDP-7 файловая система UNIX и как она приобрела нынешнюю форму.

При создании файловой системы UNIX были заимствованы некоторые идеи из системы файлов МАЛТИКС. Содержательное описание последней содержится в книге И. Органика "The MULTICS System: An Examination of its Structure" (MIT Press, 1972).

Статья Б. Морриса и К. Томпсона "Password security: a case history" посвящена интересным сравнениям механизмов паролей во многих системах. Ее можно найти в т. 2B справочного руководства программиста системы UNIX. В том же томе есть статья Д. Ритчи "On the security of UNIX", в которой поясняется, что безопасность системы в большей степени зависит от мер, принимаемых администрацией, чем от деталей таких программ, как `crypt`.

Глава 3

Возможности интерпретатора shell

Интерпретатор shell — это наиболее важная программа для пользователей UNIX, быть может, за исключением вашего любимого текстового редактора. Она исполняет ваши запросы на запуск программ и занимает гораздо больше вашего времени, чем любая другая программа системы. Значительная часть настоящей главы и гл. 5 будут посвящены описанию возможностей интерпретатора. Основная мысль, к которой мы хотим подвести вас, состоит в том, что если вы научитесь работать с интерпретатором, то сможете достичь многого и без особого труда, не прибегая к традиционным языкам программирования типа Си.

Как уже отмечалось, описание интерпретатора разделено на две части. В этой главе от простейших возможностей, показанных в гл. 1, мы перейдем к рассмотрению некоторых необычных, но широко используемых конструкций, таких, как метасимволы, кавычки, новые команды с переданными им аргументами, переменные shell и отдельные структуры управления. Все это понадобится вам для эффективной работы с интерпретатором. Материал гл. 5 более сложный. Изучив его, вы сможете писать настоящие программы на языке shell и даже предоставлять их другим пользователям. Такое деление темы, конечно, во многом произвольно, поэтому мы рекомендуем вам прочитать обе главы.

3.1 Структура командной строки

Прежде чем продолжить рассмотрение, нужно уточнить, что такое команда и как она интерпретируется shell. Этот раздел содержит более формальное описание и некоторую информацию об основных возможностях интерпретатора, описанных в первой главе.

Самая простая команда состоит из одного слова, обычно имени файла, предназначенного для выполнения (позднее вы познакомитесь с другими типами команд):

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

Команда, как правило, завершается символом перевода строки, но может завершаться и точкой 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
$
```

Однако выполнение команды не начнется, пока вы не нажмете клавишу *RETURN*. Обратите внимание на то, что интерпретатор выдает только одно приглашение после нескольких команд, но если не учитывать этого, то ввод

```
$ date; who
```

идентичен вводу двух команд в разных строках. В частности, команда `who` не будет выполняться до завершения `date`. Попробуйте послать результат выполнения этих команд по программному каналу:

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

Возможно, вы получите не то, что ожидали, поскольку только результат команды `who` передается команде `wc`. При связывании `who` и `wc` через программный канал образуется единая команда, называемая конвейером, которая выполняется после `date`. В процессе разбора командной строки `shell` считает приоритет операции '`|`' выше, чем операции '`;`'.

Для группирования команд следует использовать скобки:

```
$ (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 можно передать по программному каналу.

Информацию, поступающую по программному каналу, можно с помощью команды `tee` собрать и поместить в файл (но не в другой программный канал). Команда `tee` является частью интерпретатора `shell`, но тем не менее удобна и при манипулировании программными каналами. Ее можно использовать для сохранения промежуточного результата в файле:

```
$ (date; who) | tee save | wc
3 16 89                                  Результат команды 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 48
$
```

Команда `tee` переписывает свой входной поток в поименованный файл (или файлы), а из него — точно так же без изменений в выходной поток, поэтому `wc` получает те же самые данные, как если бы команда `tee` не присутствовала в конвейере.

В качестве еще одного символа, завершающего команду, применяют амперсанд (`&`). Действие его аналогично действию символа перевода строки и точки с запятой, но он еще и указывает интерпретатору, что не нужно ждать завершения команды. Обычно `&` используется для запуска фоновых, долго выполняющихся команд, в то время как вы продолжаете вводить новые команды в диалоге:

```
$ long-running-command &
5273                                    Номер процесса длительной команды
$                                       Приглашение появляется сразу
```

Имея возможность группировать команды, получаем некоторые интересные способы применения фоновых процессов. Команда `sleep` ожидает указанное число секунд, прежде чем закончить свое выполнение:

```
$ sleep 5
$                                   Проходит 5 секунд до появления приглашения
$ (sleep 5; date) & date
5278
Wed Sep 28 09:18:20 EDT 1983        Результат второй команды date
$ Wed Sep 28 09:18:25 EDT 1983      Появляется приглашение, затем
                                    через 5 секунд дата
```

Фоновый процесс начинается, но сразу "засыпает"; тем временем вторая команда `date` выдает текущее время, а интерпретатор — приглашение для ввода новой команды. Пятью секундами позже прекращается выполнение команды `sleep`, и первая команда `date` выдает новое время. Трудно представить на бумаге истечение времени, поэтому вам следует попытаться самостоятельно реализовать этот пример. (Разница между двумя значениями времени может и не равняться в точности 5 с, в зависимости от загруженности машины и по ряду других причин.) Это удобный способ отложить запуск команды на будущее; рассмотрите также в качестве удобного механизма такой пример:

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

(Если в строке, следующей за командой `echo`, есть символ *ctl-g*, то при появлении ее на экране зазвонит звонок.) В этих примерах нужны скобки, так как приоритет '`&`' выше, чем у '`;`'.

Символ `&` может завершать команды, а поскольку конвейеры являются командами, в скобках для запуска конвейеров как фоновых процессов нет необходимости, поэтому

```
$ pr файл | lpr &
```

позволяет выдать файл на печатающее устройство, не ожидая окончания выполнения команды. Использование скобок дает тот же эффект, но требует введения большего числа символов:

```
$ (pr файл | lpr ) &                    То же, что и в предыдущем примере
```

Большинство команд допускает наличие аргументов в командной строке, таких, как файл в предыдущем примере (аргумент команды `pr`). Аргументами служат слова, разделенные пробелами и символами табуляции, которые обычно именуют файлы, предназначенные для обработки командой. Однако они рассматриваются просто как строки, и программа может интерпретировать их любым подходящим для нее способом. Например, команда `pr` допускает имена файлов, которые нужно напечатать, команда `echo` посылает эхо своих аргументов без всякой интерпретации, а первый аргумент команды `grep` специфицирует строку–шаблон для поиска. И конечно, многие команды имеют необязательные параметры (флаги), задаваемые аргументами, начинающимися со знака "-".

Различные специальные символы, интерпретируемые `shell`, такие, как `<`, `>`, `|`, `;` и `&`, не являются аргументами команд, запускаемых интерпретатором. Они управляют самим процессом запуска. Например,

```
$ echo Hello > junk
```

требует, чтобы интерпретатор запустил команду echo с одним аргументом `Hello` и поместил выходной поток в файл `junk`. Строка `> junk` не является аргументом команды `echo`; она интерпретируется `shell`, и `echo` никогда ее "не увидит". На самом деле, данная строка может и не быть последней в командной строке:

```
$ > junk echo Hello
```

Это идентичный запуск, хотя и менее очевидный.

■ Упражнение: В чем состоит различие между следующими командами?

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

(С течением времени операция переключения `<` потеряла свою связь с программными каналами; "`cat file |`" считается более естественным, чем "`< file`".)

3.2 Метасимволы

Интерпретатор распознает еще ряд символов как специальные. Наиболее часто используется звездочка `*`, указывающая, что нужно искать в каталоге имена файлов, у которых вместо `*` может быть любая последовательность символов. Например,

```
$ echo *
```

есть не что иное, как некое подобие команды `ls`. В гл. 1 мы не отметили, что во избежание проблем с именами '`.`' и '`..`', которые присутствуют в любом каталоге, символы подстановки в именах файлов нельзя применять к именам файлов, начинающимся с точки. Правило таково: символы подстановки в именах файлов действуют на имена файлов, начинающихся с точки, только в том случае, если точка явно задана в шаблоне. Как обычно, "рассудительная" команда `echo` прояснит ситуацию:

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

Символы со специальным значением, подобные *, называются метасимволами. Существует множество метасимволов (в табл. 3.1 приводится их полный список, но некоторые символы мы обсудим только в гл. 5).

При таком количестве метасимволов интерпретатора необходимо иметь возможность экранировать специальный символ от интерпретации. Самый простой и надежный способ экранирования — заключить его в апострофы:

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

Можно также использовать кавычки "...", но интерпретатор на самом деле "заглядывает" внутрь этих кавычек в поиске метасимволов $, `...` и \, так что не применяйте "...", если только вам не требуется определенным образом обработать строку в кавычках.

Еще одну возможность дает ввод обратной дробной черты перед каждым символом, который вы хотите закрыть от интерпретатора, например:

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

Хотя строка *** не похожа на английское слово, в терминологии языка `shell` это слово, ибо им является любая последовательность символов, воспринимаемая интерпретатором как целое, включая даже пробелы, если они взяты в кавычки.

Кавычки одного вида могут экранировать кавычки другого вида:

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

и могут не заключать в себе весь аргумент:

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

В последнем примере команда `echo` получает один аргумент, не содержащий апострофов, так как, сделав свое дело, апострофы исчезают.

Строки в кавычках могут содержать символы строк:

>	prog > file — переключить стандартный выходной поток в файл
>>	prog » file — добавить стандартный выходной поток к файлу
<	prog < file — извлечь стандартней выходной поток из файла
|	p1 | p2 — передать стандартный выходной поток p1 как стандартный выходной поток для p2
<<str	“Документ здесь”: стандартный выходной поток задается в последующих строках до строки, состоящей из одного символа str
*	Задает любую строку, состоящую из нуля или более символов, в имени файла
?	Задает любой символ в имени файла
[ccc]	Задает любой символ из [ccc] в имени файла (допустимы диапазоны, такие, как 0-9 или a-z)
;	Завершает команды: p1; p2 — выполнить p1, затем p2
&	Выполняет аналогичные функции, но не ждет окончания p1
‘...‘	Инициирует выполнение команд(ы) в ...; ‘...‘ заменяется своим стандартным выводом
(...)	Инициирует выполнение команд(ы) в ... в порожденном shell
{...}	Инициирует выполнение команд(ы) в ... в текущем вызове shell (используется редко)
$1, $2, ...	Заменяются аргументами командного файла
$var	Значение переменной var в программе на языке shell
${var}	Значение var; исключает коллизии в случае конкатенации переменной с последующим текстом (см. также табл. 5.3)
\	\c — использовать непосредственно символ c, \ перевод строки отбрасывается
’...’	Означает непосредственное использование
"..."	Означает непосредственное использование, но после того, как $, ‘...‘ и \ будут интерпретированы
#	В начале слова означает, что вся остальная строка рассматривается как комментарий (но не в седьмой версии)
var=value	Присваивает value переменной var
p1 && p2	Предписывает выполнить p1; в случае успеха выполнить p2
p1 || p2	Предписывает выполнить p1; в случае неудачи выполнить p2

Таблица 3.1: Метасимволы shell

```
$ echo ’hello
> world’
hello
world
$
```

Символ > является вторичным приглашением интерпретатора, которое выдается, если ожидается продолжение ввода для завершения команды. В этом примере апостроф в первой строке должен быть уравновешен другим апострофом. Вторичное приглашение интерпретатора хранится в переменной PS2; его можно изменить по своему вкусу.

Во всех приведенных выше примерах экранирование специальных символов предохраняет их от интерпретации. Команда

```
$ echo x*y
```

выдает все имена файлов, начинающиеся с `x` и кончающиеся `y`. Как обычно, команда `echo` ничего "не знает" ни о файлах, ни о метасимволах; интерпретация `*`, если она требуется, осуществляется `shell`.

Что произойдет, если ни один файл не будет соответствовать шаблону? Интерпретатор просто пропустит строку, как если бы она была взята в кавычки, а не выразит вам свое неудовольствие (как было принято в ранних версиях). Конечно, не следует рассчитывать на это свойство, но его можно использовать, чтобы узнать о существовании файлов, соответствующих шаблону:

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

Появление обратной дробной черты в конце строки требует продолжения строки, что является способом задать интерпретатору очень длинную строку:

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

Обратите внимание на то, что символ перевода строки отбрасывается, если перед ним стоит обратная дробная черта, но он остается, если взят в кавычки. Метасимвол `#` в программе на языке `shell` практически всюду используется в качестве комментария; если слово начинается с `#`, остаток строки игнорируется:

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

Символ `#` не присутствует в оригинальной седьмой версии, но имеет очень широкое распространение, и в дальнейшем мы будем им пользоваться.

■ Упражнение: Объясните результат выполнения команды

```
$ ls .
```

Некоторые дополнительные сведения о команде echo

Команда echo выдает заключительный символ перевода строки, даже если на это нет явного запроса. Разумной и, возможно, более корректной была бы такая реализация команды echo, при которой вывод соответствовал бы только запросу. Добиться этого легко, если потребовать от интерпретатора выдачи приглашения:

```
$ правильное эхо введенная команда:
Введенная команда: $                    Нет завершающего перевода строки
```

Однако при таком решении в самой распространенной ситуации, когда перевод строки нужен, он не подразумевается по умолчанию и требует дополнительного ввода:

```
$ правильное эхо 'Привет!
>'
Привет!
$
```

Поскольку команда должна по умолчанию выполнять наиболее часто встречающееся действие, настоящее эхо автоматически добавляет перевод строки. Но как быть, если это нежелательно? В седьмой версии системы команда echo имеет единственный флаг -n, который подавляет последний символ перевода строки:

```
$ echo -n Enter a command:
Enter a command: $              Приглашение на той же строке
$ echo -
-                               Только --- является специальным случаем
$
```

Существует одна маленькая хитрость в случае получения эха от -n, за которым должен следовать символ перевода строки:

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

Такое решение некрасиво, но эффективно, к тому же это довольно редкий случай.

Другой подход принят в System V, где команда echo интерпретирует последовательность символов с обратной дробной чертой аналогично тому, как это делается в языке Си, а именно: \b обозначает "шаг назад", \c подавляет перевод строки (правда, здесь не очень точно воспроизведена конструкция Си):

```
$ echo 'Введенная команда: \c'                      Версия System V
Введенная команда: $
```

Хотя при подобном решении не возникает коллизий при получении эха от знака "-", у него есть свои недостатки. Команда echo часто используется в качестве диагностического средства, а символ обратной дробной черты интерпретируется таким множеством программ, что участие в этом команды echo только вносит дополнительную путаницу.

Итак, обе реализации команды `echo` имеют и положительные, и отрицательные стороны. Мы будем использовать вариант седьмой версии (`-n`), поэтому, если ваша команда `echo` выполняется по-другому, несколько приводимых ниже примеров потребуют незначительных изменений.

Возникает еще один, философский, вопрос: что должна делать команда, если ей не передали аргументов, в частности, следует ли ей выдавать пустую строку или вообще ничего не предпринимать? Как вы уже знаете, все настоящие реализации команды выдают пустую строку, но в ранних версиях все было иначе. По этому поводу велись большие дебаты, а Д. МакИлрой привнес в них даже элемент мистицизма.

UNIX и Эхо

Жила–была в стране Нью–Джерси UNIX, прекрасная девушка, к которой приезжали издалека, чтобы полюбоваться ею. Ослепленные чистотой UNIX, все искали ее руки и сердца: одни — за изящество, другие — за изысканную вежливость, третьи — за проворность при выполнении самых изнурительных заданий. Была она от рождения столь великодушна и услужлива, что все женихи остались довольны ею, а ее многочисленное потомство распространилось во все концы земли.

Сама природа покровительствовала UNIX и вторила ей более охотно, чем кому-либо из смертных. Простые люди поражались ее эхом, таким оно было точным и кристально чистым. Они не могли поверить, что ей отвечают те же леса и скалы, которые так искажают их собственные голоса. Когда один нетерпеливый пастушок попросил UNIX: “Пусть эхо ответит ничего”, и она послушно открыла рот, эхо промолчало. “Зачем ты открываешь рот?” — спросил пастушок. — “Отныне никогда не открывай его, если эхо должно ответить ничего!” — и UNIX подчинилась.

“Но я хочу совершенного исполнения, даже если эхо отвечает ничего,” — потребовал другой, обидчивый, юноша, — “а никакого совершенного эха не получится при закрытом рте”. Не желая обидеть никого из них, UNIX согласилась говорить разные “ничего” для нетерпеливого и обидчивого юношей. Она называла “ничего” для обидчивого как ’`\n`’. Однако теперь, когда она говорила ’`\n`’, на самом деле она не произносила ничего, поэтому ей приходилось открывать рот дважды: один раз, чтобы сказать ’`\n`’, и второй раз, чтобы не сказать ничего. Это не понравилось обидчивому юноше, который тотчас сказал: “Для меня ’`\n`’ звучит, как настоящее “ничего”, но когда ты открываешь рот второй раз, то все портишь. Возьми второе “ничего” назад”. Услужливая UNIX согласилась отказаться от некоторых эхо и обозначила это как ’`\c`’. С тех пор обидчивый юноша мог услышать совершенное эхо “ничего”, если он задавал ’`\n`’ и ’`\c`’ вместе, но говорят, что он так и не услышал его, поскольку умер от излишеств в обозначениях.

■ Упражнение: Предскажите, что сделает команда `grep` в каждом случае, а затем проверьте себя;

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

Файл, состоящий из таких команд, послужит хорошим материалом для теста, если вы хотите поэкспериментировать.

■ Упражнение: Как указать `grep`, что нужно найти шаблон, начинающийся с '-'? Почему взятие аргумента в кавычки не помогает? Подсказка: исследуйте флаг `-e`.

■ Упражнение: Рассмотрите команду

```
$ echo */*
```

Может ли она вывести все имена всех каталогов? В каком порядке появятся эти имена?

■ Упражнение: (*Хитрый вопрос.*) Как ввести `/` в локальное имя файла (т. е. символ `/`, который не является разделителем компонентов в абсолютном имени)?

■ Упражнение: Что произойдет в случае ввода команд

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

Подумайте, прежде чем броситься их выполнять.

■ Упражнение: Если вы введете `$ rm *` почему команда `rm` не сможет предупредить вас, что вы собираетесь удалить все ваши файлы?

3.3 Создание новых команд

Теперь, как мы обещали вам в гл. 1, рассмотрим создание новых команд из старых. Имея последовательность команд, которую придется многократно повторять, преобразуем ее для удобства в "новую" команду со своим именем и будем использовать ее как обычную команду. Чтобы быть точными, предположим, что нам предстоит часто подсчитывать число пользователей с помощью конвейера

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

(см. гл. 1), и для этой цели нужна новая программа `nu`.

Первым шагом должно быть создание обычного файла, содержащего '`who | wc -l`'. Можно воспользоваться вашим любимым редактором или проявить изобретательность:

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

(Что появится в файле `nu`, если не употреблять кавычки?)

Как отмечалось в гл. 1, интерпретатор является точно такой же программой, как редактор, `who` или `wc`; он называется `sh`. А коль скоро это программа, ее можно вызвать и переключить ее входной поток. Так что запускаем интерпретатор с входным потоком, поступающим 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
$
```

Результат получился таким же, каким бы он был при задании команды `who | wc -l` с терминала. Опять-таки, как и большинство программ, интерпретатор берет входной поток из файла, если он указан в качестве аргумента; вы с тем же успехом могли задать:

```
$ sh nu
```

Однако досадно вводить "`sh`" каждый раз; во всяком случае эта запись длиннее и создает различия между программами, написанными, например, на Си, и программами, написанными с помощью `shell` (Тем не менее такое различие существует в большинстве операционных систем). Поэтому если файл предназначен для выполнения и если он содержит текст, то интерпретатор считает, что он состоит из команд. Такие файлы называются командными. Все, что вам нужно сделать, это объявить файл nu выполняемым, задав

```
$ chmod + x nu
```

а затем вы можете вызывать его посредством

```
$ nu
```

С этого момента те, кто используют файл `nu`, не смогут определить способ его создания.

Способ, с помощью которого интерпретатор на самом деле выполняет `nu`, сводится к созданию нового процесса интерпретатора, как если бы вы задали

```
$ sh nu
```

Этот процесс–потомок называется порожденным интерпретатором, т. е. процессом интерпретатора, возбужденным вашим текущим интерпретатором. Но команда `sh nu` — это не то же самое, что `sh < nu`, поскольку в первом случае стандартный входной поток все еще связан с терминалом. Пока команда `nu` выполняется только в том случае, если она находится в вашем текущем каталоге (при условии, конечно, что текущий каталог включен в `PATH`, а именно это мы и предполагаем с настоящего момента). Чтобы сделать команду `nu` частью вашего репертуара независимо от того каталога, с которым вы работаете, занесите ее в свой собственный каталог `bin` и добавьте `/usr/you/bin` к списку каталогов поиска:

```
$ pwd
/usr/you
$ mkdir bin                          Создать bin, если его еще не было
$ echo $PATH                         Проверить Path, чтобы убедиться
:/usr/you/bin:/bin:/usr/bin          Должно быть нечто похожее
$ mv nu bin                          Установить команду nu в bin
$ ls nu
nu not found                         Она действительно исчезла
                                     из текущего каталога
$ nu
4                                    Но интерпретатор ее находит
$
```

Конечно, ваша переменная `PATH` должна быть правильно определена в файле `.profile`, чтобы вам не приходилось переопределять ее при каждом входе в систему.

Существуют и другие простые команды, которые вы можете адаптировать к среде по своему вкусу и создавать таким же способом. Нам показалось удобным иметь следующие команды:

- `cs` для посылки подходящей последовательности специфических символов с целью очистки экрана вашего терминала (24 символа перевода строки — практически универсальное решение);
- `what` для запуска `who` и `ps -a`, чтобы сообщить, кто работает в системе и что он делает;
- `where` для вывода идентифицированного названия используемой системы UNIX. Это удобно, если вы постоянно работаете с несколькими версиями. (Установка `PS1` служит для подобной цели.)

■ Упражнение: Просмотрите каталоги `/bin` и `/usr/bin`, чтобы выяснить, как много команд являются в действительности командными файлами. Можно ли это сделать с помощью одной команды? Подсказка: посмотрите `file(1)`. Насколько точно предположение, основанное на длине файла?

3.4 Аргументы и параметры команд

Хотя команда `nu`, как она задумывалась, удовлетворяет своему назначению, многие программы на языке `shell` могут обрабатывать аргументы, так что при их запуске можно задавать имена файлов и флаги.

Допустим, вы хотите создать программу с именем `cx` для установки права доступа к файлу на выполнение, так что

```
$ cx nu
```

есть сокращенная запись для

```
$ chmod +x nu
```

Вы уже знаете почти все, чтобы это сделать. Вам нужен файл `cx`, содержимое которого суть

```
chmod +x filename
```

Единственное, что требуется выяснить — как сообщить команде `cx` имя файла, так как при каждом запуске `cx` оно будет иным.

Если интерпретатор выполняет командный файл, то каждое вхождение `$1` заменяется первым аргументом, каждое вхождение `$2` — вторым и т. д. до `$9`. Поэтому если файл `cx` содержит строку

```
chmod +x $1
```

то при выполнении команды

```
$ cx nu
```

порожденный интерпретатор заменит “`$1`” на первый аргумент “`nu`”. Рассмотрим всю последовательность операций:

```
$ echo 'chmod +x $1' >cx                    Вначале создадим cx
$ sh cx cx                                  Сделать сам файл cx выполняемым
$ echo echo Hi, there! >hello               Приготовим тест
$ hello                                     Попробуем
hello: cannot execute
$ cx hello                                  Сделаем файл выполняемым
$ hello                                     Попробуем снова
Hi, there!                                  Работает
$ mv cx /usr/you/bin                        Установим команду cx
$ rm hello                                  Уберем ненужное
$
```

Заметьте, что мы задали

```
$ sh cx cx
```

в точности так, как сделал бы автоматически интерпретатор, если бы cx была выполняемой и можно было бы задать

```
$ cx cx
```

А как быть, если нужно работать с несколькими аргументами, например, заставить программу cx воздействовать сразу на несколько файлов? Прямолинейное решение состоит в том, чтобы включить девять аргументов в командный файл:

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

(Это годится только для девяти аргументов, так как конструкция $10 распознается как "первый аргумент, за которым следует 0"!) Если пользователь такого командного файла задаст меньше девяти аргументов, то недостающие окажутся пустыми строками. Это приведет к тому, что только настоящие аргументы будут переданы chmod порожденным интерпретатором. Такое решение, конечно, приемлемо, но не вполне корректно и не подходит для случая с числом аргументов более девяти.

С учетом упомянутой выше трудности интерпретатор предоставляет сокращенную запись $*, означающую "все аргументы". В этом случае правильно определить cx:

```
chmod +x $*
```

что является эффективным при любом числе аргументов.

Используя $* в своем репертуаре, вы можете создать некоторые полезные командные файлы, такие, как lc или m:

```
$ cd /usr/you/bin
$ cat lc
# lc: подсчет числа строк в файлах
wc -l $*
$ cat m
# m: точный способ послать почту
mail $*
$
```

Обе команды можно осмысленно использовать и без аргументов. Если нет аргументов, $* будет пустым, и wc и mail вообще не получат никаких аргументов. С аргументами или без них 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 и другие команды, описываемые в настоящей главе, являются командами пользователя, т. е. вы создаете их для себя и помещаете в свой каталог /bin, поэтому вряд ли они должны стать общедоступными. В гл. 5 мы исследуем вопрос создания общедоступных программ на языке shell.

Аргументами командного файла не обязательно должны быть имена файлов. Рассмотрим в качестве примера поиск в каталоге, где хранится личный телефонный справочник. Если у вас есть файл с именем /usr/you/lib/phone-book, содержащий строки следующего вида:

```
dial-a-joke       212-976-3838
dial-a-prayer     212-246-4200
dial santa        212-976-3636
dow jones report  212-976-4141
```

то для поиска в нем можно воспользоваться командой grep. (Ваш собственный каталог lib — хорошее хранилище таких частных баз данных.) Поскольку команда grep не определяет формат информации, можно искать имена, адреса, индексы или еще какие–нибудь нужные вам сведения. Составим справочную программу для каталога, которой дадим имя 411 по номеру одной из телефонных справочных служб:

```
$ 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 вскрывает потенциальную проблему: хотя dow jones представляет для команды 411 единый аргумент, он содержит пробел и уже не заключен в апострофы, поэтому порожденный интерпретатор, выполняющий команду 411, преобразует его в два аргумента для grep, как если бы вы задали

```
$ grep dow jones /usr/you/lib/phone-book
```

что, очевидно, неверно.

Один из возможных путей обойти эту проблему основан на том, как интерпретатор трактует кавычки. Хотя все, что заключено в '...', не затрагивается, интерпретатор "заглядывает" внутрь "..." в поиске комбинаций с $, \, `...`. Поэтому если изменить команду 411 следующим образом:

```
$ grep "$*" /usr/you/lib/phone-book
```

то $* заменяется на аргументы, но команде grep передается как один аргумент, даже при наличии пробелов:

```
$ 411 dow jones
dow jones report  212-976-4141
$
```

Кстати, можно сделать с помощью флага `-y` команду `grep` (а значит, и `411`) независимой от использования строчных или прописных букв:

```
$ grep -y pattern ...
```

При наличии флага `-y` строчные буквы из шаблона могут сопоставляться с прописными буквами из входного потока. (Такой флаг есть в седьмой версии, но отсутствует в других системах.)

Более подробно аргументы команд мы рассмотрим в гл. 5, но одно важное замечание необходимо сделать здесь. Аргумент `$0` — это имя выполняемой программы; в случае `cx` `$0` есть `"cx"`. Новое применение `$0` находит в реализации программ `2`, `3`, `4`, ..., которые печатают свой выходной поток в несколько столбцов:

```
$ who | 2
drh tty0 Sep 28 21:23     cvw tty5 Sep 28 21:09
dmr tty6 Sep 28 21:10     scj tty7 Sep 28 22:11
you tty9 Sep 28 23:00     jlb ttyb Sep 28 19:58
$
```

Реализация команд `2`, `3`, ... идентична. По существу, они являются связями с одним файлом:

```
$ ln 2 3; ln 2 4; ln 2 5; ln 2 6
$ ls -l [1-9]
167222 -rwxrwxrwx 5 you 51 Sep 28 23:21 2
167222 -rwxrwxrwx 5 you 51 Sep 28 23:21 3
167222 -rwxrwxrwx 5 you 51 Sep 28 23:21 4
167222 -rwxrwxrwx 5 you 51 Sep 28 23:21 5
167222 -rwxrwxrwx 5 you 51 Sep 28 23:21 6
$ ls /usr/you/bin | 5
2     3     4     411     5
6     ex    lc    m       nu
what  where
$ cat 5
# 2, 3, ...: печать в n столбцов
pr -$0 -t -l1 $*
$
```

Флаг `-t` убирает заголовки в начале страницы, а флаг `-ln` устанавливает размер страницы равным `n` строк. Имя программы становится числом столбцов, т. е. аргументов для команды `pr`, так что выходной поток печатается строками по несколько столбцов, число которых определено аргументом `$0`.

3.5 Результат выполнения программы в качестве аргумента

Теперь перейдем от аргументов команд для командного файла к порождению аргументов. Конечно, расширение имен файлов с помощью метасимволов, подобных *, является наиболее типичным способом порождения аргументов (иным, чем их явное задание), но столь же хорошим способом представляется и выполнение программы. Результат выполнения любой программы можно использовать в командной строке, заключив ее вызов в символы слабого ударения `...`:

```
$ 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 показывает, что `...` интерпретируется и внутри 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.
$
```

В качестве другого примера предположим, что вам необходимо послать почту группе людей, которые зарегистрированы под именем, хранящимся в файле `mailinglist`. Можно, конечно, отредактировать файл `mailinglist` так, чтобы он стал пригодным для применения команды `mail` и передать его интерпретатору, но значительно проще использовать команду

```
$ mail `cat mailinglist` <letter
```

Запуск команды `cat` порождает список имен пользователей, и эти имена становятся аргументами команды `mail`. (При обработке результата выполнения команды, помещенной между знаками слабого ударения и используемой в качестве аргумента, интерпретатор считает символы перевода строки разделителями слов, а не символами завершения командной строки; подробнее данный вопрос обсуждается в гл. 5.) Работать со знаками слабого ударения нетрудно, и поэтому, действительно, нет нужды вводить отдельный флаг команды mail, задающий список адресатов.

Несколько иной подход требуется для преобразования файла `mailinglist` из простого списка имен в программу, выдающую список имен:

```
$ cat mailinglist
echo don whr ejs mb                                    Новая версия
$ cx mailinglist
$ mailinglist
don whr ejs mb
$
```

Теперь посылка писем адресатам из списка реализуется командой:

```
$ mail `mailinglist` <letter
```

Добавив еще одну программу, получим возможность даже изменять список пользователей в диалоге. Такая программа называется `pick`:

```
$ pick аргументы...
```

и выдает свои аргументы по одному, ожидая каждый раз ответа. Результатом действия команды `pick` являются те аргументы, на которые был дан ответ y (yes — да); при всяком другом ответе аргумент отбрасывается. Например,

```
$ pr `pick *.c` | lpr
```

Здесь вначале выдаются имена файлов, оканчивающиеся на `.c`. Выбранные имена печатаются с помощью команд `pr` и `lpr`. (Команда `pick` не входит в состав команд седьмой версии, но она столь проста и полезна, что мы включили ее варианты в гл. 5 и 6).

Допустим, вы используете второй вариант команды `mailinglist`. Тогда посылка писем адресатам don и mb выглядит так:

```
$
\begin{verbatim}mail `pick \`mailinglist\`` <letter
don? y
whr?
ejs?
mb? y
$
```

Обратите внимание на вложенные знаки слабого ударения; обратная дробная черта запрещает обработку вложенной конструкции `...` при разборе внешних знаков слабого ударения.

■ Упражнение: Что произойдет, если опустить символы обратной дробной черты в команде

```
$ echo ` echo \`date\``
```

■ Упражнение: Попробуйте ввести

```
$`date`
```

и объясните результат.

■ Упражнение: Команда

```
$ grep -l pattern filenames
```

перечисляет имена файлов, которые соответствуют шаблону, но больше ничего не выдает. Попытайтесь выполнить разные вариации такого задания:

```
$ command `grep -l pattern filenames`
```

3.6 Переменные языка shell

Подобно большинству языков программирования, shell имеет переменные, которые на программистском жаргоне называются параметрами. Такие строки, как $1, являются позиционными параметрами–переменными, хранящими аргументы командного файла. Цифра показывает положение параметра в командной строке. Ранее мы имели дело с другими переменными языка shell: PATH — список каталогов, в которых происходит поиск команд, HOME — ваш начальный каталог и т. д. В отличие от переменных в обычном языке переменные, задающие позиционные параметры, не могут быть изменены; хотя PATH представляет собой переменную со значением $PATH, нет переменной 1 со значением $1, т. е. $1 — это не что иное, как компактное обозначение первого аргумента.

Если забыть о позиционных параметрах, переменные языка shell можно создавать, выбирать и изменять. Например,

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

означает присваивание, изменяющее список каталогов в процессе поиска. До и после знака равенства не должно быть пробелов. Присваиваемое значение должно выражаться одним словом, и его следует взять в кавычки, если оно содержит метасимволы, которые не нужно обрабатывать. Значение переменной выбирается, если предварить имя знаком доллара:

```
$ PATH=$PATH:/usr/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$ PATH=:/usr/you/bin:/bin:/usr/bin                Восстановим значение
$
```

Не все переменные имеют специальное значение для интерпретатора. Можно создавать новые переменные, присваивая им значения. По традиции переменные, имеющие специальное значение, обозначаются прописными буквами, а обычные переменные — строчными. Типичным примером использования переменных является хранение в них длинных строк, таких, как имена файлов:

```
$ pwd
/usr/you/bin
$ dir=‘pwd‘                              Запомним, где находимся
$ cd /usr/mary/bin                       Перейдем в другое место
$ ln $dir/cx .                           Используем переменную в имени файла
$ ...                                    Поработаем некоторое время
$ cd $dir                                Вернемся
$ pwd
/usr/you/bin
$
```

Встроенная в интерпретатор команда set показывает значения всех определенных вами переменных. Для просмотра одной или двух переменных более подходит команда 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 связано с той копией интерпретатора, который создал ее, и автоматически не передается процессам — потомкам интерпретатора.

```
$ x=Hello                       Создание x
$ sh                            Новый \cmd{shell}
$ echo $x
                                Происходит только перевод строки,
                                x не определено в порожденном интерпретаторе
$ ctl-d                         Возврат в исходный интерпретатор
$ echo $x
Hello                           x по-прежнему определено
$
```

Это означает, что в командном файле нельзя изменить значение переменной, поскольку выполнением командного файла управляет порожденный интерпретатор:

```
$ echo 'x="Good bye"    Создание \cmd{shell}-файла из двух строк...
> echo $x' >setx        ...для определения и печати x
$ cat setx
x="Good Bye"
echo $x
$ echo $x
Hello                   x есть Hello в исходном интерпретаторе
$ sh setx
Good Bye                x есть Good Bye в порожденном интерпретаторе...
$ echo $x
Hello                   ...но по-прежнему есть Hello в текущем интерпретаторе
$
```

Однако бывают ситуации, когда было бы полезно изменять переменные интерпретатора в командном файле. Очевидным примером является файл, добавляющий новый каталог к вашей переменной `PATH`. Поэтому интерпретатор предоставляет команду '.' (точка), которая выполняет команды из файла в текущем, а не порожденном интерпретаторе. Первоначально это было задумано для удобства пользователей, чтобы они могли повторно выполнять свой файл `.profile`, не входя заново в систему, но в принципе это открывает и другие возможности:

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games                          Добавим /usr/games к 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 с помощью переменной PATH, так что его можно поместить в ваш каталог bin.

Когда используется команда '.', только условно можно считать, что выполняется командный файл. Файл не "выполняется" в обычном смысле этого слова. Команды из него запускаются, как если бы вы ввели их в диалоговом режиме: стандартный входной поток интерпретатора временно переключается на файл. Поскольку файл читается, не нужно иметь право на его выполнение. Другое отличие состоит в том, что файл не получает аргументов командной строки; $1, $2 и т. д. являются пустыми строками. Было бы неплохо, если бы аргументы передавались, но этого не происходит.

Есть еще один способ установить значение переменной в порожденном интерпретаторе — присвоить его явно в командной строке перед самой командой:

```
$ echo 'echo $x' >echox
$ cx echox
$ echo $x
Hello                          Как и прежде
                               x не определено в порожденном интерпретаторе
$ x=Hi echox
Hi                             Значение x передается порожденному интерпретатору
$
```

(Первоначально присваивания всюду в командной строке передавались команде, но это противоречило dd(1).)

Операцию '.' следует использовать, чтобы навсегда изменить значение переменной, тогда как присваивания в командной строке предназначены для временных изменений. В качестве примера рассмотрим еще раз поиск команд в каталоге /usr/games, не указанном в вашей переменной PATH:

```
$ ls /usr/games | grep fort
fortune                                    Игровая команда fortune
$ fortune
fortune: not found
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                /usr/games не входит в PATH
$ PATH=/usr/games fortune
Позвони в звонок; закрой книгу; задуй свечу.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                PATH не изменилось.
$ cat /usr/you/bin/games                   команда games все еще здесь
$ . games
```

```
$ fortune
Непродуманная оптимизация --- источник всех бед --- Кнут
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games       Сейчас PATH изменилось
$
```

Можно использовать оба средства в одном командном файле. Вы можете несколько видоизменить команду games для запуска одной игровой программы без изменения переменной PATH или постоянно переопределять PATH, чтобы она включала /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                 /usr/games не входит
$ games fortune
Готов отдать свою правую руку, чтобы одинаково владеть обеими
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                 Все еще не входит
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games      Теперь входит
$ fortune
Тот, кто медлит, иногда спасается
$
```

При первом обращении к games командный файл выполняется в порожденном интерпретаторе, в котором переменная PATH временно изменена так, чтобы включать каталог /usr/games. В то же время во втором примере файл обрабатывается текущим интерпретатором при значении $*, являющемся пустой строкой, поэтому в строке нет команд и переменная PATH изменяется. Применение команды games в обоих случаях достаточно нетривиально, но в результате получаем удобное и естественное для использования средство.

Для того чтобы значение переменной было доступно в порожденном интерпретаторе, следует использовать команду export языка shell. (Вы можете поразмышлять о том, почему нет возможности экспортировать значение переменной от порожденного интерпретатора к порождающему его.) Приведем один из рассмотренных выше примеров, но теперь с экспортом переменной:

```
$ x=Hello
$ export x
$ sh                              Новый интерпретатор
$ echo $x
Hello                             x доступно в порожденном интерпретаторе
$ x='Good Bye'                    Изменим значение x
$ echo $x
Good Bye
$ ctl-d                           Выйдем из порожденного интерпретатора
$                                 Снова в исходном интерпретаторе
```

```
$ echo $x
Hello                                x по-прежнему Hello
$
```

Семантика команды `export` нетривиальна, но по крайней мере для повседневных нужд достаточно придерживаться основного правила: никогда не экспортируйте временные переменные, служащие для краткосрочных целей, и всегда экспортируйте переменные, необходимые вам во всех порожденных интерпретаторах (включая, например, интерпретаторы, запускаемые командой ! редактора `ed`). Поэтому переменные, имеющие специальное значение для интерпретатора, такие, как `PATH` и `HOME`, следует экспортировать.

■ Упражнение: Почему в значение переменной `PATH` всегда включается текущий каталог? Куда его нужно поместить?

3.7 Еще раз о переключении ввода–вывода

Понятие стандартного потока диагностики было введено для того, чтобы сообщения об ошибках всегда появлялись на терминале:

```
$ diff file1 file2 >diff.out
diff: file2: No such file or directory
$
```

Без сомнения, сообщения об ошибке должны появляться подобным образом — было бы крайне неприятно, если бы они исчезли в файле `diff.out`, оставляя вас в уверенности, что ошибочная команда `diff` выполнена правильно.

В начале выполнения каждой программы определены по умолчанию три файла, обозначаемые небольшими целыми числами и называемые дескрипторами файла (мы рассмотрим их в гл. 7). Со стандартными входным (0) и выходным (1) потоками вы уже знакомы: они часто переключаются на файл или программный канал. Последний поток с номером 2 представляет собой стандартный поток диагностики и обычно предназначается для вывода на терминал.

Иногда программы осуществляют вывод в стандартный поток диагностики, даже если они работают правильно. Типичным примером является программа `time`, которая выполняет команду и выдает в стандартный поток диагностики сообщение о том, сколько времени заняло выполнение:

```
$ 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** (между 2 и > не должно быть пробелов) переключает стандартный поток диагностики на файл; синтаксически она непривлекательна, но служит своей цели. (Для такого короткого теста, как приведенный выше, время, выдаваемое командой **time**, не совсем правильное, но для последовательности больших тестов она выводит полезную информацию, которой можно доверять в разумных границах. Вы вполне можете сохранить ее для дальнейшего анализа; обратитесь, например, к табл. 8.1.)

Допустимо также слияние 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
$
```

Обозначение 2>&1 является указанием интерпретатору, что стандартный поток диагностики нужно поместить в тот же поток, что и стандартный выходной. Амперсанд не содержит какого-либо мнемонического смысла; это просто идиома, которую следует запомнить. Для добавления стандартного выходного потока к стандартному потоку диагностики можно использовать 1>&2:

```
echo ... 1>&2
```

В командных файлах это позволяет предотвратить исчезновение сообщений в файле или программном канале.

Интерпретатор предоставляет возможность размещать стандартный входной поток вместе с командой, а не в отдельном файле, так что командный файл может хранить всю информацию в себе самом. Наша справочная программа 411, работающая с каталогом телефонов, могла быть задана так:

```
$ 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
$
```

Программирующие на языке **shell** называют такую конструкцию “документ здесь”, т. е. входной поток находится здесь, а не в каком-нибудь файле. Началом конструкции

служит <<; последующее слово (в нашем примере End) является ограничителем входного потока, включающего все строки до той, которая содержит только данное слово. Интерпретатор выполняет замену конструкций $,‘...‘ и \ в “документе здесь”, если только часть слова не экранирована кавычками или обратной дробной чертой, — в этом случае весь документ берется без изменений. В конце главы мы рассмотрим еще более интересный пример с конструкцией “документ здесь”.

В табл. 3.2 перечислены различные виды переключения ввода–вывода, допускаемые интерпретатором.

■ Упражнение: Сравните версии программы 411: использующую “документ здесь” и первоначальную. Какую легче сопровождать? Какая более подходит в качестве основы общего служебного средства?

> файл	Переключение стандартного выходного потока в файл
>> файл	Добавление стандартного выходного потока в файл
< файл	Получение стандартного выходного потока из файла
p1 | p2	Передача стандартного выходного потока программы p1 в качестве входного потока для программы p2
^	Устарелый синоним |
n> файл	Переключение выходного потока из файла с дескриптором n в файл
n>> файл	Добавление выходного потока из файла с дескриптором n в файл
n>&m	Слияние выходных потоков файлов с дескрипторами n и m
<<s	“Документ здесь”: берется стандартный входной поток до строки, начинающейся с s; выполняется подстановка для $, ‘...‘ и \
<<\s	“Документ здесь” без подстановки
<<’s’	“Документ здесь” без подстановки

Таблица 3.2: Переключение ввода–вывода интерпретатора

3.8 Циклы в shell–программах

Язык shell — действительно язык программирования: в нем есть переменные, циклы, ветвления и т. п. Здесь мы обсудим основные циклы, а структуры управления рассмотрим более подробно в гл. 5.

Типичным считается цикл по последовательности имен файлов, и оператор for языка shell является единственной структурой управления, которую обычно задают с терминала, а не помещают в файл для последующего выполнения. Синтаксис оператора for таков:

```
for перем in список_слов
do
    команды
done
```

Например, для получения эха имен файлов по одному на строке достаточно задать:

```
$ for i in *
> do
> echo $i
> done
```

Вместо `i` можно применять любую переменную языка `shell`, но это обозначение традиционно. Заметьте, что значение переменной получается с помощью `$i`, однако в заголовке цикла переменную указывают как `i`. Мы задействовали `*` для выбора всех файлов текущего каталога, но можно использовать и любой другой список аргументов. Обычно нужно сделать что-нибудь более интересное, чем печать имен файлов. Нам часто приходилось сравнивать набор файлов с их предыдущими версиями, например старую версию гл. 2 (хранимую 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 $1
> diff -b old/$i $i
> echo Добавим пустую строку для красоты
> done | pr -h "diff ‘pwd‘/old ‘pwd‘ | lpr &
3712                                                Номер процесса
$
```

Выходной поток направлен по конвейеру через команды `pr` и `lpr` просто для того, чтобы показать, что это возможно: стандартный выходной поток программ, находящихся внутри цикла `for`, попадает в стандартный выходной поток самой команды `for`. С помощью флага `-h` в команде `pr` мы поместили в выходной поток заголовок с “архитектурными излишествами”, используя два вложенных обращения к `pwd`. Вся последовательность команд запущена асинхронно (`&`), так что не нужно ждать ее окончания; `&` применяется ко всякому циклу и конвейеру.

Мы предпочитаем указанный формат для цикла `for`, но вы можете сократить его. Единственное ограничение заключается в том, что `do` и `done` распознаются как ключевые слова, только если они появляются сразу после перевода строки или точки с запятой. В зависимости от размера цикла `for` иногда лучше помещать все на одной строке:

```
for i in список; do команды; done
```

Следует использовать цикл `for` для обработки составных команд или в тех случаях, когда не подходит встроенная обработка отдельных команд. Но не применяйте его там, где в отдельной команде есть цикл по именам файлов:

```
# Плохая идея:
for i in $*
do
    chmod +x $i
done
```

Предпочтительнее сделать так:

```
chmod + x $*
```

поскольку в цикле `for` отдельная команда `chmod` выполняется для каждого файла, что требует больших вычислительных ресурсов. (Убедитесь в том, что вы понимаете разницу между командами

```
for i in *
```

в которой цикл выполняется по всем именам файлов текущего каталога, и

```
for i in $*
```

в которой цикл выполняется по всем аргументам командного файла.)

Список аргументов для цикла `for` часто получают путем выбора имен файлов по шаблону, но можно получать и любым другим способом, в частности:

```
for i in ‘cat ...‘
```

или просто вводом аргументов. Например, ранее в этой главе мы создали ряд программ для печати в несколько столбцов под именами 2, 3 и т. д. Они являются связями с одним файлом, которые можно установить следующим образом (при условии, что программа 2 написана):

```
$ for i in 3 4 5 6; do ln 2 $i; done
$
```

Цикл `for` имеет и более интересное назначение. Выберем с помощью команды `pick` те файлы, которые будут сравниваться с файлами из каталога старых версий:

```
$ 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, данный цикл следует поместить в командный файл, чтобы уменьшить ввод в следующий раз (ведь если вы что-то сделали дважды, вероятно, вы сделаете это и в третий раз).

■ Упражнение: Если цикл с командой `diff` хранится в командном файле, поместите ли вы туда команду `pick`? Объясните, почему.

■ Упражнение: Что произойдет, если последняя строка приведенного цикла будет иметь вид:

```
> done | pr | lpr &
```

т. е. кончаться амперсандом? Попробуйте сделать прогноз, а затем проверьте его.

3.9 Программа bundle: соберем все воедино

Чтобы лучше понять, как создаются командные файлы, обратимся к такому примеру. Предположим, вы получили почту от приятеля с другой машины: "`где-то!боб`" (Существует несколько вариантов обозначений для адресата на другой машине. Наиболее общим является следующее: `машина!пользователь`[1]. См. справочное руководство по `mail(1)`), и он хотел бы скопировать командные файлы из вашего каталога `bin`. Самый простой способ их пересылки заключается в ответной почте, так что вы могли бы начать вводить:

```
$ cd /usr/you/bin
$ for i in ‘pick *‘
> do
> echo ============== Это файл $i ==============
> cat $i
> done | mail где-то!боб
$
```

Однако посмотрим на это с точки зрения адресата "`где-то!боб`": он должен получить почту, в которой все файлы четко разделены, но ему придется воспользоваться редактором для разбивки сообщений на отдельные файлы. Для того чтобы адресату ничего не надо было делать, почтовое сообщение, построенное подходящим образом, должно автоматически распаковать себя, а значит, оно должно быть командным файлом, содержащим и сами файлы, и операции по их распаковке. Вторая идея заключается в том, что конструкция языка `shell` "документ здесь" является удобным способом задания информации для команды при ее запуске. Тогда остальная часть задачи сводится к тому, чтобы правильно расставить кавычки. Ниже приведена работающая программа `bundle`, которая группирует файлы в выходной поток самодокументированного командного файла:

```
$ cat bundle
# bundle: группирует файлы в распределенный пакет
echo ’# Для разбиения на файлы вызовите sh с этим файлом’
for i
do
    echo "echo $i 1>&2"
    echo "cat >$i <<’End of $i’"
    cat $i
```

[1]Это старая адресация для UUNET сетей

```
    echo "End of $i"
done
$
```

Поскольку мы взяли в кавычки “`End of $i`”, любые метасимволы из файлов будут игнорироваться.

Естественно, что вам следует выполнить пробный запуск программы, чтобы не нанести ущерб адресату “`где-то!боб`”:

```
$ bundle cx lc >junk                    Пробный запуск bundle
$ cat junk
# Для разбиения на файлы вызовите sh с этим файлом
echo cx 1>&2
cat >cx <<'End of cx'
chmod +x cx
End of cx
echo lc 1>&2
cat >lc <<'End of lc'
# lc: подсчет числа строк в файлах
wc -l $*
End of lc
$ mkdir test
$ sh ../junk                            Попробуем
cx
lc
$ ls
cx
lc
$ cat cx
chmod +x $*
$ cat lc
# lc: подсчет числа строк в файлах
wc -l $* Похоже верно
$ cd ..
$ rm junk test/*; rmdir test            Удалим ненужное
$ pwd
/usr/you/bin
$ bundle `pick *` | mail где-то!боб     Посылка файлов
$
```

Здесь могут возникнуть трудности, если окажется, что один из посылаемых файлов содержит строку вида

```
End of имя_файла
```

но это маловероятное событие. Для обеспечения полной надежности программы нам потребуются некоторые из описываемых в последующих главах средства, однако и в таком виде она удивительно полезна и удобна.

Программа `bundle` является хорошим примером приспособляемости программного мира UNIX: в ней используются циклы языка `shell`, переключение ввода–вывода, конструкция "документ здесь" и командные файлы. Она непосредственно обращается к команде `mail`, и, что особенно интересно, порождает программу. Это одна из самых "красивых" среди известных вам `shell`–программ: файл в несколько строк предлагает простое и элегантное решение.

■ Упражнение: Как бы вы использовали `bundle` для посылки всех файлов с учетом вложенных каталогов? *Подсказка*: командные файлы могут быть рекурсивными.

■ Упражнение: Модифицируйте программу `bundle` так, чтобы к каждому файлу она добавляла информацию, выведенную командой `ls -l`, в частности права доступа и время его последнего изменения. Сравните возможности `bundle` и архивной программы `ar(1)`.

3.10 Для чего нужно программировать на языке `shell`!

Программа `shell` системы UNIX не относится к типичным интерпретаторам команд, хотя и дает возможность запускать команды обычным способом. Тем не менее это язык программирования, который позволяет достичь большего. Имеет смысл сделать ретроспективный обзор данной главы, поскольку здесь приведен довольно обширный материал, и, кроме того (что является главной причиной), мы обещали вам обсудить "средства общего пользования", а затем увлеклись примерами программирования на языке `shell`. Дело в том, что используя язык `shell`, вы все время пишите маленькие, практически однострочные программы, в частности конвейер — это программа, равноценная фразе "Чай готов". Однако вы выполняете свою работу так легко и естественно (если умеете), что даже не считаете ее программированием.

Интерпретатор дает вам такие средства, как циклы, переключение ввода–вывода с помощью `<` и `>`, порождение имен файлов с помощью `*`, причем применение этих средств единообразно во всех программах. Некоторые средства, например командные файлы и программные каналы, на самом деле обеспечиваются ядром, но язык `shell` предоставляет естественную запись для их создания. Они не только удобны, но и увеличивают мощность системы в целом.

Базой для интерпретатора служит ядро системы UNIX, например, хотя интерпретатор и определяет конвейеры, именно ядро осуществляет передачу данных по ним. Способ, которым система обрабатывает выполняемые файлы, позволяет программировать командные файлы так, чтобы они выполнялись подобно оттранслированным программам. Пользователь не должен думать о том, что это командные файлы, — для передачи их на выполнение не требуется специальная команда типа `RUN`. Сам интерпретатор является программой, а не частью ядра. Его можно настраивать, расширять и обращаться с ним, как с любой другой программой. Такой подход не является уникальным, но здесь он реализован полнее, чем где бы то ни было.

В гл. 5 мы вернемся к теме программирования на языке `shell`, а пока запомните: вне зависимости от того, как вы работаете с интерпретатором, вы программируете на его языке (чем в основном и объясняются его достоинства).

Историческая и библиографическая справка. На языке интерпретатора программируют с незапамятных времен. Сперва были отдельные команды для `if`, `goto` и меток, а команда `goto` выполнялась путем просмотра входного файла от его начала до нужной метки. (Поскольку невозможно читать заново через программный канал, нельзя было и передавать по программному каналу в командный файл, использующий структуры управления.)

Седьмая версия интерпретатора была создана С. Боурном, которому оказал помощь и идейную поддержку Д. Мэшей. Как вы увидите в гл. 5, здесь есть все необходимое для программирования. Кроме того, реорганизована работа с входным и выходным потоками: теперь можно без ограничения переключать ввод–вывод из командных файлов и в них. Неотъемлемым свойством интерпретатора является обработка метасимволов в именах файлов; в ранних версиях, которые остались лишь на очень маленьких машинах, она реализовывалась отдельной программой.

Другой вариант интерпретатора, с которым вы могли встречаться (а может быть, вы предпочитаете с ним работать) — `csh`, так называемый Си-`shell`, созданный Б. Джоем на базе интерпретатора шестой версии. По сравнению с интерпретатором Боурна этот интерпретатор лучше обеспечивает диалог. Он предоставляет средство “история”, позволяющее повторять в сокращенной записи (возможно, с небольшим редактированием) предварительно введенные команды. Отличается также и синтаксис команд интерпретатора Джоя. Но, поскольку Си-`shell`, базируется на интерпретаторе ранней версии, в нем содержится меньше средств для программирования; это скорее диалоговый интерпретатор команд, чем язык программирования. В частности, исключена передача по программному каналу из (или в) командного файла со структурами управления.

Команда `pick` предложена Т. Даффом, а команда `bundle` — независимо А. Хьитом и Д. Гослингом.

Глава 4

Фильтры

Существует большое число программ UNIX, которые читают входной поток, выполняют простые операции над ним и записывают результат в выходной поток. Примерами могут служить программы `grep` и `tail`, выбирающие часть входного потока, `sort`, сортирующая его, `wc`, производящая подсчет в нем, и т. д. Такие программы называются фильтрами.

В настоящей главе обсуждаются наиболее часто используемые фильтры. Первой мы рассмотрим программу `grep`, сосредоточившись на более сложных шаблонах, чем описанные в гл. 1, а затем две другие родственные программы — `egrep` и `fgrep`. Далее вы познакомитесь с еще несколькими полезными фильтрами, включая `tr`, который предназначен для транслитерации символов, `dd`, предназначенный для работы с данными, полученными из других систем, и `uniq` — для обнаружения повторяющихся строк. Приводится дополнительная информация и о программе `sort`.

Конец главы посвящен двум преобразователям данных общего назначения, или программируемым фильтрам. Они называются так потому, что конкретное преобразование записывается как программа на некотором простом языке программирования. Различные программы могут породить совершенно разные преобразования. Речь идет здесь о программах `sed` ("stream editor" — потоковый редактор) и `awk`, имя которой составлено из начальных букв имен ее авторов. Обе программы получаются путем обобщения команды `grep`:

```
$ программа шаблон-действие имена_файлов...
```

которая сканирует последовательность файлов, ведя поиск строк, совпадающих с шаблоном, — если строка найдена, выполняется необходимое действие. Для команды `grep`, как и для редактора `ed`, шаблоном является регулярное выражение, а действие по умолчанию сводится к печати каждой строки, соответствующей шаблону.

В программах `sed` и `awk` обобщаются и шаблоны, и действия. Команда `sed`, производная от `ed`, выполняет "программу", состоящую из команд редактирования. Она пропускает данные из файлов через эту программу, выполняя для каждой строки команды из программы. Команда `awk` не так удобна, как `sed`, для манипуляций с текстом, но в ней предусмотрены арифметические операции, переменные, встроенные функции и язык программирования, схожий с Си. В данной главе не приводится полное описание обеих программ; оно есть в т. 2B справочного руководства по UNIX.

4.1 Семейство программ grep

В гл. 1 мы кратко упомянули о команде `grep`, а затем использовали ее в примерах. Конструкция

```
$ grep шаблон имена_файлов
```

проводит поиск в поименованных файлах или в стандартном входном потоке и выводит на печать каждую строку, в которую входит шаблон. Команда `grep` неоценима для поиска переменных в программах и слов в документах, а также для выбора части выходного потока программы:

```
$ grep -n variable *.[гл]            Поиск variable в тексте на Си
$ grep From $MAIL                    Печать заголовков сообщений из почтовой
                                     посылки
$ grep From $MAIL | grep -v mary     Заголовки, которые получены не от
                                     адресата mary
$ grep -y mary $HOME/lib/phone-book  Поиск номера mary
$ who | grep mary                    Выяснить, работает ли mary в системе
$ ls | grep -v temp                  Имена файлов, не содержащих temp
```

Флаг `-n` инициирует вывод номеров строк, флаг `-v` меняет на противоположное значение условия, а флаг `-y` допускает сопоставление строчных букв из шаблона с прописными буквами из файла (но прописные буквы все-таки могут сопоставляться только с прописными).

Во всех рассматривавшихся до сих пор примерах проводился поиск обычных строк из букв и чисел. Но команда `grep` может искать и более сложные шаблоны: она интерпретирует выражения согласно простому языку для описания строк. С технической точки зрения шаблон представляет в некоторой степени ограниченную форму спецификаций строк, называемую регулярным выражением. Команда интерпретирует такие же регулярные выражения, как и редактор `ed`. На самом деле, эта команда была создана (за один вечер) прямым редактированием `ed`.

Регулярные выражения характеризуются тем, что ряду символов, таким, как `*` и т. п., приписывается специальное значение, используемое интерпретатором. Есть еще несколько метасимволов, но, к сожалению, с различными значениями. В табл. 4.1 показаны все метасимволы регулярных выражений, и мы кратко их здесь рассмотрим.

Метасимволы `^` и `$` привязывают шаблон к началу (`^`) или концу (`$`) строки. Например,

```
$ grep From $MAIL
```

ищет строки, содержащие `From` в вашей почтовой посылке, но

```
$ grep '^From' $MAIL
```

выдает строки, начинающиеся с `From`, которые, вероятнее всего, будут заглавными строками сообщений. Метасимволы регулярных выражений пересекаются с метасимволами интерпретатора, поэтому всегда имеет смысл заключать шаблоны команды `grep` в апострофы.

Команда grep допускает классы символов, подобные тем, что используются интерпретатором: так, [a-z] задает любую строчную букву. Но есть и различия — если класс символов команды grep начинается с символа слабого ударения ^, то шаблон задает любой символ, кроме входящих в данный класс. Значит, [^0-9] задает любой символ, кроме цифры. Как и в интерпретаторе, обратная дробная черта экранирует символы] и — в классе символов, но команды grep и ed требуют, чтобы эти символы использовались там, где их значение недвусмысленно. Например, шаблон []-] задает открывающую или закрывающую квадратную скобку либо знак минус.

Точка '.' эквивалентна '?' в интерпретаторе: она задает любой символ. (Точка, по всей видимости, есть символ, назначение которого различно для разных программ.) Ниже приводятся два примера:

```
$ ls -l | grep '^d'          Список имен вложенных каталогов
$ ls -l | grep '^......rw'   Список файлов, доступных всем для чтения и записи
```

Символ '^' и семь точек задают любые семь символов в начале строки; в случае применения к выходному потоку команды ls -l задается любая строка права доступа.

Операция "повторитель" ('*') применима в выражении к предваряющему ее символу или метасимволу (включая класс символов), и вместе они обозначают любое число вхождений символа или метасимвола. Например, x* задает последовательность букв x произвольной длины, [a-zA-Z] — любую строку букв, .* — все до конца строки, а .*x — все до последнего символа x в строке включительно. Необходимо отметить несколько важных моментов, связанных с повторителем. Во-первых, повторитель действует только на один символ, поэтому xy* соответствует x, за которым идут yy... , но не последовательности типа xyxyxy. Во-вторых, любое число включает нуль, поэтому если вы хотите, чтобы символ присутствовал, в шаблоне его нужно повторить. Например, правильным выражением, задающим строку букв, является такое: [a-zA-Z][a-zA-Z]* (буква, за которой следует нуль или более букв). Регулярное выражение .* соответствует — *, т. е. метасимволу интерпретатора, используемому для имен файлов.

Ни одно регулярное выражение команды grep не соответствует символу перевода строки; выражения сопоставляются с каждой строкой в отдельности. Регулярные выражения делают команду grep простым языком программирования. Вспомните, что второе поле файла паролей содержит зашифрованный пароль. Приведенная ниже команда проводит поиск пользователей, не имеющих пароля:

```
$ grep '^[^:]*::' /etc/passwd
```

Шаблон расшифровывается так: начало строки, любое число символов, отличных от двоеточия, два двоеточия.

Команда grep — старейшая в семействе программ, к которому относятся команды fgrep и egrep. В основном их действие одинаково, но fgrep может одновременно искать несколько литеральных строк, тогда как egrep интерпретирует настоящие регулярные выражения, подобно grep, но с использованием операций "or" и скобок для группировки выражений, что будет объяснено ниже.

Обе команды, fgrep и egrep, имеют флаг -f для указания файла, из которого читается шаблон. В этом файле символы перевода строк разделяют шаблоны при параллельном поиске. Допустим, что некоторые слова вы пишите неправильно. В этом случае можно проверить документацию на наличие таких слов, поместив их в файл по одному на строке и воспользовавшись командой fgrep:

```
$ fgrep -f типичные_ошибки документ
```

Регулярные выражения, интерпретируемые `egrep` (они также приведены в табл. 4.1), — те же самые, что и в `grep`, но с небольшими добавлениями. Можно использовать скобки для группировки, поэтому `(xy)*` задает пустую строку или любую последовательность `xy`, `xyxy`, `xyxyxy` и т. д. Вертикальная черта `|` является операцией or (или); `today|tomorrow` соответствует `today` или `tomorrow`, как и `to(day|morrow)`. Наконец, в команде `egrep` есть еще две операции повторения: `+` и `?` Шаблон `x+` задает один или более символов `x`, а шаблон `x?` — нуль или один символ `x` (но не более).

Команда `egrep` прекрасно подходит для игр, в которых нужно искать в словаре слова со специальными свойствами. Мы будем обращаться к словарю Вебстера (второе международное издание), хранящемуся в файле в виде списка слов по одному в строке без определений их значения. В вашей системе может быть небольшой словарь `/usr/dict/words`, предназначенный для проверки правописания; просмотрите его, чтобы выяснить формат. Ниже приведен шаблон, задающий слова английского языка, содержащие все пять гласных 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  abstentions
achelious   acheirous     acleistous
affectious  annelidous    arsenious
arterious   bacterious    caesious
facetious   facetiously   fracedinous
majestious
$
```

В файле `alphvowels` шаблон не взят в кавычки. Если применяются кавычки для экранирования шаблона в команде `egrep`, интерпретатор защищает его от интерпретации командами, но кавычки убирает, и команда `egrep` никогда "не узнает" о них. Поскольку интерпретатор не заглядывает в файл, кавычки не нужны для защиты содержимого файла. Для этого примера мы могли бы использовать команду `grep`, но алгоритм `egrep` таков, что она осуществляет поиск намного быстрее в случае шаблонов с повторителями, особенно при просмотре больших файлов.

В другом примере требуется найти все английские слова, состоящие из шести или более букв, в которых буквы следуют в алфавитном порядке:

```
$ cat monotonic
^a?b?c?d?e?f?g?h?i?j?fc?l?m?n?o?D?r?s?t?u?v?w?x?y?z?$
$ egrep -f monotonic /usr/dict/web2 | grep '......' | 5
abdest   acfcnow  adipsy   agnosy  almost
bedfist  behint   befcnow  bijoux  biopsy
chintz   dehors   dehort   demos   dimpsy
egilops  ghosty
```

(Egilops — это болезнь, поражающая пшеницу.) Обратите внимание на использование команды `grep` для фильтрации выходного потока `egrep`.

Для чего нужны три сходные программы? Программа `fgrep` не интерпретирует метасимволы, но может параллельно обрабатывать тысячи слов (после инициации время ее работы не зависит от числа слов), и поэтому она применяется прежде всего для заданий типа библиографического поиска. Размеры типичных шаблонов для программы `fgrep` превосходят возможности алгоритмов, используемых в программах `grep` и `egrep`. Различия между двумя последними указать труднее. Программа `egrep` появилась намного раньше. Она интерпретирует регулярные выражения, используемые в командах редактора `ed`, в ней есть помеченные регулярные выражения и большой набор флагов. Программа `egrep` интерпретирует более общие выражения (не считая помеченных), и выполняется значительно быстрее (со скоростью, не зависящей от шаблона), но ее стандартная версия требует большего времени на инициацию в случае сложного выражения. Существует новая версия, начинающая работу мгновенно, так что программы `egrep` и `grep` теперь можно было бы скомбинировать в одну программу поиска по шаблону.

`c`	Любой неспециальный символ c соответствует самому себе
`\c`	Указание убрать любое специальное значение символа c
`^`	Начало строки
`$`	Конец строки
`.`	Любой одиночный символ
`[...]`	Любой символ из `...`; допустимы диапазоны типа `a-z`
`[^...]`	Любой символ не из `...`; допустимы диапазоны
`\n`	Строка, соответствующая n-му выражению `\(...\)` (только для `grep`)
`r*`	Нуль или более вхождений `r`
`r+`	Одно или более вхождений `r`(только для `egrep`)
`r?`	Нуль или одно вхождение `r` (только для `egrep`)
`r1r2`	За `r1` следует `r2`
`r1|r2`	`r1` или `r2` (только для `egrep`)
`\(r\)`	Помеченное регулярное выражение `r` (только для `grep`); может быть вложенным
`(r)`	Регулярное выражение `r` (только для `grep`); может быть вложенным

Таблица 4.1: Регулярные выражения `grep` и `egrep` (в порядке убывания приоритета)

Никакое регулярное выражение не соответствует концу строки

■ Упражнение: Прочтите о регулярных выражениях (`\(` и `\)`) в приложении 1 или справочном руководстве по `ed(1)`. Используйте программу `grep` для поиска полиндромов — слов, читающихся одинаково с конца и начала. Подсказка: составьте свой шаблон для слов каждой длины.

■ Упражнение: Алгоритм программы `grep` таков: прочесть одну строку, проверить ее на вхождение шаблона, затем продолжить цикл. Как повлияло бы на работу программы то, что регулярные выражения могли бы задавать перевод строки?

4.2 Другие фильтры

Здесь мы представим вам набор небольших системных фильтров, покажем их возможности и дадим несколько примеров использования. Список этих фильтров далеко не полон — существует еще множество фильтров, входящих в седьмую версию, и, конечно, каждая работающая система имеет свои специфические фильтры. Все стандартные фильтры описаны в разд. 1 справочного руководства по UNIX.

Рассмотрим сначала программу `sort`, как наиболее часто используемую. В гл. 1 было указано ее назначение: сортировка входного потока по строкам в порядке, задаваемом множеством ASCII. Хотя это очевидный порядок для сортировки по умолчанию, существует множество других полезных способов сортировки данных, и программа `sort` пытается удовлетворить всех, предоставляя множество различных флагов. Например, флаг `-f` устраняет различие между прописными и строчными буквами, флаг `-d` (словарный порядок) игнорирует при сравнении все символы, кроме букв, цифр и пробелов.

Способ сравнения в алфавитном порядке является наиболее распространенным, но иногда требуется произвести сравнение в числовом порядке, флаг `-n` сортирует по числовому значению, а флаг `-r` изменяет смысл на противоположный любого условия. Итак, имеем

```
$ ls | sort -f          Сортировка имен файлов в алфавитном порядке
$ ls -s | sort -n      Сортировка в порядке возрастания размеров файлов
$ ls -s | sort -nr     Сортировка в порядке убывания размеров файлов
```

Программа `sort` обычно сортирует целые строки, но ее можно заставить работать только с определенными полями. Обозначение +m показывает, что при сравнении пропускается m полей, а `+0` обозначает начало строки, например:

```
$ ls -l | sort +3nr    Сортировка по счетчику байтов в порядке убывания
                       размеров
$ who | sort +4nr      Сортировка по времени входа в систему, в порядке
                       возрастания размеров файлов
```

Еще одним полезным флагом программы является `-o`, задающий имя файла для выходного потока (это может быть один из входных файлов), и флаг `-u`, который удаляет все, за исключением одной из строк, совпадающих в сортируемых полях. Можно использовать несколько флагов, как показано в примере на странице `sort(1)` справочного руководства:

```
$ sort +0f +0 -u filenames
```

здесь флаг `+0f` сортирует строку, совмещая строчные и прописные буквы, но идентичные строки могут не быть соседними. Поэтому вводится второй флаг `+0`, который сортирует одинаковые строки после первой сортировки в обычном порядке ASCII. Наконец, флаг `-u` выбрасывает все, кроме одной из соседних повторяющихся строк. Таким образом, получив список слов по одному в строке, команда выдает неповторяющиеся слова. Указатель для этой книги был подготовлен с помощью сходной команды `sort`, обладающей еще большими возможностями (см. руководство по `sort(1)`).

Создание команды `uniq` явилось стимулом для введения флага `-u` в команде `sort`: флаг отбрасывает все строки, кроме одной, из группы соседних повторяющихся строк.

Выведение отдельной программы для этой операции позволяет выполнять ее независимо от сортировки. Например, `uniq` удалит повторяющиеся пустые строки, независимо от того, сортируется входной поток или нет. Флаги предусматривают специальные способы обработки повторяющихся строк: `uniq -d` печатает только повторяющиеся строки, `uniq -u` — только уникальные, т. е. неповторяющиеся строки; `uniq -c` подсчитывает число вхождений каждой строки, в чем вскоре вы убедитесь на примере.

Программа `comm` служит для сравнения файлов. Получив два отсортированных входных файла `f1` и `f2`, она выдает выходной поток в три столбца: строки, встречающиеся только в `f1`, строки, встречающиеся только в `f2`, и строки, встречающиеся в обоих файлах. С помощью флага можно убрать любой из этих столбцов:

```
$ comm -12 f1 f2
```

выдает только строки, содержащиеся в обоих файлах, а

```
$ comm -23 f1 f2
```

выдает строки, которые есть только в первом, но не во втором файле. Это удобно для сравнения каталогов и списка слов со словарем.

Команда `tr` проводит транслитерацию символов своего входного потока. Наиболее часто они используются для преобразования строчных букв в прописные и обратно:

```
$ tr a-z A-Z        Перевести строчные буквы в прописные
$ tr A-Z a-z        Перевести прописные буквы в строчные
```

Несколько отличается от всех рассмотренных выше команд `dd`. Эта команда предназначена прежде всего для обработки данных на магнитной ленте, полученных из других систем — само ее название служит напоминанием о языке управлений заданиями OS/360. Команда `dd` выполняет преобразование прописных букв в строчные, и наоборот (в нотации, отличной от нотации команды `tr`). Она осуществляет перевод из множества символов ASCII в EBCDIC, и наоборот; может читать и писать данные в формате записей фиксированного размера с дополнением пробелами, что характерно для отличных от UNIX систем. На практике команду `dd` часто используют для работы с исходными неотформатированными данными, откуда бы они ни были получены; она реализует набор средств для работы с двоичными данными.

Посмотрим, чего можно достичь с помощью взаимодействия фильтров на примере конвейера, который печатает 10 наиболее часто встречающихся во входном потоке слов:

```
cat $* |
tr -sc A-Za-z '\012' |    Сжимаем все небуквы в перевод строки
sort |
uniq -c |
sort -n |
tail |
5
```

Команда cat собирает файлы, поскольку `tr` может читать только стандартный входной поток. Команда `tr` действует, как указано в справочном руководстве: она сжимает соседние, отличные от букв, символы в символы перевода строк, преобразуя таким образом входной поток в строки из одного слова. Затем слова сортируются и с помощью `uniq`

`-c` каждая группа идентичных слов сжимается в одну строку, начинающуюся со счетчика, который используется как сортируемое поле в команде `sort -n`. (Эта последовательность двух команд сортировки, между которыми находится команда `uniq`, применяется так часто, что уже стала идиомой.) В результате получаются неповторяющиеся слова, отсортированные в порядке возрастания частоты появления в документе. Команда `tail` отбирает 10 наиболее часто встречающихся слов (т. е. конец отсортированного файла) и команда 5 печатает их в пять столбцов.

Заметьте, кстати, что введение символа | в конце строки — это законный способ ее продолжения.

■ УПРАЖНЕНИЕ: Используя средства этого раздела и файл `/usr/dict/words`, составьте простой анализатор правильности написания текста на английском языке. Каковы его недостатки и как их исправить?

■ УПРАЖНЕНИЕ: Напишите программу подсчета слов на предпочитаемом вами языке программирования. Сравните ее размер, скорость и самодокументированность с соответствующим конвейером. Насколько легко вы можете преобразовать эту программу в программу проверки правильности написания текста?

4.3 Потоковый редактор `sed`

Вернемся теперь к редактору `sed`. Поскольку он происходит непосредственно от `ed`, вы легко изучите его и закрепите свои знания о редакторе `ed`. Основа редактора `sed` проста:

```
$ sed 'список команда ed' имена_файлов...
```

Читаются строки по одной из входных файлов; команды из списка применяются к каждой строке по одной в указанном порядке и результат редактирования записывается в стандартный выходной поток. Например, можно заменить в любом из указанных файлов `UNIX` на `UNIX (TM)` с помощью команды:

```
$ sed ' s/\UNIX/\UNIX\ (TM)/g' имена_файлов...> выходной поток
```

Нужно правильно понимать действие команды. Она не изменяет содержимое своих входных файлов, а лишь пишет в стандартный выходной поток; исходные же файлы не изменяются. Вы уже достаточно разобрались в интерпретаторе, чтобы понять, что

```
$ sed '...' файл > файл
```

не лучшее решение: для замены содержимого файлов нужно задействовать временный файл или другую программу. В дальнейшем мы рассмотрим программу, реализующую задачу записи в существующий файл; обратитесь к команде `overwrite` в (гл. 5).

Редактор `sed` выдает все строки автоматически, поэтому команда `p` не нужна после ввода команды замены, приведенной выше; более того, если она задается, то каждая изменяемая строка печатается дважды. Однако кавычки необходимы почти всегда, поскольку многие метасимволы программы `sed` имеют специальные значения и для интерпретатора. Рассмотрим, например, команду `du -a`, порождающую список имен файлов. Обычно она выдает размер и имя файла:

```
$ du -a ch4*
18 ch4.1
13 ch4.2
14 ch4.3
17 ch4.4
2  ch4.9
$
```

Можно использовать `sed`, чтобы отбросить размеры файлов, но в команде редактирования нужны кавычки для защиты символов * и табуляции от обработки интерпретатором:

```
$ du -a ch4.* | sed 's/.*->//'
ch4.1
ch4.2
ch4.3
ch4.4
ch4.9
$
```

В команде замены удаляются все символы (.*) до крайнего правого символа табуляции включительно (он показан в шаблоне как ->). Аналогичным способом можно выделить из вывода команды who имена пользователей и время входа в систему:

```
$ 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` заменяет пробел и все, что следует за ним (максимально возможно, включая дополнительные пробелы) до следующего пробела на единственный пробел. Кавычки нужны.

Почти такую же команду `sed` можно использовать, чтобы создать программу `getname`, возвращающую имя пользователя:

```
$ cat getname
who am i | sed 's/ .*//'
$ getname
you
$
```

Другая команда sed применяется настолько часто, что мы поместили ее в командный файл с именем ind. Эта команда вставляет пробелы до шага табуляции; она удобна для лучшего расположения текста при печати.

Реализовать команду ind просто: достаточно установить символ табуляции в начале каждой строки:

```
$ sed 's/^/->/' $*                    Первая версия ind
```

В этой версии символ табуляции вставляется даже в пустую строку, что не требуется. В лучшей версии используется возможность sed выбирать строки для модификации. Если команде предшествует шаблон, то изменяться будут только строки, соответствующие шаблону:

```
sed '/./s/^/->/' $*                   Вторая версия
```

Шаблон /./ задает любую строку, в которой есть по крайней мере один символ, кроме символа перевода строки, поэтому команда s не выполняется для пустых строк. Вспомним, что sed выдает все строки, независимо от того, менялись они или нет, так что пустые строки по-прежнему выдаются.

Есть еще один способ определения команды ind. Можно выполнять команды только для тех строк, которые не соответствуют выбираемому шаблону, предварив команду знаком восклицания '!'. В команде

```
sed '/^$/!s/^/->/' $*                 Третья версия
```

шаблон /^$/ задает пустые строки (перевод строки сразу следует за ее началом), поэтому /^$/! предписывает не выполнять команду для пустых строк.

Как уже отмечалось, sed печатает каждую строку автоматически, независимо от того, какие операции над ней выполнялись (если только она не была удалена). Кроме того, можно использовать большинство команд редактора ed. Поэтому легко составить программу sed, которая напечатает, скажем, три первых строки входного потока, а затем завершится:

```
sed 3q
```

Хотя 3q не является законной командой ed, для sed она имеет смысл: копировать строки и завершить выполнение после третьей.

От вас может потребоваться другая работа с данными, например вставка пробелов. Один из способов заключается в том, чтобы выходной поток sed пропустить через команду ind, но поскольку редактор sed допускает несколько команд, можно сделать это путем одного обращения к sed (хотя и несколько необычного):

```
sed 'S/^/->/
    3q'
```

Обратите внимание, где находятся кавычки и символ перевода строки: команды должны быть на отдельных строках, но редактор игнорирует пробелы и символы табуляции в начале строки.

Представляется естественным с помощью рассмотренных выше приемов составить программу head, которая будет печатать несколько строк из каждого своего файла-аргумента. Но sed 3q (или 10q) настолько просто задать, что в этом никогда не возникало

потребности. Однако мы ввели команду ind, так как соответствующая последовательность для sed длиннее. (В процессе работы над книгой мы заменили существовавшую программу на языке Си в 30 строк на одну строку команды ind версии 2, приведенной выше.) Четкого критерия в отношении того, когда имеет смысл создавать отдельную программу из сложной командной строки, нет, поэтому мы предлагаем вам свое решение: поместите программу в свой каталог /bin и посмотрите, будете ли вы ее действительно применять.

Можно помещать команды редактора sed в файл и выполнять их, получая оттуда с помощью обращения:

```
$ sed -f командный_файл
```

Вы можете обращаться к строкам, используя не только их номера. Так, команда

```
$ sed '/шаблон/q'
```

выдает из входного потока все строки до первой включительно, которые соответствуют шаблону, а команда

```
$ sed '/шаблон/d'
```

удаляет каждую строку, содержащую шаблон; удаление происходит до автоматического вывода строк, поэтому удаленные строки не учитываются.

Хотя автоматический вывод обычно удобен, иногда он мешает. Его можно отключить с помощью флага -n; в этом случае в выходном потоке появятся только строки, задаваемые явной командой вывода p. Например,

```
$ sed -n '/шаблон/p'
```

эквивалентен команде grep. Условие сопоставления можно инвертировать, если завершить его символом !, поэтому

```
$ sed -n '/шаблон/!p'
```

эквивалентно команде grep -v. (Так же, как sed '/шаблон/d'.)

Для чего нужны две команды sed и grep? В конце концов, grep-всего лишь частный случай команды sed. Это в какой-то степени объясняется историческими причинами: команда grep появилась намного раньше, чем команда sed. Но она не только уцелела, но и активно применялась. В силу специфики назначения обеих команд grep значительно проще использовать, чем команду sed, так как ее использование в типичных ситуациях настолько лаконично, насколько возможно (к тому же у нее есть возможности, отсутствующие у команды sed; см., например, описание флага -b). Но все-таки программы могут "умирать". Когда-то была программа с именем gres, выполняющая простую подстановку, но она исчезла почти мгновенно, когда появилась команда sed.

Используя запись, такую же, как в редакторе ed, можно вставлять символы перевода строк с помощью команды sed:

```
$ sed '/$/\
>/'
```

Здесь добавляется символ перевода строки к каждой строке, и таким образом пустые строки вставляются во входной поток, а команда

```
$ sed 's/[->][->]*/\
>/g'
```

заменяет каждую последовательность пробелов или символов табуляции на символ перевода строки, т. е. разбивает входной поток на строки из одного слова. (Регулярное выражение '[_>]' задает пробел или символ табуляции, '[_>]*' задает нуль или более таких символов, а весь шаблон — один или более пробелов и/или символов табуляции.)

Можно также использовать пару регулярных выражений или номеров строк для задания диапазона строк, к которому будет применяться произвольная команда.

```
$ sed -n '20,30p'        Печать только строк с 20-й по 30-ю
$ sed '1,10d'            Удаление строк с 1-й до 10-й (=tail +11)
$ sed '1,/^$/cd'         Удаление всех строк до первой пустой включительно
$ sed -n '/^$/,/^end/p'  Печать всех групп строк, начиная от пустой строки до
                         строки, начинающейся с end
$ sed '$d'               Удаление последней строки
```

Строки нумеруются с начала входного потока; обнуление не происходит с началом нового файла.

У команды sed есть существенное ограничение, которое, однако, отсутствует в редакторе ed: в ней поддерживается относительная нумерация строк. В частности, операции + и - не действуют в выражениях, задающих номера строк, поэтому невозможно двигаться назад во входном потоке:

```
$ sed '$-1d'               Недопустима обратная адресация
Unrecognized command: $-1d
$
```

Если строка считана, предыдущая исчезла навсегда: нет способа специфицировать предыдущую строку, а именно это требуется в команде. В принципе такой способ есть в команде sed, но он слишком изощренный. (См. команду hold в справочном руководстве по UNIX.) Невозможна и относительная прямая адресация:

```
$ sed '/что-то/+1d'        Недопустима прямая адресация
```

Редактор sed имеет возможность записывать в несколько выходных файлов. Например, команда

```
$ sed -n '/шабл/w файл1
>         /шабл/!w файл2' имена_файлов...
$
```

записывает строки, соответствующие "шабл", в файл1, а не соответствующие — в файл2, или, если вернуться к нашему первому примеру:

```
$ sed 's/\UNIX(TM)/gw u.out'  имена_файлов...> выход
```

то здесь, как и ранее, весь выходной поток записывается в файл “выход”, но к тому же измененные строки записываются в файл `u.out`.

Иногда нужна помощь со стороны интерпретатора, чтобы в команду редактора включить аргументы командного файла. Одним из примеров служит программа newer, которая выдает все более новые, чем заданный, файлы каталога:

```
$ cat newer
# newer f: список файлов, созданных после f
ls -t | sed '/'$1'$/q'
$
```

Кавычки защищают различные специальные символы, предназначенные для редактора, оставляя `$1` открытым для интерпретатора, чтобы он заменил его на имя файла. Существует альтернативный способ записи аргумента:

```
"/^$1\$/q"
```

так как `$1` заменяется на аргумент, тогда как `\$` становится просто `$`.

`a\`	Добавлять строки к выходному потоку, пока одна из них не закончится на \
`b label`	Перейти на команду: `label`
`c\`	Заменить строки на последующий текст, как в команде `a`
`d`	Удалить строку; прочесть следующую входную строку
`i\`	Вставить последующий текст перед следующим выходным потоком
`l`	Выдать строку, напечатав все невидимые символы
`p`	Выдать строку
`q`	Выйти
`r file`	Читать `file`, содержимое его переслать в выходной поток
`s/old/new/f`	Заменить `old` на `new`. Если `f=g`, заменить все вхождения; `f=p`, вывод; `f=w файл`, запись в файл
`t label`	Проверка: переход на метку, если была замена в текущей строке
`w file`	Записать строку в файл
`y/str1/str2/`	Заменить каждый символ строки `str1` на соответствующий символ строки `str2` (диапазоны недопустимы)
`=`	Выдать текущую нумерацию входной строки
`!cmd`	Выполнить команду `sed cmd`, только если строка не выбрана
`: label`	Установить метку для команд `b` и `t`
`{`	Команды до соответствующей скобки `}` рассматривать как группу

Таблица 4.2: Сводка команд `sed`

Аналогично можно составить программу `older`, которая выдает в качестве параметра все файлы, более старые, чем заданный:

```
$ cat older
# older f: список файлов, созданных ранее f
ls -tr | sed '/'$1'$/q'
$
```

Единственное различие состоит в применении флага `-r` в команде `ls` для изменения порядка выдачи файлов.

Хотя редактор `sed` способен на гораздо большее, чем мы вам продемонстрировали, включая проверку условий, циклы и ветвления, запоминание предыдущих строк, и, конечно, в нем допустимы многие команды редактора `ed`, описанные в приложении 1. Тем не менее в основном `sed` используется так, как было показано; одна или две простые команды редактирования, а не длинные и сложные последовательности. В табл. 4.2 собраны некоторые команды `sed`, хотя и не приведены операции над несколькими строками.

Редактор `sed` удобен потому, что позволяет работать с произвольно длинными входными строками. Это "быстрый" редактор, который сходен с редактором `ed` в интерпретации регулярных выражений и в обработке отдельных строк. Однако, с другой стороны, его возможности запоминания ограничены (трудно запомнить текст от одной строки до другой) — делается только один проход по данным, нельзя двигаться назад, нет способов прямой адресации типа `/.../+1`: и нет средств для работы с числами, т. е. он является чисто текстовым редактором.

■ УПРАЖНЕНИЕ: Измените команды `older` и `newer` так, чтобы они не включали файл–аргумент в свой выходной поток. Измените их так, чтобы файлы выдавались в обратном порядке.

■ УПРАЖНЕНИЕ: С помощью редактора `sed` сделайте программу `bundle` совершенно надежной. Подсказка: в конструкции "документ здесь" слово, отмечающее конец данных, распознается только в том случае, когда оно совпадает со строкой полностью.

4.4 Язык `awk` поиска и обработки шаблонов

Некоторые ограничения `sed` преодолены в программе `awk`. Принцип работы этой программы сходен с принципом работы программы `sed`, но синтаксически она ближе к языку программирования Си, чем к текстовому редактору. Способ задания команды такой же, как и для `sed`:

```
$ awk 'программа' имена_файлов...
```

но программа другая:

```
шаблон {действие}
шаблон {действие}
```

Программа `awk` читает входной поток по одной строке из указанных файлов. Строки сопоставляются с шаблонами по порядку; для каждого шаблона, соответствующего строке, выполняется необходимое действие. Как и в редакторе `sed`, входные файлы здесь не изменяются.

Шаблоны могут быть регулярными выражениями в `sed` или более сложными условиями, напоминающими язык Си. Приведем простой пример (такого же результата можно добиться с помощью команды `egrep`):

```
$ awk '/регулярное_выражение/ {print}' имена_файлов...
```

Печатается каждая строка, соответствующая регулярному выражению.

Шаблоны или действия могут отсутствовать. Если отсутствует действие, то по умолчанию печатаются строки, соответствующие шаблону, поэтому команда

```
$ awk '/регулярное_выражение/' имена_файлов...
```

эквивалентна предыдущей. Наоборот, если отсутствует шаблон, то действие выполняется для каждой входной строки. Следовательно, команда

```
$ awk '{print}' имена_файлов...
```

дает те же результаты, что и команда `cat`, хотя действует медленнее.

Теперь перейдем к более интересным примерам, но прежде сделаем одно замечание. Как и в случае `sed`, программу команды `awk` можно получать из файла:

```
$ awk -f кмд файл имена_файлов...
```

Поля. В программе `awk` каждая входная строка автоматически разбивается на поля, т. е. последовательности символов без пробелов, разделенные пробелами и символами табуляции. По этому определению выходной поток команды `who` имеет пять полей:

```
$ who
you tty2 sep 29 11:53
jim tty4 sep 29 11:27
$
```

Поля обозначаются как `$1`, `$2`, ..., `$NF`, где `NF` — переменная, значение которой установлено равным числу полей. В нашем случае `NF=5` для обеих строк. (Учтите разницу между `NF`, числом полей и `$NF` — последним полем строки. В отличие от интерпретатора в программе `awk` только номера полей начинаются с `$`; переменные не имеют такого префикса.) Например, следующая команда выдаст поле "размер файла" из результата выполнения команды `du -a`

```
$ du -a | awk '{print $2}'
```

а для печати имен пользователей, работающих в системе, и времени входа нужно задать:

```
$ who | awk '{print $1, $5}'
you 11:53
jim 11:27
$
```

Для печати имени и времени входа в систему, упорядоченных по времени, зададим:

```
$ who | awk '{print $5, $1}' | sort
11:27 jim
11:53 you
$
```

Это альтернативные решения примеров, приведенных выше в данной главе, в которых использовалась команда `sed`. Хотя с программой `awk` проще работать в подобных случаях, она обычно выполняется медленнее как в начальной фазе, так и при большом входном потоке.

Обычно предполагается, что поля разделяются произвольным числом пробелов и символов табуляций, но можно определить в качестве разделителя любой одиночный символ. Один из способов состоит в задании в командной строке флага `-F` (здесь прописная буква). Например, поля в файле паролей `/etc/passwd` разделяются двоеточиями:

```
$ sed 3q /etc/passwd
root:3D.fHR5KoB.3s:0:1:S.User:/:
ken:y.68wdl.ijayz:6:1:K.Thompson:/usr/ken:
dmr:z4u3dJWbg7wCk:7:1:D.M.Ritchie:/usr/dmr:
```

Для печати имен пользователей, образующих первое поле, можно задать:

```
$ sed 3q /etc/passwd | awk -F : '{print $1}'
root
ken
dmr
```

Обработка пробелов и символов табуляции здесь особая. По умолчанию и пробелы, и символы табуляции служат разделителями, а разделители в начале строки отбрасываются. Однако если в качестве разделителя определен не пробел, то разделители в начале строки учитываются при определении полей. В частности, если используется символ табуляции, то пробелы не являются символами–разделителями, пробелы в начале строки вводят в поле, и каждый символ табуляции определяет поле.

Печать. В программе `awk`, помимо числа входных полей, доступна и другая интересная информация. Встроенная переменная `NR` хранит номер текущей входной "записи", т. е. строки. Поэтому для вставки номера строки перед строкой входного потока достаточно задать:

```
$ awk '{print NR, $0}'
```

Поле `$0` обозначает всю входную строку без изменений. В операторе `print` фрагменты, отделяемые запятой, печатаются через символы разделения полей выходного потока, которые по умолчанию служат пробелами.

Формат печати оператора `print` обычно является приемлемым. При несоответствующем формате используйте оператор `printf`, обеспечивающий полный контроль над выходным потоком. Например, для печати номеров строк в поле размером в четыре цифры можно задать такую команду:

```
$ awk '{printf "%4d %s\n", NR, $0}'
```

Выражение `%4` задает десятичное целое число (`NR`) в поле размером в четыре цифры, `%S` — строка символов (`$0`), `\n` — символ перевода строки, который нужен потому, что оператор `printf` не выдает автоматически пробелы или символы перевода строк. Оператор `printf` сходен с аналогичной Си–функцией (см. справочное руководство по `printf(3)`).

Мы могли бы определить программу `ind` (рассматривавшуюся в начале главы) следующим образом:

```
$ awk '{printf "\t%s\n", $0}' $*
```

Здесь выдается символ табуляции (\t) и входная строка.

Шаблоны. Предположим, что вы хотите найти в файле /etc/passwd пользователей, не имеющих пароля. Зашифрованный пароль находится во втором поле, поэтому программа состоит из одного шаблона:

```
$ awk -F: '$2 == ""' /etc/passwd
```

Шаблон проверяет, является ли второе поле пустой строкой (операция == — это проверка на равенство).

Такой шаблон можно задать различными способами:

```
$2==""              Второе поле пусто
$2~/^$/             Второе поле соответствует пустой строке
$2!~/./             Второе поле не содержит ни одного символа
length($2) == 0     Длина второго поля равна нулю
```

Символ ~ обозначает соответствие регулярному выражению, а символ ! — отсутствие соответствия. Само регулярное выражение заключено в символы дробной черты.

Встроенная функция length программы awk вычисляет длину строки символов. Шаблону может предшествовать символ ! для отрицания его, например,

```
!($2=="")
```

Операция ! подобна такой же операции в языке Си, но в редакторе sed эта операция следует за шаблоном.

Наиболее типичное использование шаблонов в программе awk сводится к задачам простой проверки данных. Большинство из них немногим сложнее, чем поиск строк, не удовлетворяющих какому-то критерию; если нет выходного потока, то считается, что данные удовлетворяют соответствующему критерию (по принципу "отсутствие новостей — хорошая новость"). Например, в следующем шаблоне проверяется с помощью операции %, вычисляющей остаток от деления, четно или не четно число полей в каждой входной строке:

```
$ NF % 2 != 0                # напечатать, если нечетное число полей
```

Другой шаблон выдает исключительно длинные строки, используя встроенную функцию length:

```
length ($0) > 72             # напечатать, если слишком длинная строка
```

В программе awk используется то же соглашение о комментарии, что и в интерпретаторе: символ # отмечает начало комментария.

Можно сделать выходной поток более информативным, снабдив его предупреждающим сообщением и частью слишком длинной строки, используя для этого встроенную функцию substr:

```
length($0) > 72 {print "Строка", NR, "длинная" : substr ($0, 1,60)}
```

Функция substr(s, m, n) выделяет подстроку из строки s, начинающуюся с символа с номером m и длиной в n символов. (Символы в строке нумеруются с 1.) Если n отсутствует, то берется подстрока от m до конца строки. Эту функцию можно использовать для выделения полей с фиксированным положением, например выделить время в часах и минутах из результата выполнения команды date:

```
$ date
Thu Sep 29 12:17:01 EDT 1983
$ date | awk '{print substr($4, 1, 5) }'
12:17
$
```

■ УПРАЖНЕНИЕ: Сколько различных программ awk вы можете составить для переписи входного потока в выходной, как это делает команда cat? Какая из них самая короткая?

Шаблоны BEGIN и END. Программа awk имеет два специальных шаблона BEGIN и END. Действия, соответствующие BEGIN, выполняются прежде, чем читается первая входная строка; можно использовать этот шаблон для инициации переменных, печати заголовков или для установки символа разделителя полей, присваивая его переменной FS.

```
$ awk 'BEGIN { FS = ":" }
> $2 == "" ' /etc/paswd
$                              Результата нет: все работают с паролями
```

Действия, указанные в шаблоне END, выполняются после обработки последней входной строки:

```
$ awk 'END {print NR}'...
```

Здесь печатается число строк входного потока.

Арифметика и переменные. До сих пор в примерах выполнялись только простые операции с текстом. Достоинством программы awk является ее возможность попутно проводить вычисления над входными данными: что-нибудь подсчитать, вычислить суммы и средние значения и т. п. Типичный пример таких вычислений — подсчет суммы столбца чисел. Так, следующая команда складывает все числа первого столбца

```
    {s=s+$1}
END {print s}
```

Поскольку число значений доступно с помощью переменной NR, изменив последнюю строку на

```
END {print s, s/NR}
```

мы получим и сумму, и среднее значение.

Этот пример показывает, как используются переменные в awk. Переменная s не является встроенной, она определяется самим фактом использования. По умолчанию переменные инициируются нулем, так что, как правило, не нужно беспокоиться об их инициации.

В программе awk есть такие же сокращенные формы арифметических операторов, как и в языке Си, поэтому естественная запись примера имела бы вид:

```
    {s+=$1}
END {print}
```

Запись `s+=$1` равноценна записи `s=s+$1`, но более компактна.

Можно обобщить пример по подсчету входных строк:

```
    { nc+=length($0) + 1     # число символов, +1 для \n
      nw+= NF                # число слов
    }
END {print NR, nw, nc }
```

Здесь подсчитывается число строк, слов и символов входного потока, т. е. выполняются те же действия, что и по команде `wc` (хотя она и не разбивает общую сумму по файлам).

В качестве другого примера выполнения арифметических операций рассмотрим программу, подсчитывающую число страниц по 66 строк в каждой. Страницы получаются при прогоне несколько файлов через команду `pr`. Это можно оформить в виде команды `prpages`:

```
$ cat prpages
# prpages: подсчет числа страниц, выдаваемых pr
wc $* |
awk '!/total$/ { n += int(($1+55)/56) }
END { print n }
$
```

Команда `pr` помещает на каждую страницу 56 строк текста (это число определяется эмпирически). Для каждой строки вывода команды `wc`, которая не содержит слово `total` в конце строки, число страниц округляется, а затем выделяется целая часть с помощью встроенной функции `int`.

```
$ 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 этого результата запустим команды `pr` и `awk` одновременно:

```
$ pr ch4.* | awk 'END {print NR/66}'
53
$
```

Переменные программы awk могут также хранить строки символов. Рассматривать ли переменную как число или как строку символов — зависит от контекста. Грубо говоря, в арифметических выражениях типа s+=$1 используется числовое значение в контексте операций со строками типа x=="abc" — строковое значение в неясных случаях, например x>y, — строковое значение, если только операнды не являются явно числовыми. (Правила четко сформулированы в справочном руководстве по применению команды awk.) Строковые переменные инициируются пустой строкой. В последующих разделах строки будут активно использоваться.

В программе awk есть несколько своих встроенных переменных обоих типов, таких, как NR и FS. Их полный список приведен в табл. 4.3, а в табл. 4.4 перечислены операции, выполняемые командой.

■ Упражнение: Наша проверка программы prpages подсказывает иную реализацию этой программы. Поэкспериментируйте, чтобы выяснить, какая из них выполняется быстрее.

FILENAME	Имя текущего входного файла
FS	Символ разделения полей (по умолчанию приняты пробел и символ табуляции)
NF	Число полей входной строки
NR	Число входных строк
OFMT	Формат вывода чисел (по умолчанию принят %g; обратитесь к руководству по printf(3y))
OFS	Строка–разделитель полей в выходном потоке (пробел по умолчанию)
ORS	Строка-разделитель строк в выходном потоке (символ перевода строки по умолчанию)
RS	Символ разделения входных строк (символ перевода строки по умолчанию)

Таблица 4.3: Встроенные переменные awk

= += -= /= %=	Присваивание; v op=expr есть v=v op (expr)
||	ИЛИ: expr1 || expr2 истина, если одно или оба истинны; expr2 не вычисляется, если expr1 истинна
&&	И: expr1 && expr2 истина, если оба истинны; expr2 не вычисляется, если expr1 ложь
!	Отрицание значения выражения
> >= < <= == != ~ !~	Операция отношения; ~ и !~ это соответствие и несоответствие
пусто	Конкатенация строк
+ -	Сложение, вычитание
* / %	Умножение, деление, вычисление остатка
++ -	Увеличение, уменьшение (префиксное или постпрефиксное)

Таблица 4.4: Операции, выполняемые awk (в порядке возрастания приоритета)

Управление. При редактировании большого файла очень легко (судя по опыту) случайно создать копию соседнего слова, что практически никогда не происходит предна-

меренно. Для устранения таких ошибок в семействе программ **Writers Workbench** (рабочий набор редактора) существует программа `double`, отыскивающая пары идентичных соседних слов. Ниже показана реализация этой программы с помощью `awk`:

```
$ cat double
awk '
FILENAME != prevfile {   # new file
    NR = 1               # reset line number
    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
    if (NF > 0)
        lastword = $NF
}' $*
$
```

Операция `++` означает автоувеличение операнда, а операция `--` — его автоуменьшение.

Встроенная переменная `FILENAME` хранит имя текущего входного файла. Поскольку в переменной `NR` подсчитывается число строк с начала входного потока, мы изменяем ее значение всякий раз при изменении имени файла, чтобы точно указать строку с двойником.

Оператор `if` — такой же, как в языке Си:

```
if (условие)
    оператор1
else
    оператор2
```

Если условие верно, то выполняется `оператор1`; если оно ложно и если альтернативная часть присутствует, то выполняется `оператор2`. Альтернативная часть не обязательна.

Цикл `for` аналогичен таковому в языке Си, но отличается от цикла в языке `shell`:

```
for (выражение1; условие; выражение2)
    оператор
```

Цикл `for` идентичен приведенному ниже оператору, который также допустим в программе `awk`:

```
Выражение1 while (условие) {
    оператор
    выражение2
}
```

Например, конструкция

```
for (i=2; i <= NF; i++)
```

является циклом с i, принимающим значения 2, 3 и т. д., включая число полей NF.

Оператор break вызывает немедленный выход из цикла while или for; оператор continue инициирует переход к следующему шагу цикла (к условию в операторе while или к выражению2 в операторе for). Оператор next вызывает чтение следующей входной строки и сопоставление ее с шаблонами с начала программы awk, а оператор exit — немедленный переход на действия, определенные в шаблоне END.

Массивы. Как и в большинстве языков программирования, в awk есть массивы. В качестве простого примера приведем программу awk, в которой каждая входная строка заносится в отдельный элемент массива, индексируемого номером строки, а затем они печатаются в обратном порядке:

```
$ cat backwards
# backwards:  print input in backward line order
awk '   { line[NR] = $0 }
END     { for (i = NR; i > 0; i--) print line[i] } ' $*
```

Заметьте, что подобно переменным, массивы не нужно описывать; размер массива ограничен только объемом памяти, доступным на вашей машине. Конечно, если очень большой файл заносится в массив, в конце концов, это может привести к исчерпанию ресурсов памяти. Для печати конца большого файла в обратном порядке следует обратиться за помощью к команде tail:

```
$ tail -5 /usr/dict/web2 | backwards
zymurgy
zymotically
zymotic
zymosthenic
zymosis
$
```

Команда tail использует возможности файловой системы — операцию "поиск" (seeking), позволяющую перейти к концу файла без чтения всей предшествующей информации. Подробнее эта операция будет рассмотрена при обсуждении функции lseek в гл. 7. (В нашей команде tail есть флаг -r, который определяет печать строк в обратном порядке, заменяя команду backwards).

При обычной обработке входная строка разбивается на поля. Эту операцию можно выполнить с помощью встроенной функции split над любой строкой:

```
n = split (s, arr, sep)
```

Строка s разбивается на поля, записываемые в элементы массива arr от 1 до n. Используется символ разделения полей sep, если он задан; в противном случае применяется текущее значение переменной FS. Например, обращение split($0, a, ":") разбивает входную строку на столбцы, что подходит для обработки файла /etc/passwd, поэтому обращение split("9/29/83", date, "/") разбивает дату по символам дробной черты.

```
$ 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.

cos(expr)	Косинус expr
exp(expr)	Возведение в степень expr
getline()	Чтение следующей входной строки; возвращает 0 в случае конца файла, в противном случае 1
index(s1, s2)	Положение строки s2 в si; возвращает 0, если строка не входит
int(expr)	Целая часть expr; округляет по минимуму
length(s)	Длина строки s
log(expr)	Натуральный логарифм expr
sin(expr)	Синус expr
split(s, a, c)	Разбиение s на a[l] ...a[n] по символу c; возвращает n
sprintf(fmt, ...)	Форматирование в соответствии со спецификацией fmt
substr(s,m,n)	Подстрока в n символов строки s, начинающаяся с индекса m

Таблица 4.5: Встроенные функции awk

Ассоциативные массивы. Стандартной задачей обработки данных является получение суммарных значений для множества пар имя–значение. Иными словами, по входному потоку типа

```
Susie 400
John  100
Mary  200
Mary  300
John  100
Susie 100
Mary  100
```

мы хотим получить суммарные значения для каждого имени:

```
John 200
Mary 600
Susie 500
```

Программа awk предлагает изящное решение этой задачи — с помощью ассоциативных массивов. Хотя обычно мы представляем себе индекс массива как целое число, в awk любое значение можно использовать в качестве индекса. Поэтому

```
     {sum[$1] += $2}
END  {for (name in sum) print name sum [name]}
```

задает всю программу подсчета и печати сумм для пар имя–значение независимо от порядка следования этих пар. Каждое имя (`$1`) служит индексом в массиве `sum`; в конце применена специальная форма цикла `for` для перебора всех элементов `sum` и их печати. Синтаксис этого варианта цикла `for` таков:

```
for (перем in массив)
    оператор
```

Хотя он может показаться вам искусственным, как цикл `for` языка `shell`, они никак не связаны. Цикл охватывает индексы массива, а не его элементы, устанавливая значение "`перем`" равным каждому индексу поочередно. Однако порядок появления индексов непредсказуем, поэтому может возникнуть необходимость в их сортировке. В приведенном примере выходной поток можно по конвейеру передать команде `sort`, чтобы имена шли в порядке убывания значений:

```
$ awk '...' | sort +1nr
```

Реализация ассоциативной памяти предполагает хэширование, чтобы доступ к одному элементу занимал столько же времени, сколько и к любому другому, и чтобы это время не зависело (по крайней мере для массивов средних размеров) от числа элементов в массиве.

Использование ассоциативных массивов эффективно для вычислительных задач, таких, как подсчет частоты появления слов во входном потоке:

```
$ 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
.PI   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` выбирается каждое слово из входной строки и заполняется массив `num`, индексируемый словами. (Не путайте `$i`, обозначающее в `awk` i–е поле входной строки, с переменными языка `shell`.) После того как файл будет прочитан, во втором цикле `for` печатаются в произвольном порядке слова и частота их появления.

■ Упражнение: В результат действия команды `wordfreq` попали команды форматирования типа `.CW`, которые применяются для печати слов определенным шрифтом. Как избавиться от таких ненастоящих слов? Как бы вы использовали команду `tr`, чтобы программа `wordfreq` работала правильно, независимо от того, прописные или строчные буквы задействованы во входном потоке? Сравните реализацию и скорость выполнения программы `wordfreq`, конвейера из разд. 4.2 и предлагаемого ниже решения.

```
sed 's/[->][->]*/\
/q' $* | sort | uniq -c | sort -nr
```

Строки. Хотя обе команды, и `sed` и `awk`, предназначены для решения небольших задач типа выбора определенного поля, только `awk` используется в той степени, в какой предполагает настоящее программирование. Примером может служить программа, которая разбивает длинные строки, чтобы они занимали не более 80 позиций. Каждая строка, превышающая 80 символов, завершается после 80-го символа; в качестве предупреждения добавляется \ и обрабатывается остаток строки. Хвост разбиваемой строки сдвигается к ее правому концу, а не к левому, что более удобно для программ печати, и именно поэтому мы обратимся к программе `fold`. Рассмотрим, в частности, строки из 20, а не из 80 позиций:

```
$ cat тест
Короткая строка
Строка немного длиннее
Эта строка еще длиннее, чем предыдущая строка
$ fold тест
Короткая строка
Строка немного длиннее
Эта строка еще длиннее,
     чем предыдущая строка
$
```

Вам может показаться странным, что в седьмой версии системы нет программы для добавления или удаления символов табуляции, хотя команда `pr` в System V выполняет и то и другое. Наша реализация программы `fold` использует редактор `sed`, чтобы перевести символы табуляции в пробелы и чтобы счетчик числа символов в `awk` принял правильное значение. Это хороший способ при табуляции в начале строки (что типично для языковых программ), но номер позиции сбивается, если символ табуляции оказывается в середине строки:

```
# fold:  fold long lines
sed 's/\(->/    /g' $* |        # convert tabs to spaces
awk '
BEGIN {
    N = 80          # folds at column 80
    for (i = 1; i <= N; i++)        # make a string of blanks
        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)
   }
} '
```

На языке awk нет явной операции конкатенации строк; строки соединяются, если они следуют подряд. Вначале blanks является пустой строкой. Цикл в части BEGIN создает длинную строку пробелов конкатенацией: каждый шаг цикла прибавляет еще один пробел к концу строки blanks. Во втором цикле входная строка разбивается на части, пока оставшаяся часть не станет достаточно короткой. Как и в языке Си, операцию присваивания можно использовать в качестве выражения, поэтому в конструкции

```
if ((n=length ($0)) <= N)...
```

длина входной строки присваивается n до проверки значения. Обратите внимание на скобки.

■ УПРАЖНЕНИЕ: Упражнение 4.10. Измените программу fold так, чтобы разрыв строки происходил на пробеле или символе табуляции, а не посреди слова. Сделайте эту программу пригодной и для длинных слов.

Взаимодействие с интерпретатором. Допустим, что вы намереваетесь написать программу field n. Эта программа будет печатать n-е поле каждой входной строки так, чтобы можно было, например, задать:

```
$ who | field 1
```

для печати только имен, под которыми пользователи входят в систему. Язык awk явно предоставляет возможность выбора полей. Наша основная задача — передать номер n программе awk. Ниже приведено одно из возможных решений:

```
$ awk '{print $'$1'}'
```

Здесь $1 открыто (не внутри каких–либо кавычек), и поэтому становится номером поля, доступным в программе awk. При ином решении используются кавычки:

```
awk "{print \$$1}"
```

Аргумент обрабатывается интерпретатором, поэтому \$ становится $, а $1 заменяется на значение n. Мы предпочитаем решение с апострофами (одиночными кавычками), поскольку при использовании кавычек в типичной программе awk появится слишком много символов \.

Другим примером может служить программа addup n, суммирующая значения n-го поля:

```
awk '{s += $'$1'}
END {print s}'
```

В третьем примере вычисляются отдельные суммы значений каждого n-го поля и полная сумма:

```
awk '
BEGIN { n = '$1' }
{ for (i=1; i <= n; i++)
    sum[i] += $1
```

```
}
END { for(i = 1; i <= n; i++)
    printf "%6g ", sum[i]
    total += sum[i]
}
printf "; total = %6g ", total
}'
```

Нам удобнее было использовать часть BEGIN для засылки значения в переменную n, чем засорять конец программы кавычками.

Основная трудность во всех приведенных выше примерах состоит не в том, чтобы следить за кавычками (хотя и это хлопотно), а в том, что программы, составленные показанным способом, могут читать только свой стандартный входной поток. Нет никакой возможности передать им сразу и параметр n, и произвольно длинный список имен файлов. Для этого требуется определенная техника программирования на языке shell; которую мы рассмотрим в следующей главе.

Служебная программа–календарь на языке awk. В нашем последнем примере демонстрируются ассоциативные массивы, а также иллюстрируется взаимодействие с интерпретатором и частично показывается процесс разработки программы.

Задача состоит в создании системы, посылающей вам каждое утро почту с напоминанием об ожидаемых событиях. (Возможно, такая календарная система уже есть; см. руководство по calendar(1).) В этом разделе применяется иной подход. Вам будут перечислены события, происходящие сегодня и, кроме того, предстоящие сегодняшние и завтрашние события. Правильный учет праздников и выходных оставлен вам в качестве упражнения.

Прежде всего нужно предусмотреть место, где будет храниться календарь. Имеет смысл разместить его в файле с именем calendar в каталоге /usr/you:

```
$ cat calendar
Sep 30 день рождения мамы
Oct  1 обед с Джо, полдень
Oct  1 встреча в 16:00
```

Далее, необходимо уметь просматривать календарь, отыскивая определенную дату. Существует масса вариантов; мы остановимся на языке awk, поскольку с его помощью легче выполнять арифметические операции по переходу от одной даты к другой, однако для этой цели подходят и другие программы, например sed и egrep. Конечно, строки, выбранные из файла calendar, посылаются командой mail.

Наконец, вам придется научиться автоматически и безотказно просматривать календарь каждый день, скажем, рано утром. Это можно сделать с помощью команды at, о которой упоминалось в гл. 1.

Если ограничить календарь таким форматом, в котором каждая строка начинается с названия месяца и числа (как это делает команда date), то составить первый вариант программы–календаря нетрудно:

```
$ date
Thu Sep 29 15:23:12 EDT 1983
$ cat bin/calendar
# calendar:  version 1 -- today only
```

```
awk <$HOME/calendar '
        BEGIN { split("'"`date`"'", date) }
        $1 == date[2] && $2 == date[3]
' | mail $NAME
$
```

Функция в части BEGIN разбивает дату, выдаваемую командой date, и заносит ее в массив; второй и третий элементы массива — месяц и число. Мы предполагаем, что в переменной интерпретатора NAME находится имя, под которым вы вошли в систему. Вы заметили, какая нужна сложная последовательность кавычек, чтобы "поймать" результат действия команды date в середине строки программы awk. Более простым решением является передача даты в первой строке входного потока:

```
$ cat /bin/calendar
# calendar:  version 2 -- today only, no quotes
(date; cat $HOME/calendar) |
awk '
  NR == 1    { mon = $2; day = $3 } # set the date
  NR > 1 && $1 == mon && $2 == day  # print calendar lines
' | mail $NAME
$
```

На следующем шаге требуется так изменить программу, чтобы искать сообщение с завтрашней датой так же, как и с сегодняшней. Наибольшие усилия затрачиваются на прибавление единицы к сегодняшнему числу. Но в конце месяца нужно перейти к следующему месяцу, а число приравнять единице. Конечно, число дней в разных месяцах различно. Именно здесь на помощь приходит ассоциативный массив. Два массива days и nextmon, индексами которых служат названия месяцев, содержат число дней месяца и название следующего месяца. Например, days["Jan"] равно 31, а nextmon["Jan"] есть Feb. Вместо того чтобы написать множество операторов типа

```
days["Jan"] = 31; nextmon["Jan"] = "Feb"
days["Feb"] = 28; nextmon["Feb"] = "Mar"
```

мы воспользуемся функцией split для преобразования удобно записываемой структуры данных в то, что требуется:

```
$ cat calendar
# calendar:  version 3 -- today and tomorrow
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 на то, что `Jan` появляется дважды в структуре данных; такое "сторожевое" значение упрощает обработку для декабря.

На последнем шаге нужно обеспечить запуск программы–календаря на каждый день. Можно делать это и самому, не забывая задавать команду (каждый день!)

```
$ at 5 am
calendar
ctl-d
$
```

Однако такое решение нельзя считать автоматическим или надежным. Хитрость заключается в том, чтобы не только запустить программу `calendar`, но и обеспечить следующий ее запуск.

```
$ cat early, morning
calendar
echo early morning | at 5am
$
```

Вторая строка файла `early`, `morning` готовит еще одну команду `at` для следующего дня, поэтому, раз начавшись, эта последовательность команд сама себя воспроизводит. В команде `at` устанавливается `PATH`, текущий каталог и другие параметры запускаемых ею команд, так что больше ничего и не требуется.

■ Упражнение: Измените программу `calendar` так, чтобы она учитывала выходные дни: для пятницы "завтра" должно означать субботу, воскресенье или понедельник. Далее измените ее так, чтобы можно было учесть високосные годы. Следует ли учитывать праздники? Как бы вы это сделали?

■ Упражнение: Должна ли программа–календарь учитывать даты, находящиеся в середине строки, а не только в ее начале? Как быть с датой, заданной в другом формате, например `10/1/83`?

■ Упражнение: Почему в программе `calendar` используется `$NAME`, а не обращение к `getname`?

■ Упражнение: Напишите вашу версию команды `rm`, которая не удаляет файлы, а пересылает их во временный каталог, используя команду `at` для очистки каталога в то время, пока вы не работаете.

Дополнительная информация. Язык `awk` довольно громоздкий, и в рамках одной главы трудно показать все его возможности. Поэтому мы перечислим здесь еще ряд моментов, на которые необходимо обратить внимание в справочном руководстве:

Переключение выходного потока оператора `print` в файлы и программные каналы: за каждым оператором `print` или `printf` может следовать символ `>` и имя файла (в виде строки в кавычках или переменной); выходной поток будет направлен в этот файл. Как и для интерпретатора, `>>` означает добавление, а не запись. Для вывода в программный канал используется символ `|`, а не `>`.

Запись в несколько строк: если разделитель записей `RS` установлен равным концу строки, то входные записи разделяются пустой строкой. В таком случае несколько входных строк могут рассматриваться как одна запись.

"Шаблон, шаблон" в качестве селектора: как и в случае команд `sed` и `ed`, с помощью пары шаблонов можно указать диапазон строк. Так выбираются строки, начиная с соответствующей первому шаблону, до строки, соответствующей второму шаблону. Приведем простой пример:

```
NR == 10, NR == 20
```

Здесь задаются строки от 10-й по 20-ю включительно.

4.5 Хорошие файлы и хорошие фильтры

Несмотря на то что в качестве примеров использования языка `awk` приводились независимые программы, в большинстве случаев его применяют для написания простых программ в одну или две строки, являющихся фильтрами в больших конвейерах. Это справедливо для большинства фильтров: редко поставленная задача может быть решена с помощью одного фильтра, чаще она разбивается на подзадачи, где фигурируют несколько фильтров, объединенных в конвейер. Такую реализацию программных компонентов называют основным принципом организации программного мира UNIX. Фильтры буквально "пронизывают" всю систему, и очень важно понимать причины этого.

Программы UNIX выдают выходной поток в таком формате, что его можно использовать в качестве входного потока для других программ. Файлы, пропускаемые через фильтр, состоят из строк текста, свободных от декоративных заголовков, завершителей или пустых строк. Каждая строка представляет интерес — это имя файла, слово, описатель выполняемого процесса, поэтому программы типа `wc` или `grep` могут рассчитывать определенные характеристики объектов или искать их по именам. Если о каждом объекте имеется большая информация, файл все равно состоит из строк, разбиваемых на поля пробелами или символами табуляции, как в выводе команды `ls -l`. Располагая данными, разбитыми на такие поля, программы типа `awk` могут легко выбрать, обработать или переупорядочить информацию.

Фильтры построены по общей схеме. Каждый из них пишет в стандартный выходной поток результат обработки файлов-аргументов или стандартного выходного потока, если аргументов нет. Аргументы задают входной поток и никогда не задают выходной (Ранняя версия файловой системы UNIX была уничтожена служебной программой, нарушившей

это правило, поскольку команда, которая выглядела безобидной, расписала весь диск.), поэтому выходной поток команда всегда может передать в конвейер. Необязательные аргументы (или аргументы, не являющиеся файлами, такие, как шаблон в команде `grep`) задаются перед именем файлов. Наконец, сообщения об ошибках пишутся в стандартный поток диагностики, поэтому они не могут исчезнуть в конвейере.

Эти соглашения не оказывают большого влияния на программы пользователя, но единообразное применение их ко всем программам обеспечивает простоту взаимодействия, что подтверждается многочисленными примерами на протяжении всей книги и наиболее наглядно продемонстрировано программой подсчета слов в конце разд. 4.2. Если каким-либо программам потребуется входной или выходной файл с конкретным именем, определенное обращение для спецификации параметров или создание заголовков и завершений, то схема конвейера работать не будет. И конечно, если бы система UNIX не предоставляла программные каналы, кому-то пришлось создать подобное стандартное средство. Однако программные каналы есть, и конвейеры работают. Их даже можно запрограммировать, но для этого вы должны знать возможности системы.

■ УПРАЖНЕНИЕ: Команда `ps` выдает поясняющий заголовок, а команда `ls -l` сообщает общее число блоков файла. Прокомментируйте действие команд.

Историческая и библиографическая справка. Хороший обзор алгоритмов сопоставления шаблонов дается в статье Э. Ахо, создателя команды `egrep`, "Pattern matching in strings" (Proceedings of fhe Symposium on Formul Language Theory, Santa Barbara, 1979). Редактор `sed` разработан и реализован на базе редактора `ed` Л. Мак-Махоном. Язык `awk` был разработан и реализован Э. Ахо, П. Вайнбергером и Б. Керниганом, но это решение не очень элегантно. К тому же выбор названия языка по первым буквам имен создателей представляется не вполне удачным. Проект обсуждался в статье авторов "AWK — a pattern scanning and processing language" (Software-Practice and Experience, July, 1978). Язык `awk` имеет несколько источников, но, безусловно, некоторые идеи заимствованы из языка Снобол4, редактора `sed`, языка проверки условий, разработанного М. Рочкиндом, языковых средств `yacc` и `lex` и, конечно, языка Си. В действительности сходство между `awk` и Си порождает ряд проблем. Язык подобен Си, но они не совпадают: одни конструкции в `awk` отсутствуют, другие отличаются от соответствующих конструкций Си неочевидным образом.

В статье Д. Комера "The flat file system FFG: a database system consisting of primitives". (Software — Practice and Experience, Nov., 1982) обсуждается использование интерпретатора и `awk` для создания системной базы данных.

Глава 5

Программирование на языке shell

Большинство пользователей считают, что shell представляет собой диалоговый интерпретатор команд. На самом же деле это язык программирования, в котором каждый оператор инициирует запуск команды. Язык shell может показаться вам несколько странным, поскольку в нем находят отражение и диалоговый, и программный аспекты выполнения команд. Он формировался по плану, хотя и имеет свою историю. Разнообразие применений языка привело к некоторой незавершенности в деталях, но для его эффективного использования вам и не нужно разбираться во всех нюансах. В данной главе мы рассмотрим основы программирования с помощью shell на примерах разработки ряда полезных программ. При изучении материала желательно иметь под рукой страницу sh(1) справочного руководства по UNIX.

Для интерпретатора, как и для многих других команд, особенности выполнения наиболее четко проявляются в ходе эксперимента. В справочном руководстве что-то может оказаться неясным, и здесь на помощь вам придет хороший пример. По этой причине материал главы построен на примерах, которые демонстрируют возможности языка. Мы будем обсуждать не только вопросы программирования с помощью интерпретатора, но и проблемы создания программ на языке shell, уделяя особое внимание тестированию и диалогу.

Если вы написали программу на языке shell или каком-то ином языке, она может оказаться полезной и другим пользователям вашей системы. Однако требования, которым, по мнению других, должна удовлетворять программа, обычно оказываются более строгими, чем предъявляемые к ней вами. Поэтому важнейший аспект программирования на языке shell — обеспечение надежности программы, чтобы она могла выполняться даже при неверно заданных входных данных и выдавать полезную информацию об ошибках.

5.1 Совершенствование команды cal

Типичная задача программирования на языке shell сводится к изменению взаимодействия между пользователем и программой, чтобы сделать это взаимодействие более удобным. В качестве примера рассмотрим команду cal(1):

```
$ cal
usage: cal [month] year                    Пока хорошо
$ cal october
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 нужно задавать числом, и к тому же, как оказалось, команда `cal 10` выдает календарь на весь 10-й год, а не на октябрь текущего года. Поэтому всегда следует указывать год, если вы хотите получить календарь на один месяц.

Указанные выше неудобства связаны с тем, что взаимодействие пользователя с программой было реализовано без привлечения команды `cal`. Вы можете изменить характер этого взаимодействия, не меняя самой программы. Если поместить команду в ваш собственный каталог `bin`, то возможен более удобный способ перевода аргументов в те, которые нужны настоящей команде `cal`. Вы можете даже вызывать свою версию команды, и тогда вам меньше придется запоминать.

Первый шаг разработки — определить функции усовершенствованной команды `cal`. В основном мы хотим от нее разумного поведения. Месяц нужно распознавать по названию. При наличии двух аргументов она должна делать то же, что делала прежняя версия, за исключением перевода названия месяца в его номер. В случае одного аргумента следует печатать календарь месяца или года (в зависимости от того, что вам требуется), а при отсутствии аргументов — календарь текущего месяца, так как большей частью именно для этого и обращаются к команде. Поэтому задача сводится к тому, чтобы определить, сколько аргументов задано, и преобразовать их в те параметры, которые требуются стандартной команде `cal`.

Язык `shell` имеет оператор `case`, который успешно применяется в таких ситуациях:

```
case слово in
шаблон) команды ;;
шаблон) команды ;;
...
esac
```

В операторе `case` слово сравнивается поочередно со всеми шаблонами от начала до конца и выполняются команды, связанные с первым (и только первым) шаблоном, соответствующим слову. Шаблоны составляются по правилам соответствия шаблонов, которые в некоторой степени обобщают правила задания имен файлов. Каждое действие завершается двумя символами `;;` (для последнего варианта можно обойтись без `;;`, но обычно мы ставим их для удобства редактирования).

В нашей версии команды определяется число заданных аргументов, обрабатываются названия месяцев, затем происходит обращение к настоящей команде `cal`. В переменной интерпретатора `$#` хранится число аргументов, с которыми была вызвана программа; другие специальные переменные интерпретатора перечислены в табл. 5.1.

```
$ cat cal
```

$#	Число аргументов
$*	Все аргументы, передаваемые интерпретатору
$@	Аналогично $*; см. разд. 5.7
$-	Флаги, передаваемые интерпретатору
$?	Возвращение значения последней выполненной команды
$$	Номер процесса интерпретатора
$!	Номер процесса последней команды, запущенной с помощью &
$HOME	Аргумент, принятый по умолчанию для команды cd
$IFS	Список символов, разделяющих слова в аргументах
$MAIL	Файл, изменение которого приводит к появлению сообщения "you have a mail" ("У вас есть почта")
$PATH	Список каталогов, в которых осуществляется поиск команд
$PS1	Строка–приглашение, по умолчанию принята '$'
$PS2	Строка–приглашение при продолжении командной строки, по умолчанию принята '>'

Таблица 5.1: Встроенные переменные интерпретатора

```
# cal:  nicer interface to /usr/bin/cal

case $# in
0)      set `date`; m=$2; y=$6 ;;  # no args: use today
1)      m=$1; set `date`; y=$6 ;;  # 1 arg: use this year
*)      m=$1; y=$2 ;;              # 2 args: month and year
esac

case $m in
jan*|Jan*)      m=1 ;;
feb*|Feb*)      m=2 ;;
mar*|Mar*)      m=3 ;;
apr*|Apr*)      m=4 ;;
may*|May*)      m=5 ;;
jun*|Jun*)      m=6 ;;
jul*|Jul*)      m=7 ;;
aug*|Aug*)      m=8 ;;
sep*|Sep*)      m=9 ;;
oct*|Oct*)      m=10 ;;
nov*|Nov*)      m=11 ;;
dec*|Dec*)      m=12 ;;
[1-9]|10|11|12) ;;              # numeric month
*)              y=$m; m="" ;;   # plain year
esac

/usr/bin/cal $m $y              # run the real one
$
```

В первом операторе case проверяется число аргументов $# и выбирается подходящее действие. Последний шаблон в этом операторе задает вариант, выбираемый по умолчанию; если число аргументов не 0 и не 1, будет выполнен последний вариант. (Поскольку

шаблоны просматриваются по порядку, вариант по умолчанию должен быть последним.) При наличии двух аргументов `m` и `y` принимают значение месяца и года, и наша команда `cal` должна выполняться как исходная команда.

Первый оператор `case` включает пару нетривиальных строк, содержащих

```
set ‘date‘
```

Хотя это сразу и не очевидно, легко установить действие команды, запустив ее:

```
$ date
Sat Oct 1 06:05:18 EDT 1983
$ set ‘date‘
$ echo $1
Sat
$ echo $4
06:05:20
$
```

Итак, мы имеем дело с встроенной командой интерпретатора, возможности которой многообразны. При отсутствии аргументов она выдает, как указывалось в гл. 3, значения переменных окружения. В случае обычных аргументов переопределяются значения `$1`, `$2` и т. д. Поэтому `set ‘date‘` присваивает `$1` — день недели, `$2` — название месяца и т. д. Таким образом, при отсутствии аргументов в первом `case` месяц и год устанавливаются из текущей даты. Если был задан один аргумент, он используется в качестве месяца, а год берется из текущей даты.

Команда `set` имеет также несколько флагов, из которых наиболее часто используются флаги `-v` и `-x` — для отключения эха команд при обработке их интерпретатором. Такое отключение может оказаться необходимым в процессе отладки сложных программ на языке `shell`.

Теперь осталось только перевести значение месяца, если оно представлено в строковом виде, в число. Это делается с помощью второго оператора `case`, который практически очевиден. Единственный нюанс состоит в том, что символ `|` в шаблонах оператора case, как и в команде `egrep`, означает альтернативу: `малый|большой` соответствует варианту “`малый`” или “`большой`”. Конечно, эти варианты можно было бы задать с помощью `[jJ]an*` и т. д. Программа допускает задание месяца строчными буквами, поскольку большинство команд работает с входным потоком, где данные записаны строчными буквами (иногда первая буква — прописная), поскольку так выглядит вывод команды `date`. Правила сопоставления шаблонов приведены в табл. 5.2.

Два последних варианта второго оператора case относятся к случаю, когда единственный аргумент может быть годом; напомним, что в первом операторе `case` предполагалось, что аргументом является месяц. Если это число, которым может быть задан месяц, то ничего не происходит (иначе предполагается, что задан год).

Наконец, в последней строке вызывается `/usr/bin/cal` (настоящая команда `cal`) с преобразованными аргументами. Наша версия команды `cal` работает так, как этого мог бы ожидать начинающий:

```
$ date
Sat Oct 1 06:09:55 EDT 1983
$ cal
```

*	Задает любую строку, включая пустую
?	Задает любой одиночный символ
[ccc]	Задает любой из символов в ccc [a-d0-3] эквивалентно [abcd0123]
"..."	Задает в точности ...; кавычки защищают от специальных символов. Аналогично действует '...'
\c	Задает с буквально
a|b	Только для выражений выбора; задает a или b
/	Для имен файлов; соответствует только символу / в выражении; для выражений выбора сопоставляется, как любой другой символ
.	Если это первый символ в имени файла, то сопоставляется только с явно заданной точкой в выражении

Таблица 5.2: Правила сопоставления шаблонов в интерпретаторе

```
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
$
```

При обращении к cal 1984 будет напечатан календарь на весь 1984 год. Наша обобщенная команда cal выполняет то же задание, что и исходная, но более простым и легко запоминающимся способом. Поэтому мы предпочитаем называть ее cal, а не calendar (что уже является командой), или как-нибудь еще с менее простой мнемоникой, например ncal. При использовании одного и того же имени пользователю не придется вырабатывать новые рефлексы для печати календаря.

Прежде чем завершить обсуждение оператора case, следует объяснить, почему правила сопоставления шаблонов в интерпретаторе отличаются от правил для редактора ed и его производных. Действительно, наличие двух видов шаблонов означает, что нужно изучать два набора правил и иметь два программных фрагмента для их обработки. Некоторые различия вызваны просто неудачным выбором, который никогда не был зафиксирован. В частности, нет никаких причин (кроме того, что так сложилось исторически), по которым ed использует '.' а интерпретатор — '?' для задания единственного символа. Но иногда шаблоны применяются по-разному. Регулярные выражения в редакторе используются для поиска последовательности символов, которая может встретиться в любом месте строки; специальные символы ^ и $ нужны, чтобы направить поиск с начала или конца строки. Но для имен файлов мы хотим, чтобы направление поиска

определялось по умолчанию, поскольку это наиболее общий случай. Было бы неудобным задавать нечто вроде

```
$ ls ^?*.c$                          Так не получится
```

вместо

```
$ ls *.c
```

■ Упражнение: Если пользователи предпочтут вашу версию команды `cal`, как бы вы сделали ее общедоступной? Что следует предпринять, чтобы поместить ее в `/usr/bin`?

■ Упражнение: Имеет ли смысл сделать так, чтобы при обращении `cal 83` был напечатан календарь за 1983 г.? Как в этом случае задать вывод календаря?

■ Упражнение: Модифицируйте команду `cal` так, чтобы можно было задавать больше одного месяца, например:

```
$ cal oct nov
```

и даже диапазон месяцев:

```
$ cal oct-dec
```

Если сейчас декабрь, а вы выполняете обращение `cal` jan, то какой должен быть напечатан календарь: на январь этого года или следующего? Когда следует прекратить расширять возможности команды `cal`?

5.2 Что представляет собой команда `which`?

При обзаведении собственными версиями команд, аналогичных `cal`, возникает ряд трудностей. В частности, когда вы работаете как пользователь Мэри и вошли в систему под именем mary, то, вводя команду `cal`, получаете стандартную версию команды вместо новой, если, конечно, не установили в своем каталоге `bin` связь с новой командой `cal`. Это может привести к путанице: вспомните, что сообщения об ошибках в исходной, команде `cal` не очень вразумительны. Мы привели всего лишь один пример возникающих трудностей. Поскольку интерпретатор осуществляет поиск команд среди каталогов, задаваемых переменной `PATH`, всегда есть вероятность столкнуться не с той версией команды, которую вы ожидали. Например, если вы задали команду, скажем `echo`, имя выполняемого на самом деле файла будет `./echo`, `/bin/echo`, `/usr/bin/echo` или какое-то другое в зависимости от компонентов вашей переменной `PATH` и от того, где находятся файлы. Может случиться, что в вашей последовательности поиска ранее, чем вы ожидали, окажется выполняемый файл с правильным именем, но не с теми результатами. Наиболее типичным примером в такой ситуации является команда `test`, которую мы обсудим позднее.

Ее название настолько распространено для временной версии программы, что вызовы "не тех" команд `test` происходят раздражающе часто (Позднее будет показано, как обойти эту трудность в командных файлах, где обычно используется команда `test`.). Здесь весьма полезным средством явилась бы команда, которая помогла бы выяснить, какая версия программы должна выполняться.

Один из вариантов решения — цикл поиска по каталогам, указанным в `PATH`, выполняемого файла с данным именем. В гл. 3 мы использовали цикл `for` по именам файлов и аргументам. Здесь же нужен такой цикл:

```
for i in компонента в PATH
do
    если заданное имя в каталоге i
        печать полного путевого имени
done
```

Поскольку любую команду можно запустить внутри символов слабого ударения ‘...‘, очевидное решение состоит в том, чтобы запустить `sed` по значению `PATH`, заменив двоеточия на пробелы. Мы можем проверить это с помощью нашего старого друга `echo`:

```
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin                    4 компонента
$ echo $PATH | sed 's/:/ /a'
/usr/you/bin /bin /usr/bin                     Только три выдано!
$ echo ‘echo $PATH | sed 's/:/ /g'‘
/usr/you/bin /bin /usr/bin                     По-прежнему только три
$
```

Однако такое решение проблематично. Пустая строка в `PATH` служит синонимом '.'. Поэтому перевод двоеточий в пробелы не слишком удачен — теряется информация о пустых компонентах. Для создания правильного списка каталогов пустую компоненту `PATH` нужно перевести в точку. Пустая компонента может быть в середине, начале или конце строки, так что вам придется потрудиться, чтобы учесть все возможные случаи:

```
$ echo $PATH | sed 's/^:/./
>                   s/::/:.:/g
>                   s/:$/:./
>                   s/:/ /g'
. /usr/you/bin /bin /usr/bin
$
```

Мы могли бы записать это с помощью четырех отдельных команд `sed`, но так как редактор `sed` производит замены по порядку, можно выполнить все операции за один вызов.

После задания каталогов в компонентах `PATH` упомянутая выше команда `test(1)` может вывести сообщение о том, существует ли файл в каждом каталоге. В принципе команда `test` — одна из самых "неуклюжих" программ UNIX. Например, команда "`test -r файл`" проверяет, существует ли файл и можно ли его читать; "`test -w файл`" проверяет, существует ли файл и можно ли в него писать, но в седьмой версии нет команды `test -x` (хотя в System V и других версиях есть), а именно она нам и нужна. Мы примем,

что обращение "`test -f файл`" будет проверять, существует ли файл и не является ли он каталогом, т. е. представляет ли он собой обычный файл. Но вам следует обратиться к соответствующей странице справочного руководства, поскольку имеет хождение несколько версий.

Каждая команда вырабатывает код завершения — значение, передаваемое интерпретатору и показывающее, что произошло. Это небольшое целое число, которое устанавливается по соглашению. Так, нуль может означать "истину" (команда успешно завершена), а ненулевое значение трактуется как "ложь" (выполнение команды было неудачным). Обратите внимание на то, что выбранные здесь значения противоположны значениям истины и лжи в языке Си.

Поскольку ложь может представлять множество различных значений, причина неудачи обозначается кодом завершения по лжи. Например, команда `grep` возвращает 0, если произошло сопоставление, 1 — если сопоставления не было, и 2 — в случае ошибки в шаблоне или именах файлов. Каждая программа возвращает код завершения, хотя обычно нас не интересует его значение. Команда `test` неординарна: ее единственное назначение состоит в передаче кода завершения. Она ничего не выводит и не изменяет файлы.

Интерпретатор хранит код завершения последней программы в переменной `$?`:

```
$ cmp /usr/you/.profile /usr/you/.profile
$                                          Выдачи нет, файлы совпадают
$ echo $?
0                                          0 означает успех, файлы идентичны
$ cmp /usr/you/.profile /usr/mary/.profile
/usr/you/.profile /usr/mary/.profile differ: char 6, line 3
$ echo $?
1                                          Не нуль означает, что файлы различны
$
```

У некоторых команд, таких, как `cmp` и `grep`, есть флаг `-s`, который заставляет их завершить выполнение с определенным кодом, но подавляет вывод. Оператор `if` языка `shell` запускает команды в зависимости от кода завершения некоторой команды, а именно:

```
if команда then
    команды, если условие верно
else
    команды, если условие ложно
fi
```

Местоположение символов перевода строк очень важно: `fi`, `then` и `else` распознаются только после символа перевода строки или точки с запятой.

Оператор `if` всегда запускает команду (условие), тогда как в операторе `case` сопоставление с шаблоном производится самим интерпретатором. В некоторых версиях UNIX, включая System V, `test` является встроенной командой интерпретатора, поэтому `if` и `test` будут выполняться так же быстро, как и `case`. Если `test` — не встроенная команда, то операторы `case` более эффективны, чем операторы `if`, и следует использовать именно их для поиска шаблонов;

```
$ case "$1" in
    hello) command
esac
```

выполняется быстрее, чем

```
if test "$1"==hello                  Медленнее, если test не встроенная
then
    command
fi
```

Это одна из причин, по которой в языке shell иногда для проверки условий применяются операторы case, хотя в большинстве языков программирования использовались бы операторы if. С другой стороны, с помощью оператора case непросто определить, имеется ли право доступа к файлу на чтение; здесь предпочтение следует отдать команде test и оператору if.

Итак, теперь мы готовы воспользоваться первой версией команды which, которая выведет сообщение о том, какой файл соответствует команде:

```
$ cat which
# which cmd:  which cmd in PATH is executed, version 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        # use test -x if you can
        then
                echo $i/$1
                exit 0          # found it
        fi
done
exit 1          # not found
$
```

Проверим ее:

```
$ cx which                                  Сделаем ее выполняемой
$ which which
./which
$ which ed
/bin/ed
$ mv which /usr/you/bin
$ which which
/usr/you/bin/which
$
```

Первый оператор case осуществляет контроль ошибки. Обратите внимание на переключение 1>&2 в команде echo, которое выполняется для того, чтобы сообщение об ошибке не пропало в программном канале. Встроенная команда интерпретатора exit может использоваться для передачи кода завершения. В нашем примере exit 2 передает код завершения в ситуации, когда команда не выполняется, exit 1 — в ситуации, когда файл не удалось найти, и exit 0 — в ситуации, когда файл найден. Если нет явного оператора exit, кодом завершения командного файла является код завершения последней выполняемой команды.

Что произойдет, если в вашем текущем каталоге есть программа под именем test? (Мы предполагаем, что test не является встроенной командой.)

```
$ echo 'echo hello' >test              Сделаем поддельную команду test
$ cx test                              Сделаем ее выполняемой
$ which which                          Попробуем which теперь
hello                                  Неудача!
./which
$
```

Вывод: требуется больший контроль. Можно запустить команду which (если нет команды test в текущем каталоге), чтобы определить полное имя для test и задать его явно. Но это не лучшее решение, поскольку команда test может присутствовать в различных каталогах в разных версиях системы, а комана which зависит от sed и echo, так что необходимо указать и их полные имена. Можно поступить проще — установить значение PATH в командном файле так, чтобы поиск команд осуществлялся только в /bin и /usr/bin. Конечно, это возможно только в команде which, причем прежнее значение PATH следует сохранить для определения последовательности каталогов при поиске.

```
$ cat which
# which cmd:  which cmd in PATH is executed, final version

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        # this is /bin/test
        then                    # or /usr/bin/test only
                echo $i/$1
                exit 0          # found it
        fi
done
exit 1          # not found
$
```

Теперь команда which выполняется даже в том случае, если есть "поддельная" команда test (sed или echo) среди каталогов, входящих в PATH.

```
$ 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 языке shell имеются две операции, объединяющие команды || и &&, использование которых часто более компактно и удобно, чем оператора if. Например, операция || может заменить некоторые операторы if:

```
test -f имя_файла || echo имя_файла не существует
```

эквивалентно

```
if test ! -f имя_файла                    ! обращает условие
then
    echo имя файла не существует
fi
```

Операция ||, несмотря на свой вид, не имеет ничего общего с конвейерами — это обычная операция, означающая ИЛИ. Выполняется команда слева от ||. Если ее код завершения 0 (успех), справа от || команда игнорируется. Если команда слева возвращает другое значение (неудача), выполняется команда справа, и значение всего выражения есть код завершения правой команды. Иными словами, || представляет собой обычную операцию ИЛИ, которая не выполняет свою правую часть, если левая часть завершилась успешно. Соответственно && есть обычная операция И, выполняющая свою правую часть, только если левая часть завершилась успешно.

■ Упражнение: Почему в команде which перед выходом из нее не восстанавливается значение PATH из opath?

■ Упражнение: Если в языке shell используется esac для завершения оператора case и fi для завершения оператора if, почему для завершения оператора do применяется done?

■ Упражнение: Введите в команду which флаг -a, чтобы выводились все файлы из PATH, а не только первый найденный. Подсказка: match='exit 0'

■ Упражнение: Модифицируйте команду `which` так, чтобы она учитывала встроенные в язык `shell` команды типа `exit`.

■ Упражнение: Модифицируйте команду `which` так, чтобы она проверяла права доступа файлов. Как изменить ее для получения диагностического сообщения, если файл не удалось найти?

5.3 Циклы `while` и `until`: контроль входа в систему

В гл. 3 цикл `for` использовался для нескольких итеративных программ. Обычно цикл `for` охватывает множество имен файлов, как в '`for i in *.c`', или все аргументы командного файла, как в '`for i in $*`'. Но циклы в языке `shell` могут быть более общими, чем в этих идиомах, например цикл `for` в команде `which`.

Имеются три вида циклов: `for`, `while` и `until`. Чаще всего используется цикл `for`. В нем выполняется последовательность команд (тело цикла) для каждого элемента из множества слов. В большинстве случаев множество образуют просто имена файлов. В циклах `while` и `until` контроль над выполнением тела цикла осуществляется с помощью кода завершения команды. Тело цикла выполняется до тех пор, пока команда условия не вернет ненулевой код для `while` или нуль для `until`. Циклы `while` и `until` идентичны, за исключением кода завершения команды.

Ниже приведены основные формы каждого цикла:

```
for i in список слов
do
    тело цикла, $i последовательно получает значения элементов
done

for i (явно перечисляются аргументы командного файла, т.е. $*)
do
    тело цикла, $i последовательно получает значения аргументов
done

while команда
do
    тело цикла выполняется, пока команда возвращает истина
done

until команда
do
    тело цикла выполняется, пока команда возвращает ложь
done
```

Вторая форма цикла `for`, в которой пустой список обозначается как `$*`, является удобным сокращением записи для наиболее типичного использования.

Командой условия, управляющей циклами `while` или `until`, может быть любая команда. Очевидным примером служит цикл `while`, в котором осуществляется контроль входа (пусть Мэри) в систему:

```
while sleep 60
do
    who | grep mary
done
```

Команда `sleep`, устанавливающая паузу на 60 с, всегда выполняется нормально (если ее не прервали) и, значит, всегда возвращает код "успех", поэтому в цикле раз в минуту будет проверяться, находится ли Мэри в системе. Недостаток такого решения состоит в том, что если Мэри уже вошла в систему, то нужно ждать 60 с, чтобы узнать об этом. О продолжении же работы Мэри в системе каждую минуту будет поступать сообщение. Цикл можно перевернуть и записать с помощью `until`, чтобы получать информацию сразу без задержки, если Мэри в данный момент работает в системе:

```
until who | grep mary
do
    sleep 60
done
```

Теперь условие представляется более интересным. Если Мэри вошла в систему, то '`who | grep mary`' выдаст запись о ней из списка команды `who` и вернет код "истина". Это связано с тем, что `grep` выдает код завершения, показывающий, удалось ли ей найти что-нибудь, а код завершения конвейера есть код завершения последней команды.

В заключение мы можем оформить команду, дав ей имя и установив в системе:

```
$ cat watchfor
# watchfor:  watch for someone to log in

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          Установим в системе
$
```

Мы заменили `grep` на `egrep`, чтобы было можно задавать

```
$ watchfor 'joe | mary' и следить за несколькими пользователями.
```

Более сложный пример: можно контролировать вход в систему и выход из нее всех пользователей и сообщать обо всех фактах входа или выхода. Это можно рассматривать

как некоторое дополнение к команде who. Основная идея проста: раз в минуту запускать команду who и сравнивать результат ее действия с результатом, полученным минутой ранее, сообщая обо всех различиях. Вывод команды who хранится в файле, и мы можем записывать его в каталог /tmp. Чтобы отличить свои файлы от файлов, принадлежащих другим процессам, в имена файлов вставляется переменная интерпретатора $$ (номер процесса команды интерпретатора), что является обычной практикой. Имя команды упоминается во временных файлах главным образом для администратора системы. Часто команды (включая данную версию watchfor) оставляют после себя файлы в /tmp, и полезно знать, какая команда это сделала. Здесь ":" — встроенная команда, которая

```
$ cat watchwho
# watchwho:  watch who logs in and out

PATH=/bin:/usr/bin
new=/tmp/wwho1.$$
old=/tmp/wwho2.$$
>$old           # create an empty file

while :         # loop forever
do
        who >$new
        diff $old $new
        mv $new $old
        sleep 60
done | awk '/>/ { $1 = "in:     "; print }
            /</ { $1 = "out:    "; print }'
$
```

только обрабатывает свои аргументы и возвращает код "истина". Мы могли бы заменить ее командой true, просто передающей код завершения "истина" (есть также команда false), но команда ':' более эффективна, поскольку не нужно выполнять эту команду, выбирая ее из файловой системы.

В выводе команды diff используются символы < и > для разделения данных из двух файлов. Программа, написанная на языке awk, обрабатывает результаты, чтобы сообщить об изменениях в более понятном формате. Обратите внимание на то, что весь цикл передает результаты работы по конвейеру awk–программе, вместо того, чтобы запускать заново awk–программу каждую минуту. Для такой обработки редактор sed не подходит, поскольку его вывод всегда задерживается по сравнению с входным потоком на одну строку: всегда есть одна входная строка, которая уже обработана, но не напечатана, а это приводит к ненужной задержке.

Поскольку файл old создается пустым, первый вывод команды watchfor содержит весь список пользователей, находящихся в системе в данный момент. Замена команды, которая создает файл old, на who > $old приведет к тому, что watchfor выдаст только изменения, но это уже — дело вкуса.

Другая программа в цикле следит за содержимым вашего почтового ящика: как только оно изменяется, программа выдает сообщение: "You have a mail" ("У вас есть почта"). Такая программа является полезной альтернативой встроенному в интерпретатор механизму, использующему переменную MAIL. Чтобы показать другой стиль программирования, мы реализовали ее с помощью переменных интерпретатора, а не файлов:

```
$ cat checkmail
# checkmail:  watch mailbox for growth

PATH=/bin:/usr/bin
MAIL=/usr/spool/mail/‘getname‘  # system dependent

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 "You have mail" }'
$
```

Мы опять воспользовались awk–программой, на этот раз — чтобы добиться вывода сообщения только в тех случаях, когда почтовый ящик пополняется, а не просто изменяется. Иначе вы получите сообщение сразу после исключения письма. (Версия, встроенная в интерпретатор, имеет такой недостаток.)

Обычно интервал времени устанавливается равным 60 с, но если командная строка содержит параметр, например

```
$ chekmail 30
```

то интервал задается им. Переменная интерпретатора принимает в качестве значения заданное параметрами время или 60 с, если время не задано, с помощью присваивания

```
t=${1-60}
```

Это еще одна возможность языка shell. ${var} эквивалентно $var и может использоваться для преодоления трудностей, связанных с появлением переменных внутри буквенно–цифровых строк:

```
$ var=hello
$ varx=goodbye
$ echo $var
hello
$ echo ${var}x
hellox
$
```

Определенные символы внутри фигурных скобок задают специальную обработку переменной. В том случае, когда переменная не определена и за ее именем идет знак вопроса, выдается строка, следующая за символом ?, и интерпретатор прекращает работу (если только он не работает в диалоговом режиме). При отсутствии строки печатается стандартное сообщение:

```
$ echo ${var?}
hello                              все в порядке, var определено
$ echo ${junk)
junk: parameter not set            стандартное сообщение
$ echo ${junk?error!)
junk: error!                       строка задана
$
```

Отметим, что в сообщении, выдаваемом интерпретатором, всегда указывается имя неопределенной переменной.

В другой конструкции `${var-thing}` выбирается `$var`, если оно определено, и `thing` — в противном случае. В подобной конструкции `${var-thing}` значение `$var` также устанавливается равным `thing`:

```
$ echo ${junk-'Hi there))
Hi there
$ echo ${junk?)
junk: parameter not set                  значение junk не изменилось
$ echo {junk='Hi there')
Hi there
$ echo ${junk?}
Hi there                                 junk принял значение Hi there
$
```

Правила получения значений переменных приведены в табл. 5.3. Возвращаясь к нашему исходному примеру

```
t=${1-60}
```

видим, что `t` присваивается `$1` или 60, если аргумент не задан.

`$var`	Значение `var`; ничего, если `var` не определено
`${var}`	То же; полезно, если за именем переменной следует буквенно-цифровая строка
`${var-thing}`	Значение `var`, если оно определено; в противном случае — `thing`; `$var` не изменяется
`${var=thing}`	Значение `var`, если оно определено; в противном случае — `thing` Если `var` не определено, то `$var` присваивается `thing`
`${var?строка}`	Если `var` определено — `$var`; в противном случае выводится строка и интерпретатор прекращает работу. При пустой строке выводится: `var: parameter not set`
`${var+thing}`	`thing`, если `$var` определено; в противном случае — ничего

Таблица 5.3: Получение значений переменных в языке

■ Упражнение: Обратите внимание на реализацию команд `true` и `false` в `/usr/bin` или `/bin`. (Как бы вы определили, где они находятся?)

■ Упражнение: Измените команду `watchfor` так, чтобы пользователь мог задавать несколько имен, а не вводить '`joe | mary`'.

■ Упражнение: Напишите версию команды `watchwho`, которая использует команду `comm` вместо `awk` для сравнения новой и старой информации. Какая версия вам больше нравится?

■ Упражнение: Напишите версию команды `watchwho`, в которой вывод команды `who` хранится в переменных языка `shell`, а не в файлах. Какая версия лучше? Какая версия быстрее работает? Следует ли в командах `watchwho` и `checkmail` автоматически использовать операцию `&`?

■ Упражнение: В чем состоит различие между пустой командой языка `shell` `:` и символом примечания `#`? Нужны ли они?

5.4 Команда `trap`: обработка прерываний

Если во время выполнения команды `watchwho` нажать клавишу *DEL* (*УДЛ*) или отключить компьютер от сети, то один или несколько временных файлов останутся в каталоге `/tmp`. Команда `watchwho` удаляет временные файлы перед окончанием своей работы. Необходимы средства обнаружения таких ситуаций и восстановления после прерывания.

При нажатии клавиши *DEL* всем процессам, запущенным с этого терминала, посылается сигнал прерывания. Аналогично в случае отключения посылается сигнал отбоя. Существуют и другие сигналы. Если в программе не предусмотрены специальные действия по обработке сигналов, то указанные сигналы прекращают ее выполнение. Интерпретатор защищает программы, запущенные с помощью `&`, от прерываний, но не от отключений.

В гл. 7 сигналы рассматриваются подробнее, но для работы с ними на языке `shell` глубоких знаний не требуется. Встроенная команда интерпретатора `trap` устанавливает последовательность команд, которая должна выполняться при возникновении сигнала:

```
trap последовательность_команд список_номеров_сигналов
```

Последовательность команд — единый аргумент, поэтому его почти всегда нужно брать в кавычки. Номера сигналов обозначаются небольшими целыми числами, например, 2 соответствует сигналу, возникающему при нажатии клавиши *DEL*, а 1 — сигналу, возникающему при отключении от сети. Номера сигналов, наиболее часто используемых в `shell`–программах, приведены в табл. 5.4.

Для удаления временных файлов в команде `watchwho` вызов команды `trap` должен указываться перед циклом, чтобы перехватить сигналы прерывания, отбоя и окончания выполнения:

```
...
trap 'rm -f $new $old; exit 1' 1 2 15
while:
...
```

0	Выход из интерпретатора (по любой причине, включая конец файла)
1	Отбой
2	Прерывание (клавиша DEL)
3	Останов (*ctl-*; вызывает распечатку содержимого памяти программы)
9	Уничтожение (нельзя перехватить или игнорировать)
15	Окончание выполнения; сигнал по умолчанию, производимый `kill(1)`

Таблица 5.4: Номера сигналов в интерпретаторе

Последовательность команд, образующих первый аргумент команды `trap`, подобна вызову подпрограммы, который происходит сразу по возникновении сигнала. Когда эта последовательность окончится, прерванная программа возобновляется с места прерывания, если только сигнал не уничтожит ее. Таким образом, последовательность команд в `trap` должна явно вызывать `exit`, иначе `shell`–программа продолжит свое выполнение после прерывания. Кроме того, последовательность команд будет читаться дважды: при установке команды `trap` и при обращении к ней. Поэтому последовательность команд лучше защищать апострофами, чтобы значения переменных вычислялись только при выполнении программ, указанных в команде `trap`. В данном случае это не имеет значения, но позднее вы столкнетесь с ситуацией, когда это важно. Кстати, флаг `-f` предписывает команде `rm` не задавать вопросов.

Иногда команду `trap` полезно применять в диалоговом режиме, чаще всего для того, чтобы не допустить уничтожения программы сигналом отбоя, возникшим при обрыве телефонной связи:

```
$ (trap "" 1; долго_выполняемая команда) &
2134
$
```

Для процесса и его потомков пустая последовательность означает, что нужно игнорировать прерывания. При наличии скобок команда `trap` и `долго_выполняемая_команда` выполняются порожденным интерпретатором вместе и как фоновые; без них команда `trap` действовала бы на исходный интерпретатор, так же как и на `долго_выполняемую_команду`.

Команда `nohup(1)` — небольшая `shell`–программа, обеспечивающая непрерывное выполнение команд. Ниже полностью приведен ее вариант из 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, направлен ли стандартный выходной поток на терминал, чтобы вы могли решить, следует ли его сохранять. Фоновая программа выполняется с

помощью команды nice, что снижает ее приоритет по сравнению с диалоговыми программами. (Обратите внимание, что команда nohup не устанавливает значение PATH. А может быть, это нужно?)

Команда exec использована только для повышения эффективности; команда nice может выполняться и без нее. Exec — встроенная команда интерпретаторов, которая заменяет процесс, играющий роль текущего интерпретатора, на указанную программу. Таким образом она избавляется от одного процесса, а именно от интерпретатора, обычно ожидающего завершения программы. Мы могли бы применять exec и в некоторых других программах, например в конце обобщенной программы cal, когда происходит обращение к /usr/bin/cal.

Кстати, сигнал 9 — это тот сигнал, который нельзя перехватить или игнорировать: он всегда уничтожает процесс. На языке shell его посылка задается с помощью

```
$ kill -9 номер_процесса
```

Обращение kill -9 не является стандартным, поскольку процессу, уничтоженному таким способом, не дается время для приведения в порядок своих дел перед "смертью".

■ Упражнение: В приведенной выше версии команды nohup стандартный поток диагностики команды соединяется со стандартным выходным потоком. Хорошее ли это решение? Если нет, то как бы вы разделили их явно?

■ Упражнение: Найдите встроенную команду times и добавьте к вашему файлу строку .profile, чтобы при вашем выходе из системы интерпретатор выдавал использованное вами процессорное время.

■ Упражнение: Напишите программу, находящую следующий свободный индентификатор пользователя в файле /etc/passwd. Если у вас есть энтузиазм (и право доступа), сделайте из нее команду, устанавливающую нового пользователя системы. Какие нужны для нее права доступа? Как следует ей обращаться с прерываниями?

5.5 Команда overwrite: замена файла

В команде sort есть флаг -o для замены файла:

```
$ sort файл1 -o файл2
```

Ее эквивалент:

```
$ sort файл1 > файл2
```

Если файл1 и файл2 — это один и тот же файл, то после операции переключения > входной файл станет пустым перед сортировкой. Но с флагом -o команда выполняется правильно, потому что входной файл сортируется и сохраняется во временном файле перед созданием выходного файла.

Могут использовать флаг -o и другие команды. Например, редактор sed может редактировать файл с заменой:

```
$ sed 's/UNIX/UNIX (TM)/g' -o ch2                Так не получится!
```

Непрактично изменять все подобные команды, вводя флаг — это не лучшее решение. Более целесообразным представляется централизованное выполнение функций, как в случае операции > интерпретатора, для чего мы создадим программу `overwrite`. Первый ее вариант выглядит так:

```
$ sed 's/UNIX/UNIX (TM)/g' гл2 | overwrite гл2
```

В основном алгоритм программы очевиден: нужно только сохранить где-нибудь весь входной поток вплоть до конца файла, а затем копировать его в файл, указанный как аргумент:

```
# overwrite:  copy standard input to output after EOF
# version 1.  BUG here

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               # collect the input
cp $new $1              # overwrite the input file
rm -f $new
```

Команда `cp` используется вместо команды `mv`, чтобы не изменились права доступа и остался прежним владелец выходного файла, если он уже существует. Хотя этот вариант и чрезвычайно прост, здесь возможна "фатальная" ошибка: если пользователь нажмет клавишу *DEL* (*УДЛ*) во время выполнения команды `cp`, первоначальный выходной файл будет уничтожен. Необходимо соблюдать осторожность, поскольку прерывание может остановить замену входного файла:

```
# overwrite:  copy standard input to output after EOF
# version 2.  BUG here too

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               # collect the input
cp $1 $old              # save original file

trap '' 1 2 15          # we are committed; ignore signals
cp $new $1              # overwrite the input file

rm -f $new $old
```

Если клавиша *DEL* будет нажата прежде, чем начнется работа с исходным файлом, то произойдет удаление временных файлов и файл останется один. После сохранения файла сигналы игнорируются, поэтому выполнение последней команды `cp` не прервется. Если команда `cp` начала выполняться, команда `overwrite` обязана заменить исходный файл.

Здесь есть некоторая тонкость. Рассмотрим последовательность:

```
$ sed 's/UNIX/UNIX (TM)g' special | overwrite special
command garbled: s/UNtX(TM)g
$ ls -l special
-rw-rw-rw- 1 you 0 Oct 1 09:02 special                    #$%@*!
$
```

Если в программе, поставляющей входной поток для команды `overwrite`, произойдет ошибка, то выходной поток будет пустым, и `overwrite` обязательно (с сознанием выполненного долга) уничтожит файл, заданный в качестве аргумента.

Во избежание такого финала можно предложить несколько решений. Команда `overwrite` могла бы запрашивать подтверждение перед заменой файла, но, сделав команду диалоговой, мы потеряем большую часть ее достоинств. Она могла бы проверять, что ее входной поток не пуст (с помощью `test -2`), но это некрасиво и к тому же неверно: выходной поток мог быть создан до обнаружения ошибки.

Наилучшее решение заключается в том, чтобы выполнять программу, поставляющую данные, под контролем команды `overwrite`, чтобы можно было проверить ее код завершения. Это, правда, противоречит традициям и здравому смыслу: ведь в конвейере команда `overwrite` обычно должна быть последней, но для правильной работы она должна идти первой. Однако `overwrite` ничего не выдает в стандартный выходной поток, поэтому можно считать, что не происходит потери общности. Более того, ее синтаксис не является каким-то необычным: `time`, `nice`, `nohup` представляют собой команды, аргументами которых служат другие команды. Ниже приведен безопасный вариант:

```
# overwrite:  copy standard input to output after EOF
# final version

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   # clean up files

if PATH=$opath "$@" >$new               # collect input
then
        cp $file $old   # save original file
        trap '' 1 2 15  # we are committed; ignore signals
        cp $new $file
else
        echo "overwrite: $1 failed, $file unchanged" 1>&2
        exit 1
fi
rm -f $new $old
```

Встроенная команда интерпретатора `shift` сдвигает весь список аргументов на одну позицию влево: `$2` становится `$1`, `$3` становится `$2` и т. д. Строка `"$@"` обозначает все аргументы (после `shift`), как и `$*`, но без интерпретации; мы вернемся к ее рассмотрению в разд. 5.7.

Заметьте, что значение `PATH` нужно восстановить перед выполнением команды пользователя; если этого не сделать, то команды, не находящиеся в `/bin` или `/usr/bin`, будут недоступны для `overwrite`.

Теперь команда `overwrite` выполняется верно (хотя и она получилась несколько громоздкой):

```
$ 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                Не изменился
$ overwrite notice sed 's/UNIX/UNIX(TM)/g' notice
$ cat notice
UNIX(TM) is a Trademark of Bell Laboratories
$
```

Типичной задачей является использование редактора `sed` для замены всех вхождений одного слова на другое слово. Имея под рукой команду `overwrite`, легко написать программу на языке `shell` для ее решения:

```
$ cat replace
# replace:  replace str1 in files with str2, in place

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
$
```

(Вспомните: если список в цикле `for` пуст, то по умолчанию он равен `$*`.) Мы использовали @ вместо / для разбиения в команде подстановки, поскольку менее вероятно, что @ вступит в конфликт с входной строкой. Команда `replace` устанавливает `PATH` равным `/bin:/usr/bin`, исключая `$HOME/bin`. Это означает, что overwrite должна находиться в `/usr/bin`, чтобы команда `replace` сработала. Мы сделали такое предположение для простоты; если вы не можете поместить `overwrite` в `/usr/bin`, вам придется добавить `$HOME/bin` к `PATH` в команде `replace` или явно задать полное имя `overwrite`. В дальнейшем будем полагать, что команды, которые мы создаем, находятся в `/usr/bin`, где им и следует быть.

■ Упражнение: Почему команда `overwrite` не использует сигнал 0 в команде `trap`, чтобы файлы удалялись при выходе из нее? Подсказка: попробуйте нажать клавишу *DEL* во время выполнения следующей программы:

```
trap "echo exiting; exit 1" 0 2 sleep 10
```

■ Упражнение: Добавьте флаг `-v` к команде `replace` для вывода всех измененных строк на `/dev/tty`. Подсказка: `s/$left/$right/g $vflag`.

■ Упражнение: Увеличьте надежность команды `replace`, чтобы ее выполнение не зависело от символов в строке замены.

■ Упражнение: Можно ли использовать `replace` для замены `i` на `index` всюду в программе? Какие вы внесли бы изменения, чтобы добиться этого?

■ Упражнение: Достаточно ли команда `replace` эффективна и удобна, чтобы находиться в каталоге `/usr/bin`? Не лучше ли вводить по мере необходимости подходящие команды редактора `sed` (да или нет) ? Обоснуйте свой ответ.

■ Упражнение: (*Усложненное.*) Команда

```
$ overwrite файл 'who | sort '
```

не выполняется. Объясните причину этого и исправьте ее. Подсказка: посмотрите `eval` в справочном руководстве по `sh(1)`. Как ваше решение повлияет на интерпретацию специальных символов в команде?

5.6 Команда `zap`: уничтожение процесса по имени

Команда `kill` только завершает процесс с указанным номером. Если нужно уничтожить определенный фоновый процесс, обычно приходится выполнить команду `ps`, чтобы узнать номер процесса, а затем ввести этот номер в качестве аргумента для команды `kill`. Однако нелепо иметь программу, выдающую номер процесса, который сразу же передается вручную другой программе. Имеет смысл написать программу, скажем `zap`, для автоматического выполнения такой работы. Здесь, правда, есть одно препятствие: уничтожение процессов опасно, поэтому следует принять меры для обеспечения сохранности нужных процессов. Хорошей защитой всегда служат диалоговое выполнение `zap` и использование команды `pick` для выбора "жертв".

Кратко напомним вам о команде `pick`: она выдает поочередно свои аргументы, спрашивая ответ у пользователя; если ответ — y, то аргумент выводится (команда `pick` обсуждается в следующем разделе). В нашем случае `pick` используется для подтверждения, что процессы, выбранные по имени, — именно те, которые пользователь хочет уничтожить:

```
$ cat zap
# zap pattern:  kill all processes matching pattern
# BUG in this version

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 знаки слабого ударения, защищенные символами обратной дробной черты, `awk`–программа выделяет номер процесса из выходных данных команды `ps`, выбранной с помощью `pick`:

```
$ sleep 1000 &
2216
$ ps -ag
PID TTY TIME CMD

...
```

```
2216 0 0:00 sleep 1000

...

$ zap sleep
2216?
0? q                                Что происходит?
$
```

Проблема состоит в том, что выходные данные команды `ps` разбиты на слова, которые воспринимаются и обрабатываются командой `pick` как отдельные аргументы вместо того, чтобы обрабатываться сразу по строке. Обычная процедура интерпретатора заключается в разбиении строк на аргументы с границами пробел/не пробел, как показано ниже:

```
for i in 1 2 3 4 5
```

В этой программе нужно контролировать процесс разбиения интерпретатором строк на аргументы, чтобы только символ перевода строки разделял соседние "слова".

Внутренняя переменная интерпретатора `IFS` (internal field separator - внутренний разделитель полей) представляет собой строку символов, которая разделяет слова в списке аргументов, находящихся в знаках слабого ударения или циклах `for`. Обычно `IFS` содержит пробелы, символы табуляции и конца строки, но мы можем заменить ее на что-либо нужное, например просто на символ перевода строки:

```
$ echo 'echo $#' >nargs
$ cx nargs
$ who
you tty0 Oct 1 05:59
pjw tty2 Oct 1 11:26
$ nargs 'who'
10                          10 полей, разделенных пробелом и концом строки
$ IFS='
'                           Только конец строки
$ nargs `who`
2                           Две строки, два поля
$
```

После установки `IFS` равным символу перевода строки команда `zap` выполняется отлично:

```
$ cat zap
# zap pat:  kill all processes matching pat
# final version

PATH=/bin:/usr/bin
IFS='
'                       # just a newline
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
2216 0 0:00 sleep 1000
$ zap sleep
PID TTY TIME CMD
2216 0 0:00 sleep 1000? y
2314 0 0:02 egrep sleep? n
$
```

Мы здесь кое-что добавили: необязательный аргумент, обозначающий сигнал (обратите внимание на то, что SIG будет неопределенным, а значит, должен рассматриваться как пустая строка, если аргумент не задан), а также `egrep` вместо `grep`, чтобы разрешить более сложные шаблоны типа '`sleep | date`'. Первая команда `echo` выдает столбец из заголовков выходных данных команды `ps`.

Вас может заинтересовать, почему эта команда называется `zap`, а не просто `kill`. Основная причина заключается в том, что в отличие от случая с командой `cal` мы не даем действительно новой команды `kill`: `zap` по необходимости является диалоговой командой, с одной стороны, а с другой — мы хотим сохранить имя `kill` для настоящей команды. К тому же `zap` чрезвычайно медленна из-за накладных расходов на все дополнительные программы, хотя самую длинную по времени реализации команду `ps` все равно нужно выполнять. В следующей главе будет продемонстрировано более эффективное решение.

■ Упражнение: Измените команду `zap` так, чтобы она, выдавая заголовки из команды `ps`, была не чувствительна к изменениям в формате вывода `ps`. Насколько это усложнит программу?

5.7 Команда `pick`: пробелы или аргументы

Вы уже достаточно подготовлены для того, чтобы написать команду `pick` на языке `shell`. Единственным новым средством является механизм чтения входного потока пользователя. Встроенная команда интерпретатора `read` читает одну строку текста из стандартного входного потока и присваивает ее (без перевода строки) в качестве значения указанной переменной:

```
$ read greeting
hello, world                          Вводим новое значение для приветствия
$ echo $greeting
hello, world
$
```

Самым типичным примером использования команды read в файле .profile служит установка значений переменных среды при входе в систему, прежде всего установка переменных интерпретатора типа TERM.

Команда read может читать только из стандартного входного потока; его нельзя даже переключить. Ни одну из встроенных команд интерпретатора (в отличие от основных структур управления типа for) нельзя переключить с помощью операций > или <:

```
$ read greeting </etc/passwd
goodbye                              Тем не менее надо ввести значение
illegal io                           Сейчас shell сообщает об ошибке
$ echo $greeting                     greeting получает введенное значение,
goodbye                              а не значение из файла
$
```

Это можно считать ошибкой интерпретатора, но такова жизнь. К счастью, можно предусмотреть переключение в цикле, охватывающем команду read, что является основным принципом реализации команды pick:

```
# pick:  select arguments

PATH=/bin:/usr/bin

for i                               # for each argument
do
        echo -n "$i? " >/dev/tty
        read response
        case $response in
        y*)     echo $i ;;
        q*)     break
        esac
done </dev/tty
```

Обращение echo -n подавляет заключительный символ перевода строки, так что переменную response можно вывести на той же строке, что и приглашение. Конечно, приглашения выдаются на устройство /dev/tty, поскольку стандартный выходной поток, по всей вероятности, не выводится на терминал.

Оператор break заимствован из языка Си: он завершает выполнение самого внутреннего цикла, в нашем случае for, когда вводится q. Мы выбрали символ q как сигнал прекращения процесса выбора потому, что это легко сделать, потенциально удобно и не противоречит другим программам.

Интересно поэкспериментировать с пробелами в аргументах для команды pick:

```
$ pick '1 2' 3
1 2?
3?
$
```

Если вы хотите узнать, как команда pick читает свои аргументы, запустите ее и нажмите клавишу *RETURN* после каждого приглашения. В том виде, в каком написана эта команда, она выполняется отлично: в цикле for i аргументы обрабатываются правильно. Мы могли бы написать цикл другими способами:

```
$ grep for pick                    Выясните, что делает эта версия
for i in $*
$ pick '1 2' 3
1?
2?
3?
$
```

Эта версия не работаете поскольку операнды в цикле снова распознаются, а наличие пробелов в первом аргументе приводит к тому, что он разбивается на два аргумента. Попробуйте взять в кавычки `$*`:

```
$ grep for pick                    Попробуем другую версию
for i in "$*"
$ pick '1 2' 3
123?
$
```

Такая версия тоже не работает, поскольку `"$*"` является единым словом, которое образовано из всех аргументов, объединенных вместе с разделяющими пробелами. Но решение все-таки есть (это почти черная магия): строка `"$@"` трактуется особым образом интерпретатором и преобразуется в нужное число аргументов для командного файла:

```
$ grep for pick                    Попробуем третью версию
for i in "$@" '
$ pick '1 2' 3
1 2?
3?
$
```

Строка `$@`, не взятая в кавычки, идентична `$*`; она обрабатывается иначе, только если заключена в кавычки. Мы использовали ее в команде `overwrite`, чтобы сохранить аргументы для команды пользователя.

В итоге мы можем сформулировать следующие правила: `$*` и `$*` раскрываются как аргументы и снова распознаются; наличие пробелов в аргументах приводит к разбиению их на несколько аргументов;

- `"$*"` является единым словом, которое образовано из всех аргументов командного файла, объединенных вместе с пробелами;
- `"$@"` идентично аргументам, получаемым командным файлом: пробелы в аргументах игнорируются, в результате получается список слов, идентичных исходным аргументам.

Если команда `pick` не имеет аргументов, она, по-видимому, должна читать стандартный входной поток, поэтому можно задать

```
$ pick < mailinglist
```

вместо

```
$ pick `cat mailinglist`
```

Но мы не будем исследовать эту версию команды `pick` во избежание некоторых неприятных осложнений. Кроме того. значительно проще написать такую же программу на Си. С ней вы познакомитесь в следующей главе.

Первые два из приведенных ниже упражнений достаточно сложны, но полезны даже для опытных программистов, работающих на языке `shell`.

■ УПРАЖНЕНИЕ: Попробуйте написать программу `pick`, которая читает аргументы из стандартного входного потока, если ничего не задано в командной строке. Она должна правильно обрабатывать пробелы. Будет ли допустим ответ `q`? Если нет, то попытайтесь выполнить следующее упражнение.

■ УПРАЖНЕНИЕ: Хотя встроенные команды интерпретатора, такие, как `read` и `set`, нельзя переключить, можно временно переключить сам интерпретатор. Прочтите в справочном руководстве раздел по `sh(1)`, в котором описывается команда `exec`, и придумайте, как читать из `/dev/tty` без вызова порожденного интерпретатора. (Может оказаться полезным сначала прочитать гл. 7.)

■ УПРАЖНЕНИЕ: (*Более простое.*) Используйте команду `read` в вашем файле `.profile` для инициации `TERM`, а также всего, что зависит от нее, например позиции табуляции.

5.8 Команда news: служба информации пользователей

В гл. 1 упоминалось о том, что в вашей системе может быть команда `news` для передачи сообщений, представляющих интерес для всех пользователей системы. Хотя названия команды и ее детали могут различаться, большинство систем имеет службу информации. Мы рассматриваем команду `news` не для замены вашей местной команды, а чтобы показать, как легко написать такую программу на языке `shell`. Неплохо было бы сравнить реализацию предлагаемой здесь команды `news` с вашей версией.

Обычно основная идея таких программ заключается в том, что отдельные фрагменты новостей хранятся по одному в файлах в специальном каталоге типа `/usr/news`. Наша команда `news` сравнивает время изменения файлов в каталоге `/usr/news` и вашем исходном каталоге (`.news_time`). В целях отладки мы можем использовать каталог '.' как для файлов новостей, так и для `news_time`. Можно заменить его на `/usr/news`, когда программа будет готова для общего пользования:

```
$ cat news
# news:  print news files, version 1

HOME=.          # debugging only
cd .            # place holder for /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` заменяет время последней модификации файла, заданного в качестве аргумента, на настоящее время, не подвергая сам файл модификации. Для отладки мы даем только эхо имен файлов новостей, а не печатаем их. Цикл завершается при обнаружении `news_time`, тем самым перечисляются только файлы со свежими новостями. Заметьте, что символ `*` в операторе `case` может быть сопоставлен с `/`, что недопустимо для шаблонов имен файлов. А что будет, если `news_time` не существует?

```
$ rm .news_time
$ news
$
```

Отсутствие ответа удивляет и является ошибочным. Это вызвано тем, что когда команда `ls` не находит файл, она выдает соответствующее сообщение в стандартный выходной поток прежде, чем вывести какую-либо информацию о существующих файлах. Такая ситуация, безусловно, ошибочна — диагностические сообщения должны передаваться в стандартный файл диагностики. Но мы можем обнаружить эту ситуацию в цикле и переключить стандартный файл диагностики на стандартный выходной поток, так что все версии будут работать одинаково. (Данная проблема решена в новой версии, но мы рассмотрели ее, чтобы проиллюстрировать, как легко устранить недоделки.)

```
$ cat news
# news:  print news files, version 2

HOME=.          # debugging only
cd .            # place holder for /usr/news
IFS='
'               # just a newline
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
```

```
$ news
news: news
news: y
news: x
$
```

Мы должны были установить IFS равным символу конца строки, чтобы сообщение

```
./.news_time not found
```

не распознавалось как три слова.

Команда news должна выводить на печать файлы новостей, а не создавать эхо их имен. Полезно знать, кто и когда послал сообщение, поэтому мы воспользуемся командами set и ls -l для вывода заголовка перед самим сообщением:

```
$ ls -l news
-rwxrwxrwx 1 you 208 Oct 1 12:05 news
$ set `ls -l news`
-rwxrwxrwx: bad option(s)                    Что-то неправильно!
$
```

Это один из тех случаев, когда взаимозаменяемость программы и данных на языке shell имеет значение. Команда set "ругается", потому что ее аргумент ("-rwxrwxrwx") начинается с минуса и, следовательно, выглядит как флаг. Очевидным (хотя и неэлегантным) решением было бы предварить аргумент обычным символом:

```
$ set X `ls -l news`
$ echo "news: ($3) $5 $6 $7"
news: (you) Oct 1 12 : 05
$
```

Здесь представлен разумный формат с указанием автора и даты сообщения вместе с именем файла. Приведем окончательный вариант команды news:

```
# news:  print news files, final version

PATH=/bin:/usr/bin
IFS='
'                       # just a newline
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 перевода строк разделяют в заголовке при печати фрагменты новостей. Первым значением `IFS` является символ перевода строки, поэтому сообщение `not found` из вывода первой команды `ls` (если оно есть) рассматривается как один аргумент. Во втором случае переменной `IFS` присваивается пробел, поэтому вывод второй команды `ls` разбивается на несколько аргументов.

■ Упражнение: Добавьте в команду `news` флаг `-n` ("notify" — извещение), чтобы сообщать о новостях, но не печатать их, и не выполняйте `touch .news_time`. Эту команду можно поместить в ваш файл `.profile`.

■ Упражнение: Сравните предложенный здесь подход и реализацию команды `news` с аналогичной командой вашей системы.

5.9 Команды get и put: контроль изменении файла

В последнем разделе этой длинной главы мы приведем большой и более сложный пример, в котором продемонстрируем вам взаимодействие языков `shell`, `awk` и `sed`.

Программа развивается по мере того, как мы устраняем ошибки и добавляем в нее новые средства. Иногда полезно сохранять ее разные версии, особенно в ситуации, когда кто-то переносит программу на другую машину, и возникает вопрос: "Что изменилось с тех пор, как мы получили версию вашей программы?" или "Как вы устранили такие-то ошибки?" К тому же наличие копий упрощает эксперимент: если у вас что-либо не получилось, то можно безболезненно вернуться к исходной программе.

Одно из решений состоит в том, чтобы хранить копии всех версий программы, но это трудно организовать и, кроме того, требует большого объема памяти на диске. Мы же будем основываться на подобии последовательных версий, что позволяет хранить только их общую часть. Команда

```
$ diff -e old new
```

порождает список команд редактора `ed`, преобразующих файл `old` в `new`. Таким образом, можно хранить все версии на базе одного файла, сохраняя одну полную версию и множество команд редактирования, преобразующих ее в любую другую версию.

Существуют два очевидных решения: хранить целиком последнюю версию и иметь команды редактирования для возврата к старым версиям или хранить самую старую версию и иметь команды редактирования для перехода к новым. Хотя второе решение чуть проще запрограммировать, первое предпочтительнее, поскольку из множества версий почти всегда интереснее выбрать новые.

Мы рассмотрим первое решение. В едином файле, называемом файлом истории, хранится текущая версия, за которой следует множество команд редактирования, преобразующих каждую версию в предыдущую (т. е. более старую). Любой набор команд редактирования начинается такой строкой:

```
@@@ пользователь дата сводка
```

Сводка — это одна строка, которая вводится пользователем и описывает изменения.

Для работы с версиями используются две команды: `get` выделяет версию из файла истории, а `put` заносит новую версию в файл истории после запроса на ввод сводки изменений. Прежде чем привести программу, покажем, как выполняются `get` и `put` и как сохраняется файл истории:

```
$ echo строка текста > junk
$ put junk
Summary: создадим новый файл                    Введите описание
get: no file junk.H                             Файл-история не существует
put: creating junk.H                            ...и put создает его
$ cat junk.H
строка текста
@@@ you Sat Oct 1 13:31:03 EDT 1983             сделаем новый файл
$ echo еще строка >>junk
$ put junk
Summary: одна строка добавлена
$ cat junk.H
строка текста
еще одна строка текста
@@@ you Sat Oct 1 13:31:28 EDT 1983 одна строка добавлена
2d
@@@ you Sat Oct 1 13:31:03 EDT 1983 сделаем новый файл
$
```

Команды редактирования представляют собой одну строку 2, которая исключает вторую строку файла, преобразуя новую версию в исходную:

```
$ rm junk
$ get junk                                      Самая новая версия
$ cat junk
строка текста
еще строка текста
$ get -l junk
$ cat junk                                      Версия новейшая, но одна
строка текста
$ get junk                                      Опять самая новая версия
$ replace еще 'другая' junk                     Изменим ее
$ put junk
Summary: изменена вторая строка
$ cat junk.H
строка текста
```

```
другая строка
@@@ you Sat Oct 1 13:34:07 EDT 1983         одна строка добавлена
2d
@@@ you Sat Oct 1 13:31:03 EDT 1983         создадим новый файл
$
```

Для получения нужной версии файла в файле истории записаны команды редактирования. Первая группа команд преобразует самую последнюю версию в предыдущую, вторая группа преобразует предыдущую в пред–предыдущую версию и т. д. Таким образом, мы преобразуем новый файл в его старую версию, запуская каждый раз редактор `ed`.

Очевидно, может возникнуть проблема, если в изменяемом файле есть строки, начинающиеся с трех символов. Кроме того, в разделе ошибок описания команды `diff(1)` (см. справочное руководство по UNIX) есть предупреждение о строках, состоящих из одной точки. Мы выбрали `@@@` для разделения команд редактирования, поскольку такая строка является редкостью для обычного текста.

Конечно, было бы полезно показать здесь процесс развития команд `put` и `get`, но из-за ограниченного объема книги мы приведем только их окончательные варианты. Команда `put` проще команды `get`:

```
# put:  install file into history

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$$ $1               # previous version
then                    # merge pieces
        cp $1 /tmp/put.b$$              # current version
        echo "@@@ `getname` `date` $Summary" >>/tmp/put.b$$
        diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$  # latest diffs
        sed -n '/^@@@/,$p' <$HIST >>/tmp/put.b$$ # old diffs
        overwrite $HIST cat /tmp/put.b$$        # put it back
else                    # make a new one
        echo "put: creating $HIST"
        cp $1 $HIST
        echo "@@@ `getname` `date` $Summary" >>$HIST
fi
```

```
rm -f /tmp/put.[ab]$$
```

После считывания одной строки сводки команда `put` обращается к `get` для получения предыдущей версии файла из файла истории. Флаг `-o` команды `get` указывает на переключение выходного файла. В том случае, когда `get` не может найти файл истории, она возвращает код завершения ошибки, и `put` создает файл истории. Если файл истории существует, то в командах после `then` создается временный файл такого формата: самая последняя версия, строка `@@@` , команды редактора для преобразования этой версии в предыдущую, старые команды редактора и строки `@@@`. В конце временный файл копируется в файл истории с помощью команды `overwrite`.

Команда `get` в отличие от `put` включает флаги:

```
# get:  extract file from history

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]$$; exit 1' 1 2 15
# split into current version and editing commands
sed <$INPUT -n '1,/^@@@/w /tmp/get.a'$$'
                /^@@@/,$w /tmp/get.b'$$
# perform the edits
awk </tmp/get.b$$ '
        /^@@@/  { count++ }
        !/^@@@/ && count > 0 && count <= - '$VERSION'
        END     { print "$d"; print "w", "'$OUTPUT'" }
' | ed - /tmp/get.a$$
rm -f /tmp/get.[ab]$$
```

Флаги выполняют обычные функции: `-i` и `-o` задают переключение входного и выходного потоков, — `-[0-9]` определяет версию: `-0` — новая версия (значение по умолчанию), `-1` — предыдущая версия и т. д.). Цикл по аргументам организуется с помощью

команд while, test и shift, а не с помощью for, поскольку некоторые флаги (-i, -o) используют еще один аргумент, и поэтому нужно сдвигать их командой shift, которая плохо согласуется с циклом for, если она находится внутри него. Флаг '-' редактора ed отключает вывод числа символов, обычный при чтении и записи в файл.

Строка

```
test -r $INPUT || {echo "get: no file $INPUT" 1>&2 exit 1;}
```

эквивалентна конструкции

```
if test ! -r $INPUT
then
    echo "get: no file $INPUT" 1>&2
    exit 1
fi
```

(такую конструкцию мы использовали в команде put), но запись ее короче, и она понятнее программистам, хорошо знакомым с операцией |. Команды, заключенные между { и }, выполняются не порожденным, а исходным интерпретатором. Это необходимо для того, чтобы команда exit обеспечивала выход из get, а не из порожденного интерпретатора. Символы { и } подобны do и done — они приобретают специальные значения, если следуют за точкой с запятой, символом перевода строки или другим символом завершения команды.

В заключение мы рассмотрим те команды в get, которые и решают задачу. Вначале с помощью редактора sed файл истории разбивается на две части, содержащие самую последнюю версию и набор команд редактирования. Затем в awk-программе обрабатываются команды редактирования. Строки @@@ подсчитываются (но не печатаются), и до тех пор, пока их число не превышает номера нужной версии, команды редактирования пропускаются (напомним, что действие, принятое по умолчанию, в awk-программе сводится к выводу входной строки). К командам редактирования из файла истории добавлены еще две команды ed : *$d* удаляет одну строку @@@, которую редактор sed оставил в текущей версии, а команда *w* помещает файл в отведенное ему место. Команда overwrite здесь не нужна, поскольку в get изменяется только версия файла, а не сам файл истории.

■ Упражнение: Напишите команду version, выполняющую два задания:

```
$ version -5 файл
```

выдает сводку изменений, дату изменения и имя пользователя, произведшего изменения выбранной из файла истории версии, а

```
$ version sep 20 файл
```

выдает номер версии, являющейся текущей 20 сентября. Типичное использование этой возможности ясно из обращения:

```
$ get `version sep 20 файл`
```

(Команда version может для удобства создавать эхо имени файла истории.)

■ Упражнение: Измените команды `get` и `put` так, чтобы для работы с файлом истории они использовали отдельный каталог, а не загромождали текущий каталог файлами.

■ Упражнение: Когда программа уже работает, не имеет смысла запоминать все версии. Как бы вы организовали исключение версий из середины файла истории?

5.10 Заключение

Когда перед вами встает задача написать новую программу, возникает естественное желание сделать это на своем любимом языке программирования. Для нас таким языком чаще всего оказывается `shell`, хотя синтаксис его несколько необычен. `Shell` — удивительный язык программирования. Безусловно, это язык высокого уровня: операторами в нем являются целые программы. Поскольку он диалоговый, программы могут создаваться в диалоговом режиме и доводиться до рабочего состояния небольшими шагами. Далее, если они предназначены не только для личного пользования, их можно "вылизывать" и повышать надежность в расчете на широкий круг пользователей. В тех редких случаях, когда `shell`-программа оказывается неэффективной, часть ее или вся она может быть переписана на языке Си, но на основе уже проверенного алгоритма с работающей реализацией. (В следующей главе мы несколько раз пройдем этот путь.)

Такой подход вообще характерен для программного мира UNIX — начинать работу не с нуля, а на базе того, что сделали другие, идти от простого к сложному, использовать программные средства для проверки новых идей.

В настоящей главе мы привели много примеров, которые легко реализовать с помощью языка `shell` и существующих программ. Иногда достаточно лишь переопределить аргументы, как это было сделано в случае с командой `cal`. Иногда полезны циклы языка `shell` по последовательности имен файлов или наборам команд (см., например, `watchtor` или `checkmail`). Для более сложных вариантов все равно требуется меньше усилий, чем при программировании на Си. Так, наша версия команды `news` на языке `shell` заменяет программу на Си в 350 (!) строк.

Однако дело не только в том, чтобы иметь возможность программировать на командном языке. Недостаточно иметь и множество программ. Проблема состоит в том, что все компоненты должны работать согласованно и придерживаться соглашений о представлении и передаче данных. Каждый компонент предназначен для решения одной задачи, причем наилучшим образом. Язык `shell` позволяет всякий раз, когда перед вами встает новая задача, связать различные программы воедино простым и эффективным способом. Именно интеграция — причина высокой продуктивности программного мира UNIX.

Историческая и библиографическая справка. Идеей использования команд `put` и `get` мы обязаны системе управления исходными текстами (Source Code Control System — SCCS), созданной М. Рочкиндом (The Source Code Control System. — IEEE Trans. on Software Engineering, 1975). Эта система более мощная и гибкая, чем наши простые программы; она предназначена для поддержания процесса создания больших программ. Однако основу SCCS составляет все та же программа `diff`.

Глава 6

Программирование с помощью стандартных функций ввода–вывода

До сих пор мы использовали существующие инструменты, чтобы разрабатывать новые, но сейчас уже достигнут разумный предел в создании новых средств с помощью `shell`, `sed` и `awk`. В этой главе нам предстоит написать простые программы на языке программирования Си. Основополагающая философия конструирования объектов, функционирующих совместно, будет по-прежнему оказывать влияние на построение программ, так как наша цель — подготовить инструменты, с которыми можно работать и на которые можно положиться. В каждом случае мы также попытаемся показать вам приемлемую стратегию реализации таких инструментов: начинать с минимума, обеспечивающего некоторые полезные свойства, а затем добавлять новые средства, если в них возникает необходимость.

Существуют веские причины для того, чтобы писать новые программы "с нуля". Так, может оказаться, что проблема, с которой мы столкнулись, просто не может быть решена с помощью имеющихся программ. Часто приходится иметь дело с нетекстовыми файлами. Например, большинство программ, которые демонстрировались ранее, действительно хорошо работали лишь с текстовой информацией либо слишком трудно достигалась должная ясность или эффективность, если применялись только `shell` и другие средства общего назначения. В подобных случаях реализация с использованием `shell` может быть полезна для апробирования программы и ее интерфейса с пользователем. (Если же программа работает достаточно хорошо, нет причины для ее переделки.) Уже знакомая вам программа zap является в этом смысле неплохим примером: требуется всего несколько минут, чтобы написать первую версию на `shell`, которая имеет адекватный пользовательский интерфейс, но слишком медленна.

Мы будем писать программы на языке Си — стандартном языке системы UNIX (ядро и все пользовательские программы написаны на Си), поскольку нет иного языка, хотя бы отчасти также хорошо поддерживаемого. Вы должны знать этот язык, по крайней мере в такой степени, чтобы свободно разбираться в предлагаемом здесь материале. Если это не так, прочтите книгу "Язык программирования Си" Б. Кернигана и Д. Ритчи (М.: Финансы и статистика, 1985)[1]. Мы также воспользуемся "стандартной библиотекой ввода–вывода" — набором функций, обеспечивающих программы на Си эффективными и переносимыми средствами ввода-вывода и системными услугами. Стандартные библиотеки ввода–вывода есть во многих, отличных от UNIX, системах, поддерживающих Си, поэтому программы, взаимодействия с системой которых ограничены возможностями

[1] Сейчас выпущено как переиздание этой книги, так и новое, третье издание на русском языке

таких библиотек, могут быть легко перенесены.

Примеры, подобранные в настоящей главе, имеют общее свойство: они представляют собой небольшие "инструменты", которые используются регулярно, но не являются частью седьмой версии системы. Если ваша система обладает подобными программами, вам будет легче сравнивать системы. Если же вы с ними еще не знакомы, то они могут оказаться вам чрезвычайно полезными. Эти программы помогут вам понять, что совершенной системы не существует, и для того чтобы добиться улучшения и устранить те или иные дефекты, достаточно приложить лишь небольшие усилия

6.1 Стандартные входной и выходной потоки: программа `vis`

Многие программы читают только из одного входного потока и пишут в один выходной поток: для таких программ полностью подходят функции ввода–вывода, использующие лишь стандартные входной и выходной потоки, и для того чтобы начать работу, этого почти всегда достаточно.

Проиллюстрируем изложенное с помощью программы `vis`, которая копирует свой стандартный входной поток в стандартный выходной, изображая при этом все непечатаемые символы в виде `\nnn`, где `nnn` — восьмиричное значение символа. `Vis` полезна для обнаружения "посторонних" или нежелательных символов, которые могут попасть в файлы. Например, `vis` будет печатать каждый символ "шаг назад" как `\010`, что является его восьмиричным значением:

```
$ cat x
abc
$ vis < x
abc\010\010\010 ___
$
```

Чтобы просмотреть несколько файлов с помощью этой элементарной версии `vis`, вы можете использовать `cat` для сбора файлов

```
$ cat файл1 файл2 ... | vis

...

$ cat файл1 файл2 ... | vis | grep '\\'

...
```

и избежать тем самым выяснения способа доступа к файлам из программы. Между прочим, может показаться, что подобную работу следует выполнить с привлечением `sed`, поскольку команда '1' выдает на экран непечатаемые символы в наглядном виде:

```
$ sed -n 1 x
abc<<<___
$
```

Результат выполнения программы sed, вероятно, вам покажется яснее, чем результат выполнения vis. Но применение sed к нетекстовым файлам бессмысленно:

```
$ sed -n 1 /usr/you/bin
$                                        Ничего в ответ!
```

(Так получилось на **PDP-11**; в одной из систем для **VAX** sed аварийно завершилась, возможно, потому, что ввод был воспринят как очень длинная текстовая строка.) Таким образом, sed нам не подходит, и мы вынуждены писать новую программу.

Простейшие функции ввода и вывода — getchar и putchar. При каждом вызове getchar появляется очередной символ из стандартного входного потока, которому может быть поставлен в соответствие файл, конвейер или терминал (последнее принимается по умолчанию). Программа "не знает", что конкретно он собой представляет. Аналогично putchar(c) помещает символ в стандартный выходной поток, который по умолчанию также связан с терминалом.

Функция printf(3) выполняет форматное преобразование при выводе. Вызовы printf и putchar могут следовать в любом порядке; выходной поток отразит порядок этих вызовов. Для форматного преобразования входного потока предусмотрена функция scanf(3); она читает входной поток и разбивает его, как требуется, на строки, числа и т. п. Вызовы scanf и getchar также могут чередоваться.

Приведем первую версию vis:

```
/* vis:  make funny characters visible (version 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);
}
```

Getchar возвращает из входного потока очередной байт или значение EOF, когда встречает конец файла (или ошибку). Между прочим, EOF не является байтом из файла; вспомните: во второй главе объяснялось, что такое "конец файла". Значение EOF отличается от значения любого байта, поэтому его трудно спутать с реальными данными; переменная c описана как int (целая), а не как char (символьная), так что она может хранить значение EOF. Строка

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

должна находиться в начале каждого исходного файла. Это заставляет компилятор Си читать файл макроопределений (`/usr/include/stdio.h`), в котором специфицированы стандартные функции и имена, в том числе и `EOF`. Мы будем использовать `<stdio.h>` как краткую запись полного имени файла.

Файл `<ctype.h>` — еще один файл макроопределений в `/usr/include`, который задает машинно–независимые макрокоманды (макросы) для классификации символов. Чтобы выяснить, принадлежит ли входной символ набору ASCII (т. е. его значение меньше 0200) и печатается ли он, мы использовали здесь `isascii` и `isprint`. Остальные макросы перечислены в табл. 6.1. Отметим, что `<ctype.h>` определяет символы "перевод строки", "табуляция" и пробел как непечатаемые.

`isalpha(c)`	Буква принадлежит алфавиту: `a-z A-Z`
`isupper(c)`	Прописная буква: `A-Z`
`islower(c)`	Строчная буква: `a-z`
`isdigit(c)`	Цифра: `0-9`
`isxdigit(c)`	Шестнадцатиричная цифра: `0-9 a-f A-F`
`isalnum(c)`	Буква или цифра
`isspace(c)`	Пробел, символ табуляции, символ перевода строки, символ вертикальной табуляции, символ перевода страницы, символ возврата
`ispunct(c)`	Не буквенно-цифровой символ, не управляющий, не пробел
`isprint(c)`	Печатаемый: любой графический символ
`iscntrl(c)`	Управляющий символ: `0 <= c < 040 || c == 0177`
`isascii(c)`	Символ ASCII: `0 <= c <= 0177`

Таблица 6.1: Макросы классификации символов `<ctype.h>`

Вызов `exit` в конце `vis` не является необходимым для корректной работы программы, но гарантирует тому, кто эту программу вызвал, получение нормального кода ее завершения (обычно нуля). Другой способ возврата кода завершения — выполнить в теле функции `main` оператор return 0; возвращаемое значение `main` и есть код завершения программы. Если нет явно указанных `return` или `exit`, код завершения не определен.

Для компиляции программы на Си поместите исходный текст в файл, имя которого оканчивается на `.c`, например `vis.c`, оттранслируйте его с помощью `cc` и запустите на выполнение результат, оставляемый компилятором в файле с именем `a.out` ('a' — ассемблер):

```
$ cc vis.c
$ a.out
hello worldctl-g
hello world\007
ctl-d
$
```

`a.out` можно переименовать после первого запуска или сделать это сразу с помощью флага `-o` команды `cc`:

```
$ cc -o vis vis.c                    Результат в vis, а не в a.out
```

```
%Упражнение 6.1.
  Мы решили, что символы табуляции не следует делать видимыми,
  изображая их как \code{\textbackslash 011}, \code{->} или
  \code{\textbackslash t}, поскольку главное назначение \cmd{vis} ---
  поиск действительно аномальных символов. Можно принять
  альтернативное решение и недвусмысленно идентифицировать каждый
  символ в выходном потоке: символы табуляции, неграфические символы,
  пробелы в конце строки и т. п. Модифицируйте \cmd{vis} так, чтобы
  символы табуляции, обратная дробная черта, ``шаг назад'', перевод
  страницы и др. печатались в традиционном, принятом в Си
  представлении: \code{\textbackslash t}, \code{\textbackslash
    \textbackslash}, \code{\textbackslash b}, \code{\textbackslash f}
  и т. д., причем пробелы в конце строки должны быть помечены.  Можете
  сделать это недвусмысленным образом? Сравните ваш вариант с
  приведенным ниже:

\begin{verbatim}
$ sed -n 1
\end{verbatim}
%$
```

```
%Упражнение 6.2.
  Модифицируйте \cmd{vis} так, чтобы она приводила длинные строки к
  строкам некоторой разумной длины. Как это согласуется с требованием
  недвусмысленности результата из предыдущего упражнения?
```

6.2 Аргументы программы: `vis` версия 2

Когда выполняется программа на Си, функции `main` передаются следующие аргументы из командной строки: счетчик `argc` и массив `argv`, состоящий из указателей символьных строк, содержащих аргументы. По соглашению `argv[0]` это имя самой команды, так что argc всегда больше нуля; "полезными" же являются аргументы `argv[1]` ... `argv[argc - 1]`. Вспомните, что переключение входного или выходного потоков с помощью < и > осуществляется в `shell`, а не отдельными программами, поэтому такое переключение не влияет на число аргументов, видимых программой.

Для иллюстрации работы с аргументами модифицируем `vis`, добавив флаг: `vis -s` удаляет любые непечатаемые символы вместо того, чтобы выделять их. Такое удаление удобно для "чистки" файлов из других систем, например тех, которые используют для завершения строки *CRLF* (символы возврата каретки и перевода строки) вместо одного символа перевода строки.

```
/* vis:  make funny characters visible (version 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);
}
```

Здесь `argv` — указатель массива, элементы которого служат указателями массивов символов; каждый такой массив заканчивается символом ASCII NUL ('\0'), поэтому массив можно считать строкой. Эта версия `vis` начинает свою работу с того, что проверяет, есть ли аргумент и является ли он `-s`. (Неверные аргументы игнорируются.) Функция `strcmp(3)` сравнивает две строки, возвращая нуль, если они одинаковы.

`strcat(s,t)`	Добавляет строку `t` к строке `s`; возвращает `s`
`strncat(s,t,n)`	Добавляет не более `n` символов `t` к `s`
`strcpy(s,t)`	Копирует `t` в `s`; возвращает `s`
`strncpy(s,t,n)`	Копирует точно `n` символов; при необходимости добавляет `NULL`
`strcmp(s,t)`	Сравнивает `s` и `t`, возвращает `<0`, `0`, `>0` при `<`, `==`, `>`
`strncmp(s,t,n)`	Сравнивает не более `n` символов
`strlen(s)`	Возвращает длину `s`
`strchr(s,c)`	Возвращает указатель на первый символ `c` в `s` и `NULL`, если `c` отсутствует
`strrchr(s,c)`	Возвращает указатель на последний `c` в `s` и `NULL`, если `c` отсутствует.
`atoi(s)`	Возвращает целое значение `s`
`atof(s)`	Возвращает "плавающее" значение `s`; необходимо описание `double atof()`
`malloc(n)`	Возвращает указатель на область памяти в `n` байт и `NULL`, если это невозможно
`calloc(n,m)`	Возвращает указатель на `n*m` обнуленных байтов и `NULL`, если это невозможно;

Таблица 6.2: Стандартные функции, выполняемые над строками

`malloc` и `calloc` возвращают значение типа `char*`, `free(p)` освобождает память, выделенную `malloc` и `calloc`

В табл. 6.2 перечислены некоторые средства работы со строками и ряд полезных функций, одна из которых — `strcmp`. Как правило, лучше воспользоваться этими стандартными функциями, чем писать собственные, так как они отлажены и зачастую выполняются быстрее, чем написанные вами, поскольку были оптимизированы для конкретных машин (нередко благодаря использованию Ассемблера).

```
%Упражнение 6.3.
  Измените аргумент \cmd{-s} так, чтобы \cmd{vis -sn} печатала только
  строки из \code{n} или более печатаемых символов, опуская
  непечатаемые символы и короткие последовательности обычных,
  печатаемых символов. Это полезно при выделении ``текстовых'' частей
  в нетекстовых файлах, таких, как рабочие программы.  Некоторые
  версии системы содержат для подобных целей программу \cmd{strings}.
  Что лучше: иметь отдельную программу или пользоваться специальным
  аргументом \cmd{vis}?
```

```
%Упражнение 6.4.
  Доступность исходной программы на Си --- одно из достоинств системы
  \UNIX; такая программа демонстрирует элегантные решения многих
  программистских проблем. Прокомментируйте баланс между наглядностью
  программы на Си и встречающимися ``оптимизированными'' фрагментами,
  переписанными на Ассемблере.
```

6.3 Доступ к файлам: `vis` версия 3

Две первые версии `vis` читают из стандартного входного потока и пишут в стандартный выходной поток, причем оба потока наследуются от `shell`. Следующий шаг — модификация `vis` для работы с файлами по их именам, так что

```
$ vis файл1 файл2 ...
```

будет просматривать эти именованные файлы вместо стандартного входного потока. Если же имен файлов в качестве аргументов нет, `vis` должна читать стандартный входной поток.

Возникает вопрос: как организовать чтение файлов, т. е. как связать имена файлов с операторами ввода–вывода, реально читающими данные? Правила просты. Прежде чем быть прочитанным или записанным, файл должен быть открыт стандартной библиотечной функцией `fopen`. Последняя берет имя файла (например, `temp` или `/etc/passwd`), взаимодействует с ядром и возвращает обратно “внутреннее имя”, которое используется при последующих операциях с данным файлом.

Внутреннее имя является на самом деле указателем (называемым указателем файла) на структуру, содержащую информацию о файле, такую, как расположение буфера, текущую позицию символа в буфере, режим чтения или записи и т. п. Эта структура определяется в файле `<stdio.h>` и имеет имя `FILE`. Описание указателя файла таково:

```
FILE *fp;
```

Оно означает, что `fp` — указатель на `FILE`. `fopen` возвращает указатель на `FILE`; в `<stdio.h>` имеется описание типа для `fopen`. Реальный вызов функции `fopen`:

```
char *name, *mode;
fr = fopen(name, mode);
```

Первый аргумент fopen представляет собой имя файла (строку символов). Второй аргумент также является символьной строкой, показывающей, как вы намереваетесь использовать файл; допустимые режимы: читать (`"r"`), писать (`"w"`) или дописать (`"a"`).

Если файл, который вы открыли для записи или дописывания, не существует, он создается, если это возможно. Открытие для записи существующего файла вызывает уничтожение старого содержимого. Попытка читать несуществующий файл считается ошибкой, так же как и попытка читать или писать файл без разрешения. При возникновении ошибки fopen возвращает значение несуществующего указателя `NULL` (которое обычно определяется в `<stdio.h>` как `(char*)0`).

Далее, нужен способ читать или писать файл после того, как он открыт. Есть несколько способов, из которых использование `getc` и `putc` — самый простой. Функция `getc` выбирает из файла очередной символ:

```
c = getc(fr)
```

помещает в c следующий символ из файла, на который указывает `fp`. Эта функция возвращает `EOF` по достижении конца файла. Функция `putc` аналогична `getc`:

```
putc(c, fp)
```

помещает символ `c` в файл `fp` и возвращает `c`. Функции `getc` и `putc` возвращают `EOF` в случае ошибки.

Когда программа начинает выполняться, уже открыты три файла и имеются их указатели. Это стандартные потоки: входной, выходной и поток диагностики; соответствующие указатели называются `stdin`, `stdout` и `stderr`. Указатели на файлы описаны в `<stdio.h>` и могут использоваться там, где может быть объект типа `FILE*`. Они являются не переменными, а константами, так что им нельзя присвоить значения. Вызов `getchar()` есть `getc(stdin)`, а `putchar(c)` есть `putc(c, stdout)`. На самом деле все эти четыре "функции" определены в `<stdio.h>` как макрокоманды. Они выполняются быстрее обычных вызовов функций ввиду отсутствия накладных расходов по вызову функции для каждого символа (см. табл. 6.3 с некоторыми другими определениями из `<stdio.h>`).

Теперь вернемся снова к нашей теме и напишем третью версию `vis`. Если есть аргументы командной строки, они обрабатываются в порядке очередности, если же аргументов нет, — используется стандартный входной поток.

```
/* vis:  make funny characters visible (version 3) */

#include <stdio.h>
#include <ctype.h>
int strip = 0;          /* 1 => discard special characters */
```

stdin	Стандартный входной поток
stdout	Стандартный выходной поток
stderr	Стандартный поток диагностики
EOF	Конец файла; обычно -1
NULL	Несуществующий указатель; обычно 0
FILE	Используется для описания указателей на файлы
BUFSIZ	Обычно 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
putchar(c)	putc(c,stdout)
feof(fp)	Не нуль, если достигнут конец файла для потока fp
ferror(fp)	Не нуль, если в потоке fp есть ошибка
fileno(fp)	Дескриптор файла для потока fp (см. гл. 7)

Таблица 6.3: Некоторые определения из <stdio.h>

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    FILE *fp;

    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
        case 's':       /* -s: strip funny chars */
            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",
                    argv[0], argv[i]);
                exit(1);
            }
            else {
                vis(fp);
```

```
                fclose(fp);
            }
    exit(0);
}
```

В программе принято соглашение, по которому флаги стоят в начале списка аргументов. После обработки каждого флага argv и argc модифицируются так, что остальная часть программы не зависит от присутствия этого флага. Даже если `vis` распознает единственный флаг, мы написали программу в виде цикла, чтобы продемонстрировать единый способ обработки аргументов. В гл. 1 отмечалось, что программы UNIX обрабатывают флаги в произвольном порядке. Как одну из причин (помимо склонности к анархии) здесь можно назвать очевидную легкость написания программы разбора аргументов при любой модификации. Включение функции `getopt(3)` в некоторые системы является попыткой рационально объяснить ситуацию; вы можете ее исследовать, прежде чем писать собственную.

Процедура `vis` выводит на печать единственный файл:

```
vis(fp) /* make chars visible in 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` идентична `printf`, за исключением аргумента–указателя, специфицирующего файл, в который нужно писать.

Функция `fclose` разрывает связь между указателем и внешним именем файла, установленную с помощью `fopen`, освобождая указатель для другого файла. Так как существует ограничение (около 20) на число файлов, которые одновременно могут быть открыты в программе, лучше всего закрывать уже не требующиеся вам файлы. Обычно выходной поток, выдаваемый любой стандартной библиотечной функцией, подобной `printf`, `putc` и т. д., для большей эффективности буферизуется так, чтобы его можно было писать большими фрагментами. (Исключение составляет выходной поток терминала, который, как правило, пишется по мере своего формирования или при печати символа перевода строки.) Применение `fclose` к выходному файлу инициирует выдачу последней буферизованной порции. `fclose` также вызывается автоматически для каждого открытого файла, когда программа выполняет `exit` или возвращается из `main`.

Стандартный поток `stderr` присваивается программе тем же способом, что и `stdin` и `stdout`. Информация, записанная в `stderr`, оказывается на терминале пользователя даже при изменении назначения стандартного выходного потока. `Vis` пишет свою диагностику в `stderr` вместо `stdout`, так что если один из файлов по каким-то причинам недоступен, сообщение найдет путь на терминал пользователя, а не исчезнет в программном канале или в выходном файле. (Стандартный поток диагностики был изобретен позднее, чем

программные каналы: после того, как сообщения об ошибках стали исчезать при передаче через эти каналы.)

Мы решили, отчасти произвольно, что `vis` завершается, если не может открыть входной файл; это разумно для программы, чаще всего используемой в режиме диалога и с одним входным файлом. Однако вы можете предложить и другое решение.

```
%Упражнение 6.5.
  Напишите программу \cmd{printable}, которая печатает имя каждого
  аргумента--файла, содержащего только печатаемые символы; если в
  файле хранится любой непечатаемый символ, имя не печатается.
  \cmd{printable} полезна в ситуациях, подобных следующей:

\begin{verbatim}
$ pr `printable *` | lpr
\end{verbatim}
%$

  Добавьте фрлаг \cmd{-v}, чтобы изменить смысл проверки на обратный,
  как в \cmd{grep}. Что следует делать, если среди аргументов нет имен
  файлов? Какой код завершения должна передавать \cmd{printable} при
  возврате?
```

6.4 Вывод на экран порциями: программа p

До сих пор мы использовали `cat` для просмотра файлов. Но если файл длинный, а связь с системой высокоскоростная, `cat` выдает выходной файл слишком быстро, что затрудняет его чтение, даже если вы успеваете делать это с помощью *ctl-s* и *ctl-q*.

Очевидно, нужно иметь программу для печати файла небольшими удобными порциями, но такая программа не является стандартной, возможно, потому, что первоначальная система UNIX была написана в те времена, когда использовались терминалы "твердой копии" (печати на бумаге) и медленные линии связи. Поэтому наш следующий пример — программа `p`, которая выводит файл по частям, занимающим полный экран, за один раз, ожидая ответа от пользователя после выдачи каждой порции прежде, чем начать очередной вывод (краткое имя `p` весьма удобно для программы, которая часто используется). Подобно другим программам, `p` читает или из файлов, поименованных, как аргументы, или из стандартного входного потока:

```
$ p vis.c

...

$ grep '#define' *.[ch] | p

...

$
```

Эту программу легче всего писать на Си; стандартные средства неудобны, когда происходит смешанный ввод из файла или конвейера и с терминала. Решение состоит в том, чтобы печатать входной поток небольшими порциями. Удобный размер порции — 22 строки, что составляет немногим меньше, чем размер в 24 строки на большинстве видеотерминалов, и одну треть стандартной страницы в 66 строк. Простой способ подсказки пользователю — не печатать последний символ перевода строки каждой порции. Курсор остановится на правом конце строки, а не на левой границе (новой строки). При нажатии клавиши *RETURN* выполняется перевод строки, и следующая строка появляется в нужном месте. Если пользователь печатает *ctl-d* или *q* в конце экрана, выполнение программы `p` заканчивается.

Мы не станем принимать специальных мер для вывода длинных строк. Мы также не будем беспокоиться о множестве файлов: просто перейдем от одного к другому без примечаний. Следующая команда

```
$ p имена файлов...
```

по своему действию аналогична команде

```
$ cat имена файлов... | p
```

Если нужны имена файлов, их можно добавить циклом `for`:

```
$ for i in имена файлов
> do
>     echo $i:
>     cat $i
> done | p
```

В самом деле, у нас оказывается слишком много средств, которые мы можем внести в программу. Лучше сделать "неоснащенную" версию, а затем развивать ее, как подскажет опыт. Иными словами, необходимые средства — это те, которые действительно вам нужны, а не те, которые, по нашему мнению, вы хотели бы иметь.

Структура `p` аналогична структуре `vis`: основная процедура выполняет цикл по файлам, вызывая функцию `print`, выполняющуюся с каждым файлом:

```
/* p:  print input in chunks (version 1) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

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` реализует весьма общую операцию: пытается открыть файл. Если же это невозможно, она выводит на печать сообщение об ошибке, и ее выполнение завершается. Чтобы обеспечить выдачу сообщений об ошибках, идентифицирующих программу, в которой происходит (или произошла) ошибка, `efopen` ссылается на внешнюю строку `progname`, где содержится имя программы, устанавливаемое в `main`:

```
FILE *efopen(file, mode)        /* fopen file, die if can't */
    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);
}
```

Мы испытали две версии программы `efopen`, прежде чем остановиться на данной. Одна из них должна была после печати сообщения завершиться, возвратив нулевой указатель, свидетельствующий о неудаче. Это позволяет вызвавшей программе продолжить свое выполнение или завершиться. Другая версия снабжала efopen третьим аргументом, указывающим, следует ли возвращаться после того, как файл открыть не удалось. Почти во всех наших примерах, однако, нет смысла продолжать работу, если файл недоступен, так что текущая версия `efopen` является для нас наилучшей.

Непосредственное выполнение команды `p` осуществляется в `print`:

```
print(fp, pagesize)     /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;   /* number of lines so far */
    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;
        }
}
```

Мы использовали здесь BUFSIZ, который определен в <stdio.h> как размер буфера входного потока. Функция fgets(buf,size,fp) выбирает следующую строку входного потока из fp до символа перевода строки (включая его) в буфер и добавляет завершающий символ \0. Копируется на более size - 1 символов. По достижении конца файла возвращается NULL. (Конструкция fgets оставляет желать лучшего: она возвращает buf вместо счетчика символов и, кроме того, выдает предупреждение о том, что входная строка была слишком длинной. Символы не потеряны, но вы должны взглянуть на buf, чтобы понять, что в самом деле случилось.)

Функция strlen возвращает длину строки, поэтому мы можем отбросить завершающий символ перевода строки последней входной строки. После вызова fputs(buf, fp) строка buf записана в файл fp. При вызове fflush в конце страницы происходит вывод буферизованного выходного текста.

Считывание ответа пользователя в конце каждой страницы возложено на функцию ttyin. Функция ttyin не может читать стандартный входной поток, тогда как p должна выполняться, даже если входной поток поступает из файла или конвейера. Чтобы справиться с этим, программа открывает файл /dev/tty, которому поставлен в соответствие пользовательский терминал при любом переключении стандартного входного потока. Приведенная ниже функция ttyin возвращает первую букву ответа, но здесь это свойство не используется.

```
ttyin() /* process response from /dev/tty (version 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 devtty описан как статический, так что его значение сохраняется от одного вызова ttyin до другого; файл /dev/tty открывается только при первом вызове.

Очевидно, есть дополнительные средства, которые без особых усилий можно ввести в p, однако наша первая версия этой программы только печатает 22 строки и ждет следующей порции. Прошло немало времени, прежде чем в нее были добавлены другие

средства, но в настоящее время ими мало кто пользуется. В частности, весьма простое дополнение — ввод переменной `pagesize` для хранения числа строк на странице. Значение переменной можно установить из командной строки

```
$ p -n...
```

Она печатает порции по `n` строк. Для этого требуется лишь добавить несколько знакомых вам операторов в начале `main`:

```
/* p:  print input in chunks (version 2) */

...

    int i, pagesize = PAGESIZE;

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }

...
```

Функция `atoi` превращает строку символов в целое число (см. справочное руководство по `atoi(3)`).

Еще одно средство — временно остановить вывод на экран в конце каждой страницы, чтобы выполнить какую-либо иную команду. По аналогии с `ed` и многими другими программами, если пользователь печатает строку, начинающуюся восклицательным знаком, остальная часть строки воспринимается как команда и передается `shell` для выполнения. Данное средство также тривиально, поскольку для этой цели предусмотрена функция `system(3)`, речь о которой пойдет ниже. Модифицированная версия `ttyin` такова:

```
ttyin() /* process response from /dev/tty (version 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);  /* BUG here */
            printf("!\n");
        }
```

```
        else    /* ordinary line */
            return buf[0];
    }
}
```

К сожалению, эта версия `ttyin` имеет серьезный недостаток. Команда, запущенная с помощью `system`, получает стандартный входной поток от `p`, так что если `p` читает из программного канала или файла, их входные потоки могут мешать друг другу:

```
$ cat /etc/passwd | p -1
root:3d.fHR5KoB.3s:0:1:S.User:/:!ed     Вызвать ed из p
?                                       ed читает /etc/passwd
! ...                                   запутывается и завершается
```

Для решения этой проблемы необходимо знать, как управлять процессами в UNIX, о чем речь пойдет в разд. 7.4. Пока же примите к сведению, что использование стандартной библиотечной функции `system` может создать неприятности, однако `ttyin` работает правильно, если компилируется с версией `system`, описанной в гл. 7.

Итак, мы написали две программы `vis` и `p`, которые можно считать вариантами `cat` с некоторыми "украшениями". Может быть, им следует быть частью `cat`, доступной с помощью флагов `-v` и `-p`? Вопрос о том, писать ли новую программу или добавлять какие-то средства к старой, возникает всегда, как только у людей появляются новые идеи. Мы не можем со всей определенностью ответить на данный вопрос, но приведем здесь некоторые принципы, которые, возможно, вам помогут.

Основной принцип состоит в том, что программе следует выполнять только свою основную работу. Если у нее появляется слишком много функций, она становится большой, медленной, ее трудно сопровождать и использовать. В самом деле, ряд свойств часто остается невостребованным, поскольку пользователи никак не могут запомнить флаги.

Поэтому `cat` и `vis` совмещать не рекомендуется. Если `cat` просто копирует входной поток без изменений, то `vis` его трансформирует. Соединение их дает программу с двумя разными функциями. Это очевидно также для `cat` и `p`: `cat` предназначена для быстрого эффективного копирования страниц, `p` — для их "перелистывания". Кроме того, `p` преобразует выходной поток. Каждый 22-й символ перевода строки пропускается. Три отдельные программы представляются в таком случае правильным решением.

■ Упражнение: Работает ли `p` нормально, если `pagesize` не является положительным?

■ Упражнение: Что еще можно было бы сделать с `p`? Оцените и реализуйте (если оно вам подходит) свойство вновь выводить части ранее введенного текста. (Это дополнительное средство нам очень нравится.) Добавьте возможность выводить неполное содержимое экрана после каждой паузы, а также просматривать текст вперед или назад по строкам, задаваемым номером или содержимым.

■ Упражнение: Используйте средства манипуляций файлами, встроенные в `exec shell` (см. справочное руководство по `sh(1)`), чтобы фиксировать обращения к system с терминала `ttyin`.

■ Упражнение: Если вы забыли определить источник ввода для `p`, то программа "молча" ожидает ввода с терминала. Стоит ли искать эту возможную ошибку? Если да, то как? Подсказка: `isatty(3)`.

6.5 Пример: pick

Версия `pick` из гл. 5, несомненно, увеличивает возможности `shell`. Версия на Си, приведенная ниже, в чем-то отличается от рассмотренной в гл. 5. Если эта версия имеет аргументы, то они обрабатываются так же, как и ранее, но если определен единственный аргумент '`-`', `pick` обрабатывает свой стандартный входной поток.

Почему бы в отсутствие аргументов просто не читать стандартный входной поток? Рассмотрим вторую версию команды `zap` из разд. 5.6:

```
kill $SIG `pick\`ps-ag |egrep "$*"\` | awk '{print $1}'`
```

Что происходит, если шаблон `egrep` ни с чем не совпадает? В этом случае `pick` не имеет аргументов и читает свой стандартный входной поток; команда `zap` терпит неудачу загадочным образом. Требование явного аргумента — простой способ устранить неоднозначность, и соглашение о '`-`' в `cat` и других программах показывает, как его определить.

```
/* pick:  offer choice on each argument */

#include <stdio.h>
char    *progname;      /* program name for error message */

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';  /* drop newline */
            pick(buf);
        }
    else
        for (i = 1; i < argc; i++)
            pick(argv[i]);
    exit(0);
}
```

```
pick(s) /* offer choice of s */
    char *s;
{
    fprintf(stderr, "%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}
```

Версия `pick` предоставляет возможность диалогового выбора аргументов в одной программе. Это не только обеспечивает полезное средство, но и уменьшает потребность в "интерактивных" флагах для других команд.

■ Упражнение: Если есть `pick`, существует ли необходимость в `rm -i`?

6.6 Об ошибках и отладке

Если вы писали программы ранее, вам знакомо понятие ошибки. Однако важно не только создавать программы, свободные от ошибок, но и заботиться о том, чтобы ваш проект был прост, тщательно реализован и сохранял свою "чистоту" в процессе модификации.

В UNIX много инструментов, которые помогут вам находить ошибки, хотя ни один из них не является действительно первоклассным. Для того чтобы продемонстрировать их, нам нужна ошибка; все же программы в этой книге совершенны. Поэтому мы "создадим" типичную ошибку. Рассмотрим приведенную выше функцию `pick`, но на сей раз с ошибкой (заглядывать в первоначальный вариант нечестно):

```
pick(s) /* offer choice of s */
    char *s;
{
    fprintf("%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}
```

Что произойдет, если мы откомпилируем и запустим ее?

```
$ cc pick.c -o pick
$ pick *.c                                    Попробуем
Ошибка при обращении к памяти - сделан дамп   Катастрофа!
$
```

Сообщение "Ошибка при обращении к памяти" свидетельствует о том, что ваша программа пыталась работать с недозволенной областью памяти. Обычно в таком случае указатель содержит неправильное значение. "Ошибка адресации шины" — другое диагностическое сообщение со сходным значением, часто обусловленное просмотром бесконечной строки. "Сделан дамп памяти" означает, что ядро сохранило состояние вашей выполняемой программы в файле core текущего справочника. Вы также можете заставить программу сделать дамп памяти, напечатав *ctl*-\, если она выполняется как фоновая, или с помощью команды `kill -3`, если она основная.

Существуют две программы adb и sdb, назначение которых — разбираться в "посмертной выдаче". Подобно большинству отладчиков, они "хитроумны", сложны и без них трудно обойтись. Программа adb есть в седьмой версии системы, а sdb доступна в более поздних версиях.

Из-за ограниченного объема книги мы лишь частично покажем вам применение каждой программы, а именно распечатаем содержимое стека, т. е. выведем функцию, выполнявшуюся при аварийном завершении программы, функцию, которая ее вызывала, и т. д. Первая функция, указанная в распечатке стека, это то место, где находилась программа, когда она была аварийно завершена.

Чтобы получить распечатку стека с помощью adb, нужно ввести команду *$C*:

```
$ adb pick core                          Вызывает adb
$C                                       Запрос содержимого стека
~_strout(0175722,011,0,011200)
    adjust:     0
    fillch:     060542
__doprnt(0177345,0176176,011200)
~fprintf(011200,0177345)
    iop:     01120
    fmt:     0177345
    args:    0
~pick(0177345)
    s:       0177345
~main(035,0177234)
    argc:    035
    argv:    0177234
    i:       01
    buf:     0
ctl-d                                    Завершение
$
```

Здесь речь идет о том, что main была вызвана из pick, которая вызвала fprintf, а она в свою очередь вызвала __doprnt, вызвавшую _strout. Так как __doprnt не упомянута где-либо в pick.c, ошибка должна быть где-то в fprintf или выше. (Строки после каждой функции в распечатке показывают значения локальных переменных. *$C* подавляет данную информацию так же, как сама *$C* делает это в некоторых версиях adb.)

Попытаемся теперь сделать то же самое с помощью sdb:

```
$ sdb pick core
Предупреждение: 'a.out не компилируется с -g
lseek: adress 0xa64              Функция, где программа аварийно завершилась
*t                               Запрос распечатки стека
lseek()
fprintf(6154,2147479154)
pick(2147479154)
main(30,2147478988,2147479112)
*q                               Выход
$
```

Информация размещена по-иному, но есть общая основа: `fprintf`. (Распечатка стека другая, так как это сделано на машине **VAX-11/750**, на которой стандартная библиотека ввода–вывода реализована иначе.) И если мы взглянем на вызов `fprintf` в неправильной версии `pick`, то обнаружим некорректность:

```
fprintf ("%s?", s);
```

Здесь нет `stderr`, так что строка формата используется как ссылка к `FILE`, и, конечно, получается хаос.

Мы показали вам типичную ошибку, которая является скорее результатом просмотра, а не неправильного программирования. Искать подобные ошибки при вызове функции с неверными аргументами можно также с помощью верифицирующей программы для Си `lint(1)`. Эта программа рассматривает Си-программы с точки зрения наличия ошибок, аспектов переносимости и сомнительных конструкций. Если мы запустим `lint` с файлом `pick.c`, ошибка идентифицируется:

```
$ lint pick.c

...

fprintf, arg. 1 несовместим "llib-lc"(69) :: "pick.c"(28)

...

$
```

Это означает, что первый аргумент в стандартной библиотеке определен иначе, чем в строке 28 вашей программы. Таким образом дана точная информация о том, что неверно.

Программа `lint`, с одной стороны, указывает на недостатки в вашей программе, а с другой — выдает много не относящихся к делу сообщений, которые мы выше опустили, и нужен некоторый опыт, чтобы уметь разбираться, какие из них необходимы, а какие следует игнорировать. Однако имеет смысл постараться, так как `lint` помогает обнаружить некоторые ошибки, которые человек увидеть практически не может. После длительного редактирования всегда стоит запустить `lint` и убедиться в том, что каждое выдаваемое этой программой предупреждение вам понятно.

6.7 Пример: `zap`

Программа `zap`, которая избирательно уничтожает процессы, отличается от той, что была представлена в виде файла `shell` в гл. 5. Главная проблема данной версии — скорость. Она создает много процессов и поэтому работает медленно, что недопустимо для программы, уничтожающей процессы с ошибками. Если переписать `zap` на Си, ее быстродействие повысится. Мы, однако, снова воспользуемся ps, чтобы найти информацию о процессе. Это намного легче, чем выуживать информацию из ядра, и, кроме того, мы имеем переносимый вариант. Программа `zap` открывает программный канал, входной поток для которого берется из `ps`, и читает из него, как из файла. Функция `popen(3)` аналогична `fopen`, за исключением того, что первый аргумент является командой, а не именем файла. То же самое справедливо и для `pclose`, но здесь она нам не нужна.

```
/* zap:  interactive process killer */

#include <stdio.h>
#include <signal.h>
char    *progname;      /* program name for error message */
char    *ps = "ps -ag"; /* system dependent */

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);    /* get header line */
    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);
}
```

Мы писали программу, чтобы использовать `ps -ag` (этот флаг системно–зависим), но если вы не являетесь привилегированным пользователем, то можете уничтожать лишь свои собственные процессы.

Первый вызов `fgets` выбирает заголовок из `ps`; интересно выяснить, что случится, если попытаться уничтожить "процесс", соответствующий данному заголовку.

Функция `sscanf` представляет собой член семейства `scanf(3)` для форматного преобразования входной строки. Она преобразует строку, а не файл. Вызов `kill` из системы посылает специальный сигнал процессу; сигнал `SIGKILL`, определенный в `<signal.h>`, не может быть перехвачен или проигнорирован. Вы можете вспомнить пятую главу, где его численное значение равно девяти, но лучше использовать символические константы из файлов макроопределений, чем включать в свои программы загадочные числа.

Если аргументы отсутствуют, `zap` предоставляет каждую строку выходного потока `ps` как возможность для выбора. При наличии аргумента `zap` предлагает только те выходные строки `ps`, которые ему соответствуют. Функция `strindex(si, s2)` проверяет,

соответствует ли аргумент какой-либо части строки выходного потока `ps`, используя `strncmp` (см. табл. 6.2). Функция `strindex` возвращает позицию `s2` в `si` или `-1`, если ее там нет.

```
strindex(s, t)  /* return index of t in s, -1 if none */
    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;
}
```

В табл. 6.4 представлены широко используемые функции из стандартной библиотеки ввода–вывода.

■ Упражнение: Модифицируйте `zap` так, чтобы можно было применять любое число аргументов. В настоящем виде `zap` высвечивает на экране строку, соответствующую выбранному варианту. Будет она делать это? Если нет, модифицируйте программу соответствующим образом. Подсказка: `getpid(2)`.

■ Упражнение: Постройте `fgrep(1)` на основе `strindex`. Сравните время работы при сложных поисках, например 10 слов на документ. Почему `fgrep` выполняется быстрее?

6.8 Диалоговая программа сравнения файлов: `idiff`

Поддерживать две чем-то отличающиеся версии файла, каждая из которых содержит часть нужного вам файла, — довольно распространенная проблема. Зачастую она возникает в тех случаях, когда изменения вносятся независимо двумя разными людьми. Программа `diff` подскажет вам, чем различаются файлы, но вы не получите никакой помощи, если захотите выбрать какую-то информацию из одного файла, а какую-то — из другого.

В этом разделе мы напишем программу `idiff` (диалоговая `diff`), которая предоставляет пользователю каждую порцию выходного потока `diff` и предлагает ему возможность выбора фрагментов “от и до” или их редактирования. Программа `idiff` помещает выбранные фрагменты в соответствующем порядке в файл `idiff.out`. Допустим, даны такие два файла:

```
file1:              file2:
This is             This is
a test              not a test
of                  of
your                our
skill               ability
```

`fp=fopen(s, mode)`	Открыть файл `s`; значения mode `"r"`, `"w"`, `"a"` соответствуют чтению, записи и добавлению (при ошибке возвращается `NULL`)
`c=gets(fp)`	Читать символ: `getchar()` это `getc(stdin)`
`putc(c, fp)`	Записать символ: `putchar(c)` это `putc(c, stdout)`
`ungetc(c, fp)`	Вернуть символ во входной файл `fp`; можно вернуть не более одного символа за раз
`scanf(fmt, a1, ...)`	Читать символы из `stdin` в `a1, ...` в соответствии с `fmt`. Каждый `ai` должен быть указателем
`fscanf(fp,...)`	Читать из файла `fp`
`sscanf(s,...)`	Читать из строки `s`
`printf(fmt, a1, ...)`	Форматировать `a1, ...` в соответствии с `fmt`; печатать в `stdout`
`fprintf(fp, ...)`	Печатать `...` в файл `fp`
`sprintf(s, ...)`	Печатать `...` в строку `s`
`fqets(s, n, fp)`	Читать не более `n` символов в `s` из `fp` (возвращается `NULL` по концу файла)
`fputs(s, fp)`	Печатать строку `s` в файл `fp`
`fflush(fp)`	Занести буферизованные данные выходного потока в файл `fp`
`fclose(fp)`	Закрыть файл `fp`
`fp=popen(s, mode)`	Открыть программный канал для команды `s` (см. `fopen`)
`pclose(fp)`	Закрыть программный канал `fp`
`system(s)`	Запустить команду `s` и ждать ее окончания

Таблица 6.4: Полезные стандартные функции ввода–вывода

and comprehention.

`diff` вырабатывает следующее:

```
$ diff filel file2
2c2
< a test
---
> not a test
4,6c4,5
< your
< skill
< and comprehention.
---
> our
> ability.
$
```

Диалог с `idiff` может выглядеть так:

```
$ idiff file1 file2
2c2                          Первое различие
< a test
```

```
---
> not a test
? >                          Пользователь выбрал вторую версию
4,6c4,5                      Второе различие
< your
< skill < and comprehension.
---
> our
> ability.
? <                           Пользователь выбрал первую (<) версию
idiff output in file idiff.out
$ cat idiff.out               Выходной поток направляется в этот файл
This is
not a test
of
your
skill
and comprehension.
$
```

Если вместо < или > выдан ответ *e*, `idiff` вызывает `ed` с двумя группами уже прочитанных строк. Если вторым был ответ *e*, буфер редактора выглядел бы следующим образом:

```
your
skill
and comprehension.
---
our
ability.
```

Все, что пишется редактором обратно в файл, идет в окончательный выходной поток.

И, наконец, любая команда может быть выполнена внутри `idiff` с помощью временного выхода посредством *!cmd*.

Технически самая трудная часть работы — `diff`, и она уже выполнена. Таким образом, в задачи `idiff` входит разбор выходного потока `diff`, открытие, закрытие, чтение и считывание соответствующих файлов в нужное время. Главная функция `idiff` поддерживает файлы и запускает процесс `diff`:

```
/* idiff:  interactive diff */

#include <stdio.h>
#include <ctype.h>
char    *progname;
#define HUGE    10000   /* large number of lines */

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. `Mktemp` переписывает свой аргумент: шесть символов `X` заменяются идентификатором процесса и буквой. Системный вызов `unlink(2)` удаляет поименованный файл из файловой системы.

Циклическая обработка изменений, о которых сообщает `diff`, выполняется функцией `idiff`. Основная идея достаточно проста: печатать порцию выходного потока `diff`, пропускать нежелательные данные в одном файле, а затем копировать требуемый вариант из другого файла. В программе есть много утомительных подробностей, так что она оказывается несколько больше, чем нам бы хотелось, но по частям ее довольно легко понять.

```
idiff(f1, f2, fin, fout)        /* process diffs */
    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; /* #lines from 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);  /* can fail on very long files */
    unlink(tempfile);
}
```

Функция parse выполняет рутинную, но тонкую работу по разбору строк, выдаваемых diff, извлекая четыре номера строки и команду (одну из a, c или d). При этом parse немного усложняется, так как diff может выдать либо один номер строки, либо два с той или другой стороны буквы команды:

```
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;
}
```

Макрокоманда a2i выполняет специальное преобразование из ASCII в целое в тех четырех местах, где она встречается.

Функции nskip и ncopy пропускают или копируют указанное число строк из файла:

```
nskip(fin, n)   /* skip n lines of file fin */
    FILE *fin;
{
    char buf[BUFSIZ];

    while (n-- > 0)
        fgets(buf, sizeof buf, fin);
}

ncopy(fin, n, fout)     /* copy n lines from fin to fout */
    FILE *fin, *fout;
{
    char buf[BUFSIZ];
```

```
    while (n-- > 0) {
        if (fgets(buf, sizeof buf, fin) == NULL)
            return;
        fputs(buf, fout);
    }
}
```

Программа `idiff`, если ее прервать, оставляет несколько файлов, хранящихся в `/tmp`. В следующей главе мы покажем, как перехватывать прерывания, чтобы убрать временные файлы, подобные использованным здесь.

Если критически подойти к `zap` и `idiff`, то оказывается, что самая трудная работа была уже кем-то сделана ранее. Эти программы только обеспечивают удобное взаимодействие с другой программой, которая обрабатывает нужную информацию. Всегда имеет смысл воспользоваться плодами чужих трудов — это позволяет повысить эффективность своей работы.

■ Упражнение: Добавьте команду q к `idiff`: ответ $q <$: автоматически выберет остаток от альтернатив '<'; $q >$ возьмет все оставшееся от альтернатив '>'.

■ Упражнение: Модифицируйте `idiff` так, чтобы некоторые аргументы `idiff` передавались к `diff`; `-b` и `-h` — вероятные кандидаты. Выполните еще одну модификацию `idiff`, позволяющую определять другой редактор, как в команде

```
$ idiff -e другой редактор file1 file2
```

Как взаимодействуют эти две модификации?

■ Упражнение: Измените `idiff`, чтобы использовать `popen` и `pclose` вместо временного файла для выходного потока `diff`. Как это скажется на сложности и скорости выполнения программы?

■ Упражнение: Если один из аргументов `diff` – -каталог, то в этом каталоге идет поиск файла с именем, заданным другим аргументом. Но если вы попробуете сделать то же самое с `idiff`, то она почему-то собъется. Объясните, что в данном случае происходит, и исправьте дефект.

6.9 Доступ к среде

Из Си-программы легко "добраться" до переменных в среде `shell`, что можно использовать для упрощения адаптации программы к окружению. Допустим, например, что размер экрана вашего терминала больше обычного (24-строкового). Чего вы сможете добиться, применив `p` и воспользовавшись преимуществами своего терминала? Необходимость определять размер экрана всякий раз, когда вы вводите `p`, надоедает:

```
$ p -36...
```

Вы могли бы всегда вставлять файл `shell` в свой `bin`:

```
$ cat /usr/you/bin/p
exec /usr/bin/p -36 $*
$
```

Третье решение — модифицировать `p`, чтобы использовать те переменные среды, которые определяют свойства вашего терминала. Предположим, что вы определили переменную `PAGESIZE` в своем `.profile`:

```
PAGESIZE=36
export PAGESIZE
```

Функция `getenv("var")` ищет в среде командную переменную `var` и возвращает ее значение как строку символов или `NULL`, если переменная не определена. При наличии `getenv` легко модифицировать `p`: достаточно лишь добавить пару описаний и вызов `getenv` к началу основной программы.

```
/* p:  print input in chunks (version 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++;
    }

...
```

Флаги обрабатываются вслед за переменной среды, так что любой явно заданный размер страницы в конце концов заменит неявно заданный.

■ Упражнение: Модифицируйте `idiff` так, чтобы она искала в среде имя редактора, который следует применить. Измените 2, 3 и т. д., чтобы использовать `PAGESIZE`.

Историческая и библиографическая справка. Стандартная библиотека ввода-вывода была разработана Д. Ритчи вслед за переносимой библиотекой ввода–вывода М. Леска. Оба пакета имели целью предоставить пользователю стандартные средства, чтобы можно было переносить программы без изменений с UNIX в другие системы.

Наша версия `p` основана на программе Г. Спенсера. Программа `adb` написана С. Борном, `sdb` — Г. Катцефом, а `lint` — С. Джонсоном.

Программа `idiff` в общих чертах построена на базе программы, первоначально написанной Дж. Маранзано. Сама `diff` — детище Д. МакИлроя и основана на алгоритме, созданном независимо Г. Стоуном и В. Хантом совместно с Т. Шиманским (Hunt J. W., Szymanski T. G. "A fast algorithm for computing longest common subseguences." CACM, May 1977.) Алгоритм "diff" описан в работе М. Д. МакИлроя и Д. В. Ханта "An algorithm for differential file comparison" (Bell Labs Computing Science Technical Report 41, 1976). В заключение приведем слова МакИлроя: "Я опробовал три различных алгоритма, прежде чем выбрал окончательный вариант. По сути `diff` позволяет не только хорошо понять программу, но и пересматривать ее до тех пор, пока она не станет правильной".

Глава 7

Системные вызовы в UNIX

В настоящей главе мы рассмотрим самый низкий уровень взаимодействия с операционной системой UNIX — системные вызовы. Они являются входами в ядро. Эти средства предоставляются операционной системой; все остальные средства построены на их основе.

Материал главы охватывает несколько важных областей. К ним относится прежде всего система ввода–вывода, являющаяся основной для функций типа `fopen` и `putc`. Речь пойдет также о файловой системе, в частности о каталогах и индексных дескрипторах. Затем мы обсудим процессы, т. е. как запускать задачи из программы, после чего поговорим о сигналах и прерываниях: что происходит, когда вы нажимаете клавишу *DELETE*, и как такую ситуацию должным образом обработать в программе.

Как и в гл. 6, во многих примерах приводятся полезные программы, не вошедшие в седьмую версию. Даже если они не помогут вам непосредственно, знакомство с ними может навести вас на мысль о сходных средствах, которые целесообразно создать и для вашей системы.

Подробное описание системных вызовов вы можете найти во втором разделе справочного руководства по UNIX, так как здесь освещаются лишь основные вопросы.

7.1 Ввод–вывод низкого уровня

Низкий уровень ввода–вывода представляет собой непосредственный вход в операционную систему. Ваши программы читают или пишут файлы порциями некоторого подходящего размера. Ядро буферизует данные порциями, которые подходят для периферийных устройств, и планирует операции для устройств так, чтобы их производительность была оптимальной для всех пользователей.

Дескрипторы файлов. Ввод и вывод обычно осуществляются посредством чтения или записи файлов, так как все периферийные устройства, и даже ваш терминал, представлены как файлы в файловой системе. Таким образом, единый интерфейс обеспечивает связь между программой и периферийными устройствами.

В самом общем случае, прежде чем читать или писать файл, необходимо сообщить системе о вашем намерении — открыть файл. Если вы собираетесь писать в файл, может быть, его необходимо и создать. Система проверяет ваше право сделать это (существует ли файл и есть ли у вас разрешение на доступ к нему?), и при положительном результате проверки возвращает неотрицательное целое, называемое дескриптором файла. Всякий раз, когда нужно выполнить ввод-вывод через файл, для его идентификации вместо имени используется дескриптор файла. Вся информация об открытом файле поддерживается системой. Ваша программа ссылается на файл только через дескриптор

файла. Указатель на FILE, как отмечалось в гл. 6, является ссылкой на структуру, которая наряду с прочим содержит дескриптор файла: макрокоманда fileno(fp), определенная в <stdio.h>., возвращает дескриптор файла.

Для обеспечения удобства ввода и вывода на терминал предусмотрены специальные меры. Когда программа запускается из shell, она получает три открытых файла с дескрипторами 0, 1 и 2 — стандартный входной поток, стандартный выходной поток и стандартный файл диагностики. Всем им по умолчанию поставлен в соответствие терминал, поэтому если программа читает только через дескриптор файла 0 и пишет через дескрипторы файлов 1 и 2, она может выполнять ввод-вывод без открывания файлов. Если же программа открывает любые другие файлы, они будут иметь дескрипторы 3, 4 и т. д.

При переключении ввода-вывода на файлы или программные каналы (к ним или от них) shell изменяет назначение терминала по умолчанию для дескрипторов файлов 0 и 1. Обычно дескриптор файла 2 закрепляется за терминалом, так что сообщения об ошибках могут поступать на него. Использование символики shell, такой, как 2> filename и 2>&1, вызовет переназначение файла, присвоенного по умолчанию, но при этом присвоение файлов меняется в shell, а не программой. (Программа сама может переназначить их впоследствии, если потребуется, что, правда, бывает редко.)

Файловый ввод-вывод: read и write. Весь ввод и вывод обеспечиваются двумя системными вызовами, read и write, которые доступны в Си с помощью функций с теми же именами. Для обеих первый аргумент — это дескриптор файла, второй — массив байтов, который служит источником данных или назначением, а третий — число байтов, которые следует передать.

```
int fd, n, nread, nwritten;
char buf [SIZE];

nread = read(fd, buf, n);
nwritten = write (fd, buf, n);
```

Каждый вызов возвращает число переданных байтов. При чтении возвращенное число может быть меньше, чем запрошенное, поскольку для чтения оставлено менее n байт. (Когда файлу поставлен в соответствие терминал, read обычно читает до следующей строки, что составляет меньшую часть запрошенного.) Возвращаемое значение 0 подразумевает конец файла, а значение -1 обозначает некоторую ошибку. При записи возвращаемое значение есть число действительно записанных байтов; если оно не равно числу байтов, которое предполагается записать, возникает ошибка. Несмотря на то, что число байтов, которые следует читать или писать, не ограничено, наиболее часто используются два значения: 1, что соответствует одному символу за одно обращение ("не буферизовано"), и размер блока на диске, как правило,- 512 или 1024 байта (такое значение имеет BUFSIZ в <stdio.h>). Для иллюстрации изложенного здесь приведена программа копирования входного потока в выходной. Так как входной и выходной потоки могут переключаться на любой файл или устройство, она действительно скопирует что-нибудь куда-либо: это “скелетная” реализация cat.

```
/* cat:  minimal version */
#define SIZE    512     /* arbitrary */

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 числу `SIZE`, некоторый вызов `read` вернет меньшее число байтов, которые должны быть записаны с помощью `write`; следующий затем вызов `read` вернет нуль.

Чтение и запись порциями, подходящими для диска, будут наиболее эффективными, но даже ввод-вывод по одному символу за раз осуществим для умеренных объемов данных, так как данные буферизуются ядром. Дороже всего обходятся обращения к системе. Программа `ed`, например, использует однобайтовый способ, чтобы читать стандартный входной поток. Мы хронометрировали работу данной версии `cat` для файла в 54 000 байт при шести значениях `SIZE`:

Время (пользователь +система, в сек.)

Размер	PDP-11/40	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 на диске для системы на **PDP-11** составляет 512 байт и 1024 байта — для **VAX**.

Доступ нескольких процессоров к одному и тому же файлу в одно и то же время является совершенно законным: в самом деле, один процесс может писать, в то время как другой читает. Это обескураживает, если не входит в ваши планы, но иногда оказывается полезным. Даже несмотря на то, что при одном обращении к функции `read` возвращается 0, который сигнализирует о конце файла, следующий вызов `read` обнаружит наличие некоторого количества байтов, если еще данные пишутся в файл. Это соображение лежит в основе программы `readslow`, которая продолжает читать свой входной поток вне зависимости от того, достигла она конца файла или нет. Программа `readslow` удобна для наблюдения за работой программы.

```
$ slowprog >temp &
5213                                    идентификатор процесса
$ readslow <temp | grep something
```

Иными словами, медленная программа выполняет вывод в файл; `readslow`, возможно, совместно с некоторыми другими программами "наблюдает", как накапливаются данные.

По составу `readslow` идентична `cat`, за исключением того, что она зацикливается, а не завершается, когда встречает конец входного потока. Программа `readslow` должна использовать ввод-вывод низкого уровня, так как стандартная библиотечная функция продолжает выдавать `EOF` после первого конца файла.

```
/* readslow:  keep reading, waiting for more */
#define SIZE    512     /* arbitrary */

main()
{
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

Функция sleep заставляет программу остановиться на определенное число секунд (см. справочное руководство по sleep(3)). Мы не хотим, чтобы программа долго занималась поиском дополнительных данных, так как на это расходуется время центрального процессора. Таким образом, наша версия `readslow` копирует свой входной поток до конца файла, "спит" какое-то время, затем снова возобновляет работу. Если пока она была "в паузе", пришли еще данные, они будут прочитаны следующим read.

■ Упражнение: Добавьте `readslow` аргумент `n`, так что установленное по умолчанию время паузы может быть изменено на `n` секунд. Некоторые системы обеспечивают флаг `-f` ("навсегда") для `tail`, которая объединяет функции `tail` и `readslow`. Прокомментируйте этот вариант.

■ Упражнение: Что происходит с `readslow`, если читаемый файл обрывается? Как бы вы исправили ситуацию? Подсказка: читайте о `fstat` в разд. 7.3.

Создание файла: open, creat, close, unlink. Все стандартные файлы, кроме установленных по умолчанию, — входной, выходной и файл диагностики вы должны явно открыть для чтения или записи. Это можно сделать с помощью двух системных вызовов — открыть и создать (Однажды К. Томпсона спросили, что бы он хотел изменить, если бы ему пришлось заново конструировать систему UNIX. Он ответил: "Я бы написал creat с e.").

Функция open весьма похожа на fopen из предыдущей главы, за исключением того, что вместо указателя файла она возвращает дескриптор файла, имеющий тип int.

```
char *name;
int fd, rwmode;
fd = open (name, rwmode);
```

Как и для fopen, аргумент name есть символьная строка, содержащая имя файла. Аргумент вида доступа, однако, другой: rwmode равен 0 для чтения, 1 — для записи, 2 — в том случае, когда нужно открыть файл для чтения и записи. При вызове open возвращается -1, если возникает какая-либо ошибка; иначе возвращается корректный дескриптор файла.

Попытка открыть несуществующий файл является ошибкой. Системный вызов creat позволяет создать новые файлы или переписать старые.

```
int perms;
fd = creat (name, perms);
```

Вызов creat возвращает дескриптор файла, если можно создать файл name, и -1 в противном случае. Если файл не существует, creat создает его с правами доступа, определяемыми аргументом perms. Существующий файл creat сокращает до нулевой длины, т. е. применение creat к уже существующему файлу не является ошибкой (права доступа при этом не изменяются). Безотносительно к правам доступа файл, к которому было обращение creat, открыт для записи.

Как известно из второй главы, с файлом связаны девять битов информации о защите, контролирующих разрешение на чтение, запись или выполнение, так что число из трех восьмиричных цифр удобно для спецификации этой информации. Например, 0755 дает разрешение владельцу файла читать, писать и выполнять его, а чтение и выполнение файла доступно любому пользователю. Не забывайте о первом нуле, который определяет восьмиричные числа в языке Си.

Иллюстрацией изложенного может служить упрощенная версия cp. Ее главный недостаток состоит в том, что она копирует только один файл и не разрешает использовать в качестве второго аргумента каталог. Кроме того, наша версия не сохраняет права доступа файла-источника; в дальнейшем мы покажем, как это исправить.

```
/* cp:  minimal version */
#include <stdio.h>
#define PERMS 0644 /* RW for owner, R for group, others */
char *progname;

main(argc, argv)        /* cp: copy f1 to 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);
}
```

error мы обсудим ниже.

Число файлов, которые одновременно могут быть открыты программой, ограничено (обычно порядка 20; см. NOFILE в <SYS/param.h>). Поэтому любая программа, которой предстоит обрабатывать много файлов, должна быть готова неоднократно использовать одни и те же дескрипторы файлов. Системный вызов close разрывает связь между именем и дескриптором файла, освобождая дескриптор для использования с некоторым другим файлом. Завершение программы посредством exit и возврат из основной программы закрывают все открытые файлы. Вызов системы unlink удаляет файл из файловой системы.

Обработка ошибок: errno. Обсуждаемые здесь системные вызовы, а по сути все системные вызовы, могут вызывать ошибки. Обычно они сигнализируют об ошибке, возвращая значение -1. Иногда полезно знать, какая именно ошибка произошла, поэтому системные вызовы, когда это приемлемо, оставляют номер ошибки во внешней целой переменной, называемой errno. (Значение различных номеров ошибок объясняется во введении к разд. 2 справочного руководства по UNIX.) С помощью errno ваша программа может определить, например, чем вызвана неудача при открытии файла — тем, что он не существует, или тем, что у вас нет разрешения на его чтение. Кроме того, есть массив символьных строк sys_errlist, индексируемый errno, который переводит число в строку, передающую смысл ошибки. Наша версия error использует эти структуры данных:

```
error(s1, s2)   /* print error message and die */
    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);
}
```

Errno первоначально равна нулю и всегда должна быть меньше, чем sys_herr. Она не становится нулевой вновь при нормальной работе, поэтому вы должны обнулять ее после каждой ошибки, если ваша программа будет продолжать выполняться.

Сообщения об ошибках в нашей версии cp появляются следующим образом:

```
$ cp foo bar
cp: can't open foo (Нет такого файла или каталога)
$ date >foo; chmod 0 foo                          Создать нечитаемый файл
$ cp too bar
cp: can't open foo (В разрешении отказано)
$
```

Произвольный доступ: lseek. Файл ввода-вывода обычно последовательный: каждый read или write занимает место в файле непосредственно после использованного при предыдущем вызове. Однако при необходимости файл может быть прочитан или записан

в произвольном порядке. Системный вызов lseek позволяет перемещаться по файлу, не осуществляя ни чтения, ни записи:

```
int fd, origin;
long offset, pos, lseek();

pos = lseek(fd, offset, origin);
```

Текущая позиция в файле с дескриптором fd перемещается к позиции offset, которая отсчитывается относительно места, определяемого origin. Последующие процессы чтения или записи будут начинаться с этой позиции. Origin может иметь значения 0, 1, 2, задавая тем самым начало отсчета значения offset — от начала, от текущей позиции или от конца файла соответственно.

Возвращаемое значение есть новая абсолютная позиция или -1 при ошибке. Например, при добавлении информации в файл нужно дойти до его конца, а затем выполнить запись:

```
lseek(fd, 0L, 2);
```

Чтобы вернуться обратно к началу ("перемотать"), необходимо вызвать

```
lseek(fd, 0L, 0);
```

Для определения текущей позиции следует выполнить

```
pos = lseek(fd, 0L, 1);
```

Обратите внимание на аргумент 0L: смещение есть длинное целое. ('l' в lseek означает 'long' — длинный, чтобы отличить его от системного вызова seek в шестой версии, где используются короткие целые.)

С помощью lseek можно обращаться с файлами как с большими массивами, однако при этом время доступа к ним возрастает. Например, следующая функция читает любое число байтов из любого места в файле:

```
get(fd, pos, buf, n) /* read n bytes from position pos */
    int fd, n;
    long pos;
    char *buf;
{
    if (lseek(fd, pos, 0) == -1)    /* get to pos */
        return -1;
    else
        return read(fd, buf, n);
}
```

■ Упражнение: Модифицируйте `readslow` так, чтобы обрабатывать имя файла в качестве аргумента, если оно присутствует. Добавьте `-e`:

```
$ readslow -e
```

заставляет `readslow` искать конец входного потока, прежде чем начать чтение. Каковы функции `lseek` при работе с программным каналом?

■ Упражнение: Перепишите `efopen` из гл. 6, чтобы вызвать `error`.

7.2 Файловая система: каталоги

Наша следующая тема — как ориентироваться в иерархии каталогов. При этом мы будем использовать не новые системные вызовы, а лишь несколько старых в новом контексте. В качестве примера приведем функцию spname, которая пытается справиться с неверно написанными именами файлов. Функция

```
n = spname (name, newname);
```

ищет файл с именем, "достаточно близким" к name. Если такое имя найдено, оно копируется в newname. Значение n, возвращаемое spname, равно -1, если ничего достаточно близкого не найдено, 0 — при точном совпадении и 1, если была сделана коррекция.

Spname является удобным дополнением к команде p: если вы пытаетесь печатать файл, но неверно написали имя, p спросит вас, не имели ли вы в виду что-либо другое:

```
$ p /urs/srx/ccmd/p/spnam.c          Очень плохое имя
"/usr/src/cmd/p/spname.c"? y         Предложенная коррекция принята
/* spname: возвращает верно написанное имя файла */

...
```

Пока мы пишем имя файла, spname пытается исправить каждую его составную часть, в которой несовпавшая буква была опущена, оказалась лишней, просто неверна или поменялась местами с другой буквой. Это удобное средство рассчитано на того, кто печатает не очень внимательно.

Прежде чем писать программу, уместно сделать короткий обзор структуры файловой системы. Каталог представляет собой файл, содержащий список имен файлов и указание, где они размещены. Место размещения определяется индексом в так называемой индексной таблице файлов. В записи индексной таблицы содержится вся информация о файле, кроме его имени. Строка каталога, таким образом, состоит из двух элементов — индекса файла и его имени. Точное описание можно найти в файле <sys/dir.h>:

```
$cat /usr/include/sys/dir.h
#define DIRSIZ 14 /'"максимальная длина имени файла*/
struct direct /* структура строки каталога */
{
    ino_t d_ino; /* номер индексного дескриптора */
    char d_name [DIRSIZ]; /* имя файла */
};
$
```

"Тип" `ino_t` это typedef, описывающий индекс в индексной таблице. Он является коротким целым без знака (unsigned short) в версиях системы для PDP-11 и VAX и не должен включаться в программу, так как может быть иным на другой машине. Поэтому мы воспользуемся определением типа typedef. Полный набор "системных" типов находится в <sys/types.h>, который должен быть включен до <sys/dir.h>.

Действия spname достаточно прямолинейны, хотя и требуют выполнения нескольких граничных условий. Предположим, что имя файла /d1/d2/f. Основная идея состоит в следующем: отделить первую компоненту (/), найти в каталоге имя, близкое к следующей компоненте (d1), затем найти имя, близкое к d2, и т. д. до тех пор, пока не будет достигнуто полное совпадение для каждой составной части. Если на какой-то стадии в каталоге не окажется подходящего кандидата, поиск прекратится.

Мы разбили процесс на три функции. Сама spname выделяет компоненты пути и составляет из них имя файла, наилучшим образом совпадающее с исходным. Функция mindist ищет в данном каталоге файл с именем, ближайшим к составленному функцией spname. Функция spdist вычисляет "расстояние" между двумя именами.

```
/* spname:  return correctly spelled filename */
/*
 * 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 == '/')     /* skip slashes */
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0')       /* exact or corrected */
            return strcmp(oldname,newname) != 0;
        p = guess;      /* copy next component into guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZ)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
            return -1;      /* hopeless */
        for (p = best; *new = *p++; ) /* add to end */
```

```
            new++;                                      /* of newname */
    }
}

mindist(dir, guess, best)       /* search dir for guess */
    char *dir, *guess, *best;
{
    /* set best, return distance 0..3 */
    int d, nd, fd;
    struct {
        ino_t ino;
        char  name[DIRSIZ+1];   /* 1 more than in dir.h */
    } nbuf;

    nbuf.name[DIRSIZ] = '\0';       /* +1 for terminal '\0' */
    if (dir[0] == '\0')         /* current directory */
        dir = ".";
    d = 3;  /* minimum distance */
    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)         /* exact match */
                    break;
            }
        }
    close(fd);
    return d;
}
```

Если имя каталога, данное mindist, пустое, отыскивается '.'. Функция mindist читает одну строку каталога за один раз. Отметим, что буфер для read представляет собой структуру, а не массив символов. Мы используем sizeof, чтобы вычислить число байтов и привести адрес к символьному указателю.

Если строка каталога в данный момент не используется (поскольку файл удален), то поле индекса в ней равно нулю и она пропускается. Проверка расстояния осуществляется как

```
if (nd <= d...)
```

а не как

```
if (nd < d...)
```

поэтому любой одиночный символ дает лучшее совпадение, чем имя '.', которое всегда является первой строкой в каталоге.

```
/* spdist:  return distance between two names */
/*
 *      very rough spelling metric:
 *      0 if the strings are identical
 *      1 if two chars are transposed
 *      2 if one char wrong, added or deleted
 *      3 otherwise
 */

#define EQ(s,t) (strcmp(s,t) == 0)

spdist(s, t)
    char *s, *t;
{
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;           /* exact match */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1]
              && *t == s[1] && EQ(s+2, t+2))
                return 1;       /* transposition */
            if (EQ(s+1, t+1))
                return 2;       /* 1 char mismatch */
        }
        if (EQ(s+1, t))
            return 2;           /* extra character */
    }
    if (*t && EQ(s, t+1))
        return 2;               /* missing character */
    return 3;
}
```

Поскольку у нас есть spname, несложно вставить функции по коррекции написания в p:

```
/* p:  print input in chunks (version 4) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

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:    /* no match possible */
                fp = efopen(argv[i], "r");
                break;
            case 1:     /* corrected */
                fprintf(stderr, "\"%s\"? ", buf);
                if (ttyin() == 'n')
                    break;
                argv[i] = buf;
                /* fall through... */
            case 0: /* exact match */
                fp = efopen(argv[i], "r");
                print(fp, pagesize);
                fclose(fp);
            }
    exit(0);
}
```

Функции по коррекции написания не следует слепо применять к каждой программе, которая использует имена файлов. Они хорошо сочетаются с p, так как p — диалоговая программа, но подходят и для недиалоговых программ.

■ Упражнение: Насколько вы можете улучшить эвристику для выявления наилучшего совпадения в `spname`? Например, неразумно рассматривать регулярный файл так, как если бы он был каталогом; текущая версия это допускает.

■ Упражнение: Имя `tx` совпадает с каким-либо именем `tc`, которое оказывается последним в каталоге для любого одиночного символа `c`. Можете ли вы придумать лучшую меру расстояния? Реализуйте ее и посмотрите, насколько хорошо эта конструкция работает с реальными пользователями.

■ Упражнение: Работает ли `p` ощутимо быстрее, если чтение каталога выполняется большими порциями?

■ Упражнение: Модифицируйте `spname`, чтобы возвращать имя, которое является префиксом желаемого имени, если нельзя найти более точного совпадения. Как следует разрешить ситуацию, если несколько имен совпадают с префиксом?

■ Упражнение: Какую пользу могли бы извлечь другие программы из `spname`? Сконструируйте отдельную программу, которая корректировала бы другие аргументы прежде, чем передать их другой программе, как в

```
$ fix prog filenames...
```

Можете написать версию `cd`, которая использует `spname`? Как бы вы ее встроили?

7.3 Файловая система: индексные дескрипторы

Теперь мы обсудим системные вызовы применительно к файловой системе, в частности к такой информации о файлах, как размеры, даты изменений, права доступа и т. д. Эти системные вызовы позволяют получить полностью всю' информацию, о которой упоминалось во второй главе.

Для начала разберемся в самом индексном дескрипторе. Часть индексного дескриптора описывается структурой stat, определенной в <sys/stat.h>:

```
struct stat /* структура, возвращаемая stat */
{
    dev_t st_dev;    /* устройство, содержащее файл */
    ino_t st_ino;    /* индекс */
    short st-mod;    /* биты режима */
    short st_nlink;  /* число связей файла */
    short st_uid;    /* пользовательский идентификатор владельца */
    short st_gid;    /* идентификатор группы владельцев */
    dev_t st_rdev;   /* для специальных файлов */
    off_t st_size;   /* размер файла в символах */
    time_t st_atime; /* время последнего чтения файла */
    time_t st_mtime; /* время последней записи или создания файла */
    time_t st_ctime; /* время последнего изменения индексного дескриптора
                        или файла */
}
```

Большинство полей поясняются примечаниями. Типы вроде `dev_t` и `ino_t` определены в <sys/types.h>, как отмечено выше. Поле `st_mode` содержит множество флагов, описывающих файл; для удобства определения флагов также являются частью файла <sys/stat.h>:

```
#define S_IFMT   0170000 /* тип файла */
#define S_IFDIR  0040000 /* каталог */
#define S_IFCHR  0020000 /* байт-ориентированный */
#define S_IFBLK  0060000 /* блок-ориентированный */
#define S_IFREG  0100000 /* регулярный */
#define S_ISU1D  0004000 /* установка идентификатора пользователя при
                            выполнении */
#define S_ISGID  0002000 /* установка идентификатора группы при выполнении */
#define S_ISVTX  0001000 /* сохранить выгруженный текст даже после
                            использования */
#define S_IREAD  0000400 /* разрешение читать, владелец */
#define S_IWRITE 0000200 /* разрешение писать, владелец */
#define S_IEXEC  0000100 /* разрешение на выполнение/поиск, владелец */
```

Индексный дескриптор для файла доступен двум системным вызовам stat и fstat. При вызове stat параметром является имя файла, а результатом — информация из индексного дескриптора для этого файла (или — 1 при наличии ошибки). Fstat выполняет те же функции в отношении дескриптора открытого файла (не в отношении указателя на FILE). Иными словами,

```
char *name;
int fd;
struct stat stbuf;

stat(name, &stbuf);
fstat(fd, &stbuf);
```

заполняет структуру `stbuf` информацией из индексного дескриптора для имени файла или дескриптора файла `fd`.

Зная все это, мы можем приступить к написанию некоторой полезной программы. Начнем с версии checkmail — программы на Си, которая следит за содержимым вашего почтового ящика. Если файл увеличивается, checkmail выдает сообщение: “У вас есть корреспонденция” и включает звонок. (При уменьшении файла, видимо, из-за того, что вы успели прочитать и сбросить некоторую почтовую корреспонденцию, сообщение не требуется.) Для первого шага вы сделали вполне достаточно, а когда ваша программа заработает, вы станете знатоком.

```
/* checkmail:  watch user's mailbox */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;
char *maildir = "/usr/spool/mail";      /* system dependent */

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)     /* no mailbox */
            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 ваше регистрационное имя или NULL, если это невозможно, checkmail переходит к почтовому каталогу с помощью системного вызова chdir, так что последующие вызовы stat не должны будут "добираться" до почтового каталога через все каталоги, начиная от корневого. Возможно, вы должны адаптировать maildir для своей системы. Мы написали checkmail так, чтобы она работала, даже если нет почтового ящика, поскольку большинство версий mail убирают почтовый ящик в том случае, когда он пуст.

Мы приводили эту программу в гл. 5 для иллюстрации циклов `shell`. Всякий раз при проверке почтового ящика она создает несколько процессов и загружает систему больше, чем хотелось бы. Версия на Си — единственный процесс, который выполняет stat для файла каждую минуту. Сколько времени требуется на то, чтобы checkmall постоянно выполнялась как фоновая задача? Как показали наши измерения, это время составляет меньше секунды в час, так что им вполне можно пренебречь.

sv: иллюстрация обработки ошибок. Следующей мы собираемся написать похожую на cp программу sv, которая будет копировать множество файлов в каталог, заменяя каждый файл лишь в том случае, если его нет в каталоге или он "старше" копируемого с тем же именем (имя sv означает "сохранять") . Суть действия программы состоит в том, что она не переписывает новую версию файла, sv использует больше информации из индексного дескриптора, чем checkmail.

Вызов sv будет иметь такую конструкцию:

```
$ sv file1 file2 ... dir
```

Она копирует file1 в dir/file1, file2 в dir/file2 и т. д., если только целевой файл не новее, чем файл-источник; в этой ситуации копирование не происходит и печатается Соответствующее предупреждение. Во избежание создания большого числа копий или связанных файлов sv не допускает применения символов '/' в любом исходном имени файла.

```
/* sv:  save new files */
#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, хранящиеся в индексных дескрипторах, исчисляются в секундах (за начало отсчета принято время 0:00 по Гринвичу, 1 января 1970 г.), так что более старые файлы имеют меньшие значения в поле `st_mtime`.

```
sv(file, dir)   /* save file in 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() in some systems */
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1)   /* target not present */
        sto.st_mtime = 0;       /* so make it look old */
    if (sti.st_mtime < sto.st_mtime)    /* target is newer */
        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);
}
```

Мы заменили стандартные функции ввода-вывода функцией creat, так что sv может сохранять режим работы входного файла. (Заметьте, что index и strchr — разные имена одной и той же процедуры; посмотрите в справочном руководстве по string(3), какое имя использует ваша система.)

Хотя программа sv довольно специфична, в ней отражены некоторые важные идеи. Многие программы не являются системными, но тем не менее используют информацию, поддерживаемую операционной системой и доступную через системные вызовы. Для таких программ существенно, что представление информации хранится только в стандартных файлах макроопределений типа <stat.h> и <dir.h> и что в программы включены эти файлы вместо действительных описаний. Подобные программы с большей степенью вероятности переносимы с одной системы на другую.

Отметим, что по крайней мере две трети кода sv составляет контроль ошибок. На ранних этапах написания программы было искушение сэкономить на обработке ошибок, поскольку это отвлекает от основной задачи. Когда же программа уже работает, трудно решиться на то, чтобы вернуться назад и вставить в нее процедуры проверки, которые превращают специальную программу в унифицированную.

Программа sv не защищена от возможных сбоев. Она, например, не обрабатывает прерывания в неподходящие моменты, но более аккуратна, чем большинство других программ. Хотелось бы обратить ваше внимание на финальный оператор write. Программа редко сбивается на этом операторе, поэтому многие программы игнорируют такую возможность. Однако переполнение дискового пространства, неполадки в линии связи или иные нарушения могут вызвать ошибки в write, и вы гораздо лучше справитесь с ними, если программа выдает вам соответствующее сообщение.

Дело в том, что контроль ошибок весьма утомителен, но тем не менее важен. Из-за ограниченного объема книги и обширности излагаемого в ней материала мы не уделяли должного внимания этому вопросу. Но в настоящих, "производственных" программах не следует позволять себе игнорировать ошибки.

■ Упражнение: Модифицируйте `checkmail` так, чтобы идентифицировать посылающего сообщение: `"У вас есть почта"`. Подсказка: `sscanf`, `lseek`.

■ Упражнение: Модифицируйте `checkmail` так, чтобы она не переходила к каталогу почты перед входом в цикл. Окажет ли это ощутимое влияние на ее производительность? Более трудный вопрос: можете ли вы написать версию `checkmail`, которая обходится только одним процессом для оповещения всех пользователей?

■ Упражнение: Напишите программу `watchfile`, которая управляет файлом и печатает его с начала всякий раз, как он изменится. Когда бы вы ее использовали?

■ Упражнение: Программа `sv` очень "прямолинейна" при обработке ошибок. Модифицируйте ее так, чтобы она продолжала выполняться, даже если не удается обработать некоторый файл.

■ Упражнение: Сделайте `sv` рекурсивной: если один из исходных файлов–каталог, то этот каталог и все его файлы обрабатываются таким же образом. Сделайте рекурсивной `cp`. Подумайте, следует ли `cp` и `sv` объединить в одну программу, чтобы `cp -v` не создавала копию, если целевой файл новее файла–источника.

■ Упражнение: Напишите программу `random`.

```
$ random filename
```

должна выдавать одну строку, произвольно выбранную из файла. Если есть файл `people`, содержащий имена, `random` можно использовать в программе `scapegoat` ("козел отпущения"), полезной при случайном определении виновных:

```
$ cat scapegoat
echo "В этом виноват ‘random people‘!"
$ scapegoat
В этом виноват Кен!
$
```

Убедитесь в том, что `random` хороша независимо от распределения длины строк.

■ Упражнение: Помимо прочего в индексном дескрипторе указаны адреса размещения блоков файла на диске. Рассмотрите файл `<sys/into.h>`, а затем напишите программу `icat`, которая должна читать файлы, описываемые номером записи каталога и устройством диска. (Она, конечно, будет работать только в том случае, если требуемый диск открыт на чтение.) При каких обстоятельствах `icat` полезна?

7.4 Процессы

В этом разделе мы покажем вам, как выполнить одну программу, вызвав ее из другой. Самый легкий путь — привлечь стандартную библиотечную программу system, упомянутую, но забракованную в гл. 6. Программа `system` использует один аргумент — командную строку в том виде, в каком она вводится с терминала (за исключением символа перевода строки), и выполняет ее порожденным `shell`. Если командная строка должна быть создана из кусочков, можно прибегнуть к форматированию памяти прог-

раммой sprintf. В конце раздела мы рассмотрим более безопасную версию system для работы с диалоговыми программами, но прежде чем изучать программу в целом, обсудим структуры, из которых она составляется.

Создание процесса низкого уровня: execlp и execvp. Самая важная операция - выполнение другой программы без возврата с помощью системного вызова execlp. Например, чтобы напечатать дату и выполнить тем самым последнее действие запущенной программы, используют

```
execlp ("дата", "дата", (char*) 0);
```

Первый аргумент execlp есть имя файла команды; execlp выбирает путь поиска (т. е. $PATH) из вашего окружения и выполняет такой же поиск, как shell. Второй и последующие аргументы — это имена и аргументы команд; для новой программы они становятся массивом argv. Конец списка отмечен аргументом 0. (См. справочное руководство по exec(2), и вы поймете конструкцию execlp.)

Вызов execlp перекрывает существующую программу новой, запускает ее и затем завершается. Первоначальная программа получает управление обратно только при возникновении ошибки, например, когда файл не удается найти или он является невыполнимым:

```
execlp ("дата", "дата", (char*)0);
fprintf(stderr, "Не удалось выполнить 'дата'\n"); exit(1);
```

Если число аргументов вам заранее не известно, полезно применить execvp (вариант execlp). Вызов выглядит так:

```
execvp (filename, argp);
```

где argp означает массив указателей к аргументам (таким, как argv). Последним в массиве должен быть указатель NULL, так что execvp может отметить конец списка. Как и для execlp, filename — это файл, в котором находится программа, argp — массив argv для новой программы, а argp[0] — имя программы.

Ни одна из перечисленных выше программ не обеспечивает расширения в списке аргументов метасимволов типа <, >, *, кавычки и т. п. Если они вам нужны, воспользуйтесь execlp и вызовите /bin/sh из shell, которая выполнит эту работу. Сконструируйте строку commandline, содержащую полную команду, как если бы она была напечатана на терминале, например:

```
execlp ("/bin/sh/", "sh", "-c", commandline, (char*) 0);
```

Аргумент -c предписывает трактовать следующий аргумент как целую командную строку.

В качестве иллюстрации exec рассмотрим программу waitfile. Команда

```
$ waitfile filename [command]
```

периодически проверяет поименованный файл. Если он не менялся после последней проверки, выполняется command. В том случае, когда команда не указана, файл копируется в стандартный выходной поток. С помощью waitfile мы контролируем работу troff, как в

```
$ waitfile troff .out echo troff done &
```

Программа waitfile использует fstat, чтобы выявить время последнего изменения файла.

```
/* waitfile:  wait until file stops changing */
#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) {    /* copy file */
        execlp("cat", "cat", argv[1], (char *) 0);
        error("can't execute cat %s", argv[1]);
    } else {            /* run process */
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    exit(0);
}
```

Мы рассмотрели пример работы как execlp, так и execvp. Эта программа выбрана в качестве иллюстрации, поскольку она весьма полезна, но возможны и другие варианты. Так, waitfile могла бы просто завершиться по окончании изменения файла.

■ Упражнение: Модифицируйте `watchfile` (упр. 7.12) так, чтобы она имела то же свойство, что и `waitfile`: в отсутствие command копируется файл, в противном случае выполняется команда. Могли бы `watchfile` и `waitfile` разделять исходную программу? Подсказка: `argv[0]`.

Управление процессами: fork и wait. Следующий шаг — вновь получить управление после запуска программы с помощью execlp и execvp. Так как эти программы просто "перекрывают" старую программу новой, для сохранения старой требуется сначала разбить ее на две копии. Одна из копий может быть перекрыта, в то время как другая ждет новую, перекрывающую ее программу, чтобы завершиться. Разбиение выполняется с помощью системного вызова fork:

```
proc_id = fork();
```

Программа разбивается на две копии, каждая из которых продолжает работать. Они отличаются лишь значением, возвращаемым fork, — номером процесса process-id. В первом процессе (потомке) `proc_id` равен нулю, во втором (родительском) `proc_id` есть номер процесса-потомка. Итак, вызвать другую программу и вернуться можно следующим образом:

```
if (fork() == 0)
    execlp ("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

Фактически этого достаточно, за исключением обработки ошибок. Fork делает две копии программы. В процессе-потомке fork возвращает нуль, так что он вызывает execlp, которая выполняет commandline и затем завершается. В родительском процессе fork возвращает не нуль, поэтому execlp пропускается. (При наличии ошибки fork возвращает -1.)

Чаще родительский процесс ожидает, пока потомок закончит работу, прежде чем продолжить свое выполнение, для чего используется системный вызов wait:

```
int status;

if(fork() == 0)
    execlp(...);  /* потомок */
wait(&status);     /* родитель */
```

Однако при этом не контролируются ошибки, такие, как сбои execlp и fork, или возможность одновременной работы нескольких процессов-потомков (wait возвращает номер завершившегося процесса-потомка, если вы хотите сравнить его со значением, возвращенным fork). Тем не менее эти три строки являются сердцевиной стандартной функции system.

Значение status, возвращаемое wait, содержит в своих младших восьми разрядах системное представление кода завершения процесса-потомка; оно равно нулю при нормальном завершении и не равно нулю при разного рода затруднениях. Следующие старшие восемь битов берутся из аргумента вызова exit или возвращаются из main, которая вызывает окончание выполнения процесса-потомка.

Если программа вызывается из `shell`, три дескриптора файла, 0, 1 и 2, ссылаются на соответствующие файлы, и все остальные дескрипторы доступны для использования. Когда эта программа вызывает другую, в соответствии с профессиональной этикой указанные условия должны быть соблюдены. Ни fork, ни exec не влияют никоим образом на открытые файлы; оба процесса, родитель и потомок, имеют одни и те же открытые файлы. Если процесс-родитель буферизует выходной поток, который необходимо вывести до процесса-потомка, родитель должен очистить свой буфер ранее execlp. И, наоборот,

при буферизации родителем входного потока потомок потеряет информацию, которая читалась родителем. Выходной поток может быть выведен, но входной нельзя "положить назад". Обе ситуации являются следствием реализации входного или выходного потока стандартной библиотекой ввода-вывода, обсуждавшейся в гл. 6, поскольку при этом и ввод, и вывод буферизуются обычным образом.

Именно свойство наследования дескрипторов файлов через execlp используется в system: если у вызывающей программы стандартные входной и выходной потоки не связаны с терминалом, то этим же свойством обладает команда, вызванная из system. Возможно, такой вариант нам и нужен. В списке команд редактора `ed`, например, входной поток команды, начинающейся с символа ! , вероятно, должен поступить из того же списка. Даже тогда `ed` должен считывать из своего входного потока по одному символу во избежание возникновения проблем буферизации ввода.

Для диалоговых программ, подобных p, system должна тем не менее вновь связать стандартный входной и выходной потоки с терминалом, в частности /dev/tty.

Системный вызов dup(fd) дублирует дескриптор файла fd на незанятый дескриптор файла с наименьшим номером и возвращает новый дескриптор, ссылающийся на тот же самый открытый файл. Следующая программа "присоединяет" стандартный входной поток программы к файлу:

```
int fd;

fd = open("file", 0);
close(0);
dup(fd);
close(fd);
```

Вызов close(fd) освобождает дескриптор файла 0 (стандартный входной поток), но, как правило, не влияет на процесс-родитель. Здесь приведена наша версия system для диалоговых программ, использующая progname для вывода сообщений об ошибках. Вам следует игнорировать те части функции, которые имеют дело с сигналами (мы вернемся к ним позднее).

```
/*
 * Safer version of system for interactive programs
 */
#include <signal.h>
#include <stdio.h>

system(s)       /* run command line 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;
}
```

Отметим, что /dev/tty открыта с режимом 2 — чтение и запись. С помощью dup формируются стандартный входной и выходной потоки. Здесь можно провести аналогию со сборкой системой стандартных входного и выходного потоков и потока ошибок, когда вы в нее входите. Поэтому в ваш стандартный входной поток можно писать:

```
$ echo hello 1>&0
hello
$
```

Это означает, что вам следует применить dup к дескриптору файла 2, чтобы вновь связать стандартные ввод и вывод, но открытие /dev/tty является более естественным и безопасным. Даже system имеет потенциальные проблемы: открытые файлы в вызывающей программе, такие, как tty в подпрограмме ttin программы p, будут передаваться процессу-потомку.

Смысл изложенного выше состоит не в том, что вы должны использовать нашу версию system для своих программ (она могла бы разрушить недиалоговый `ed`, например), а в том, чтобы понять, как управляют процессами и корректно используют примитивы; значение слова “корректно” меняется в зависимости от приложения и может быть не согласовано со стандартной реализацией system.

7.5 Сигналы и прерывания

Теперь мы рассмотрим работу с сигналами извне (такими, как прерывания) и ошибками программы. Последние возникают главным образом из-за некорректных обращений к памяти, выполнения привилегированных команд или при выполнении операций с плавающей запятой. Наиболее распространенными внешними сигналами являются прерывание,

посылаемый при печати символа del, выйти, генерируемый символом FS (*ctrl-*), отбой, вызываемый завершением телефонной связи, и закончить, генерируемый командой kill. Когда происходит одно из этих событий, посылается сигнал всем процессам, запущенным с того же терминала, и если не были приняты другие меры, процесс завершается. Для большинства сигналов пишется файл образа памяти, который может потребоваться при поиске ошибок (см. справочное руководство по adb(1), sdb(1)).

Системный вызов signal изменяет действие, заданное по умолчанию. Он имеет два аргумента: номер, определяющий сигнал, и адрес функции или код, предписывающий игнорировать сигнал либо запустить процедуру, принятую по умолчанию. Файл <signal.h> содержит определения для различных аргументов. Так,

```
#include <signal.h>

...

signal(SIGINT, SIG_IGN);
```

Специфицирует игнорирование прерываний, тогда как

```
signal(SIGINT, SIG_DEL);
```

восстанавливает действие по умолчанию, означающее завершение процесса. В любом случае signal возвращает предыдущее значение сигнала. Если второй аргумент signal представляет собой имя функции, которая уже должна быть описана в том же самом исходном файле, то функция будет вызвана, когда возникнет сигнал. Это практикуется довольно часто, чтобы программа могла "подчищать" неоконченные работы перед своим завершением, например удалять временный файл:

```
#include <signal.h>

char *tempfile = "temp.xxxxxx";

main() {
    extern onintr();

    if(signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    mktemp(tempfile);
    /* Process ... */
    exit(0);
}

onintr() { /* почистить, если прервано */
    unlink(tempfile);
    exit(1);
}
```

Почему в main имеют место проверки и двойной вызов signal? Вспомните, что сигналы посылаются всем процессам, запущенным с данного терминала. Соответственно если

программа должна быть запущена не в диалоговом режиме (с помощью &), shell делает так, что она будет игнорировать прерывания. Поэтому сигналы прерывания, посланные основным процессам, не остановят ее. Если бы эта программа началась с объявления о том, что все прерывания, которые должны быть посланы подпрограмме onintr, не принимаются во внимание, были бы сведены на нет все усилия shell защитить ее при запуске в фоновом режиме.

Решение, показанное выше, состоит в том, чтобы проверить состояние обработки прерываний, если они игнорировались ранее. Функции программы в том виде, в каком она написана, зависят от возвращаемого signal предыдущего состояния конкретного сигнала. Если сигналы уже игнорировались, процесс должен продолжить это дело; в противном случае их следует перехватывать.

Более сложная программа может перехватить прерывание и интерпретировать его как запрос на прекращение своих действий и возврат к основному циклу обработки команд. Подумаем о текстовом редакторе: прерывание длинного вывода на печать не должно вызывать завершения редактирования и потерю уже отредактированного текста. Программа для такого случая может быть написана следующим образом:

```
#include <signal.h>
#include <setjmp.h>

jmp_buf sjbuf;

main() {
    int onintr();
    if(signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    setjmp(sjbuf);
    /* сохранить текущую позицию стека */
    for(;;) {
    /* главный рабочий цикл */
    }
    ...
}

onintr() {                    /* установить если прервано */
    signal(SIGINT, onintr);   /* установить для следующего прерывания */
    printf("\nInterrupt\n");
    longjmp(sjbuf, 0);        /* вернуться в сохраненное состояние */
}
```

Файл <setjmp.h> описывает тип jmp_buf как объект, в котором сохраняется позиция стека; sjbuf считается таким объектом. Функция setjmp(3) сохраняет запись о том, где выполняется программа. Значения переменных не сохраняются. Когда происходит прерывание, выполняется обращение к подпрограмме onintr, которая может печатать сообщения, устанавливать флаги и т. д. Функция longjmp берет в качестве аргумента объект, сохраненный setjmp, и возвращает управление в ячейку после вызова setjmp. Поэтому управление (и значение уровня стека) будет возвращено обратно в основную программу — ко входу в головной цикл.

Отметим, что после прерывания сигнал вновь настраивается на onintr. Это обусловлено тем, что когда сигналы возникают, они автоматически настраиваются на реакцию по умолчанию:

Некоторые программы, которые "хотят" обнаружить сигналы, просто не могут быть остановлены в произвольный момент, например в середине обновления сложных составных данных. Решение состоит в том, что подпрограмма обработки прерывания должна установить флаг и вернуться к месту вызова exit или longjmp. Выполнение программы продолжится точно с того места, где оно было прервано, а флаг прерывания будет проверен позднее.

С этим подходом связана одна трудность. Предположим, что, когда посылается сигнал прерывания, программа читается с терминала. Описанная подпрограмма непременно вызывается; она устанавливает свой флаг и возвращается. Если бы, как отмечалось выше, было верно то, что выполнение возобновляется точно с того места, где оно прервалось, программа продолжала бы чтение с терминала до ввода пользователем другой строки. Однако здесь возникает недоразумение, поскольку пользователь может не знать, что программа читает, и предположительно предпочел бы, чтобы сигнал сразу оказал действие. Для разрешения проблемы система должна закончить read, но с сообщением об ошибке, указывающим, что произошло: errno присваивается EINTR, определенное в <errno.h>, чтобы обозначить прерванный системный вызов.

Так, программы, которые "ловят" сигналы и продолжают после этого свою работу, должны быть готовы к появлению ошибок, вызванных прерванными системными вызовами. (Следует остерегаться системных вызовов read — чтение с терминала, wait, pause). Такая программа при чтении стандартного входного потока могла бы использовать фрагмент, подобный следующему:

```
#include <errno.h>
extern int errno;

...

if (read(0, &c, 1) <= 0)   /* EOF или прерывание */
    if (errno == EINTR) {  /* EOF, вызванный прерыванием */
        errno = 0;         /* устанавливается для следующего раза */
    } else {               /* настоящий конец файла */

    ...

    }
```

Очень сложно постоянно следить за тем, как реакция на сигнал комбинируется с выполнением других программ. Предположим, программа ловит сигналы прерывания и располагает средствами (типа "!"в ed) для выполнения других программ. Тогда программа могла бы выглядеть так:

```
if (fork() == 0)
    execlp(...);
signal(SIGINT, SIG_IGN);  /* родитель игнорирует прерывание */
wait(&status);           /* пока потомок не завершился */
signal(SIGINT, onintr);   /* восстанавливает прерывания */
```

Почему? Сигналы посылаются всем вашим процессам. Предположим, программа, которую вы вызвали, ловит свои собственные сигналы прерывания, как это делает редактор. Если вы прервете выполнение подпрограммы, она получит сигнал, вернется к своему главному циклу и, возможно, начнет читать с вашего терминала. Но вызывающая программа также перейдет от wait к подпрограмме и будет читать с терминала. Два процесса, читающие с вашего терминала, создадут трудную ситуацию, так как в результате системе придется гадать, к кому попадет та или иная строка входного потока. Решение состоит в том, чтобы родительская программа игнорировала прерывания, пока не завершился процесс-потомок. Это решение нашло свое отражение при обработке сигнала в system:

```
#include <signal.h>

system(s)       /* run command line 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;
}
```

Несколько слов по поводу описаний: функция signal, очевидно, имеет довольно странный второй аргумент. Фактически он представляет собой указатель на функцию, поставляющую целое значение, и в то же время это тип самой подпрограммы сигнала. Две величины, `SIG_IGN` и `SIG_DFL`, имеют правильный тип, но выбраны так, что не совпадают ни с одной из существующих функции. Для любознательных покажем, как они определены для PDP-11 и VAX: определения, видимо, достаточно "неуклюжи", чтобы стимулировать использование <signal.h>.

```
#define SIG_DFL (int(*)())0
#define SIG_IGN (int(*)())1
```

Будильники. Системный вызов alarm(n) обеспечивает посылку сигнала SIGALRM вашему процессу через n секунд. Сигнал будильника может быть использован для того,

чтобы удостовериться в возникновении каких-то событий за соответствующее время. Если что-нибудь произошло, сигнал будильника может быть выключен; в противном случае процесс может получить управление, перехватив этот сигнал.

Для иллюстрации приведем программу `timeout`, которая запускает другую команду; если команда не закончила свое выполнение за определенное время, она будет завершена по звонку будильника. Например, вспомните команду watchfor из гл. 5. Вместо того чтобы запускать ее без ограничения времени работы, установите ограничение в часах:

```
$ timeout -3600 watchfor dmg &
```

Программа `timeout` демонстрирует почти все возможности, которые мы обсуждали в последних двух разделах. Создан процесс-потомок, родительский процесс устанавливает будильник и затем ждет, пока потомок завершит работу. Если будильник "зазвенел" первым, потомок уничтожается. Делается попытка вернуть состояние потомка при выходе:

```
/* timeout:  set time limit on a process */
#include <stdio.h>
#include <signal.h>
int     pid;        /* child process id */
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()       /* kill child when alarm arrives */
{
```

```
    kill(pid, SIGKILL);
}
```

■ Упражнение: Можете ли вы представить, как реализована `sleep`? Подсказка: `pause(2)`. При каких обстоятельствах (если это вообще возможно) `sleep` и `alarm` могли бы помешать друг другу?

Историческая и библиографическая справка. Детального описания реализации системы UNIX не существует отчасти потому, что программа является собственностью фирмы. В статье К. Томпсона "UNIX implementation" (BSTJ, July, 1978) описываются основные идеи. Другие статьи, в которых обсуждаются связанные с UNIX темы, это "The UNIX system — a retrospective" (BSTJ, July, 1978) и "The evolution of the UNIX time-sharing system" (Symposium on Language Design and Programming Methodology, Springer — Verlag, Lecture Notes in Computer Science #79, 1979). Обе статьи принадлежат Д. Ритчи.

Программа `readslow` была разработана П. Вейнбергером как недорогое средство, позволяющее следить за успехами шахматной машины "Белла" К. Томпсона и Дж. Кондона во время шахматных турниров. "Белла" записывала позиции своей игры в файл, а зрители просматривали файл с помощью `readslow`, не отнимая драгоценного времени у машины. (Новейшая версия "Беллы" лишь небольшую долю вычислений выполняет на основной машине, так что проблема снята.)

На создание spname нас вдохновил Т. Дафф. Статья А. Дархема, Д. Лэмба и Дж. Сакса "Spelling correction in user interfaces" (CACM, October, 1983) представляет иной способ коррекции написания в контексте программы почты.

Глава 8

Разработка программ

Первоначально системе UNIX предназначалась роль среды для разработки программ. В настоящей главе мы обсудим некоторые применяемые с этой целью программные средства на примере солидной программы — интерпретатора языка программирования, сравнимого по мощности с Бейсиком. Мы выбрали реализацию языка, потому что возникающие здесь проблемы характерны для больших программ. Более того, многие программы полезно рассматривать как языковые процессоры, преобразующие входной поток определенной структуры в последовательность действий и выходной поток, т. е. мы хотим продемонстрировать вам программные средства разработки языков.

В частности, вашему вниманию предлагаются следующие программы:

- `yacc` — генератор синтаксических анализаторов; программа, которая по описанию грамматики языка порождает анализатор;
- `make` — программа, определяющая процесс компиляции сложных программ и управляющая им;
- `lex` — программа, аналогичная `yacc`, но создающая лексические анализаторы.

Мы покажем вам приемы разработки программ в несколько этапов — от простого к сложному. Ниже описаны шесть этапов реализации языка, каждый из которых поучителен уже сам по себе. Эти этапы отражают реальный процесс написания программы:

1. Создание калькулятора с четырьмя действиями: +, -, *, / (и со скобками). Калькулятор выполняет операции над числами с плавающей точкой, каждая строка состоит из одного выражения; полученное значение печатается сразу.

2. Добавление переменных с именами от a до z. В этой версии есть также унарный минус и некоторые средства защиты от ошибок.

3. Добавление переменных с именами произвольной длины, встроенных функций для sin, exp и т. п., полезных констант типа (обозначается как PI) и операции возведения в степень.

4. Внесение внутренних изменений: оператор вычисляется не непосредственно, а порождает код, который впоследствии интерпретируется. Новые возможности не добавляются, но подготавливается переход к п. 5.

5. Добавление структур управления: if-else и while — группирование операторов с помощью и и операции отношений типа >, <= и т. п.

6. Добавление рекурсивных процедур и функций с параметрами, а также операторов для ввода–вывода строк и чисел.

Окончательная версия языка описана в гл. 9 как пример программных средств подготовки документации системы UNIX. В приложении 2 приводится справочное руководство по калькулятору.

Эта глава довольно объемная, поскольку в ней детально рассматривается, как правильно написать нетривиальную программу. Предполагается, что вы знаете язык Си и имеете под рукой экземпляр справочного руководства по системе UNIX (том 2), поскольку просто невозможно объяснить все нюансы. Будьте готовы к тому, что вам придется прочитать главу несколько раз. Окончательная версия полностью представлена в приложении 3. Заметим, кстати, что мы долго спорили из-за имени языка, но так и не придумали подходящее. Остановились на hoc, что означает "калькулятор высокого уровня" (high level calculator).

Его версии соответственно называются `hoc1`, `hoc2` и т. д.

8.1 Этап 1: калькулятор с четырьмя действиями

Прежде всего рассмотрим реализацию `hoc1` — программы с такими же возможностями, как и простейший карманный калькулятор, но гораздо менее удобной для переноса. Она выполняет четыре операции: `+`, `-`, `*`, `/` и, имеет скобки с произвольной глубиной вложенности, чем обладают лишь немногие калькуляторы. Если вы введете выражение и символ *RETURN*, результат будет напечатан в 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 near line 4 No unary minus yet
$
\end{verbarim}

Грамматика. С появлением формы Бэкуса --- Наура, предложенной для
Алгола, языки стали описываться с помощью формальной грамматики.
Абстрактное описание грамматики \cmd{hoc1} простое и краткое:

\begin{verbatim}
список: выраж \n
    список выраж \n
выраж: NUMBER
```

```
    выраж + выраж
    выраж --- выраж
    выраж * выраж
    выраж / выраж
    ( выраж )
\end{verbarim}
```

Здесь список --- последовательность выражений, каждое из которых завершается символом перевода строки, а выражение --- число или пара выражений, объединенных операцией, либо выражение в скобках.

Приведенное описание не полное, так как в нем не определены естественный приоритет и ассоциативность операций, а также не заданы значения конструкциям языка. Хотя список специфицируется через выраж, а оно в свою очередь через \kbd{NUMBER}, само \kbd{NUMBER} нигде не определено, Поэтому чтобы перейти от упрощенного описания к работающей программе, необходимо внести ясность в эти вопросы.

Программа \cmd{yacc}. Генератор синтаксических анализаторов \cmd{yacc} (Автор \cmd{yacc} С.~Джонсон назвал свою программу ‘‘еще одним компилятором компиляторов’’ (yet another compiler--compiler), поскольку во время ее разработки (1972~г.) уже существовало довольно большое число таких программ, \cmd{yacc} --- одна из немногих, получивших признание.) преобразует компилятор грамматических правил языка, подобных приведенным выше, в анализатор, который разбирает операторы языка. \cmd{Yacc} обладает возможностью приписывать значения компонентам грамматики таким образом, что в процессе разбора значение может быть ‘‘вычислено’’. Используется \cmd{yacc} поэтапно,

На первом этапе записывается грамматика языка, но более точно, чем было показано ранее, т.~е. определяется синтаксис. На этом этапе назначение \cmd{yacc} - предупреждение появления ошибок и двусмысленностей в грамматике.

На втором этапе каждое правило (правило вывода грамматики) сопровождается описанием действия на тот случай, когда найден экземпляр грамматической конструкции в разбираемой программе. Часть действия записывается на Си, при\-чем должны выполняться определенные соглашения о связи между грамматикой и текстом. Здесь определяется семантика языка.

Третий этап --- создание лексического анализатора, который должен читать разбираемый входной поток и разбивать его для анализатора на осмысленные единицы. Примером лексической единицы длиной в несколько символов может служить \code{NUMBER}; операции из одного символа, такие, как \code{+} и \code{*}, также являются лексическими единицами. По традиции лексические единицы называют лексемами.

На следующем этапе разрабатывается управляющая процедура, которая
вызывает анализатор, созданный \cmd{yacc}.

Программа \cmd{yacc} преобразует грамматику и семантические процедуры
в функцию разбора с именем yyparse и записывает ее в виде файла с
текстом на Си. Если \cmd{yacc} не находит ошибок, то анализатор,
лексический анализатор и управляющую процедуру можно откомпилировать,
возможно, связать с другими программами на Си и выполнить.

Действие \cmd{yacc} сводится к многократному обращению к лексическому
анализатору за лексемами, распознаванию грамматических
(синтаксических) конструкций во входном потоке и выполнению
семантических процедур по мере распознавания грамматических правил.
Вызывать лексический анализатор нужно по имени \code{yylex}, так как
именно эту функцию инициирует анализатор \code{yyparse} всякий раз,
когда ему нужна следующая лексема. (Все имена, используемые
\cmd{yacc}, начинаются с \code{y}.)

Чтобы быть более точными, укажем, что входной поток для \cmd{yacc}
должен иметь такой вид:

```
\begin{verbatim}
%{
    Операторы Си типа #include, описания и т.д.
    Эта часть необязательна.
%}
yacc-описания: лексемы, грамматические переменные, информация о приоритетах и
ассоциативности
%%
грамматические правила и действия
%%
еще операторы Си (необязательно):
main() { ...; yyparse(); ... }
yylex() { ... }
...
```

Этот поток поступает на вход `yacc`, а результат записывается в файл `y.tab.c`, имеющий следующую структуру:

```
Операторы на Си между %{ и %}, если есть
Операторы на Си из части после второй комбинации %%, если есть:
main() { ...; yyparse(); ... }
yylex() { ... }
...
yyparse () { анализатор, который вызывает yy1ex() }
```

Такой подход типичен для системы UNIX: `yacc` выдает текст на Си, а не оттранслированный файл (`.o`), что является наиболее гибким решением, так как созданный текст,

переносим и легко поддается любому другому преобразованию (если появится хорошая идея).

Генератор yacc сам по себе представляется мощным программным средством. Его изучение потребует от вас, конечно, некоторых усилий, но все ваши "затраты" многократно окупятся. Анализаторы, создаваемые yacc, — небольшие, эффективные и корректные (хотя за семантические преобразования отвечаете вы). Кроме того, многие неприятные проблемы, связанные с процессом разбора, решаются автоматически. Программы языковых распознавателей достаточно легко создавать и, что, возможно, еще более важно, изменять по мере совершенствования определения языка.

Использование программ на этапе 1. Исходный текст hoc1 состоит из грамматических правил с описанием действий лексической процедуры yylex и функции main, хранимых в одном файле hoc.y. (Имена файлов, содержащих текст для yacc, традиционно оканчиваются на .y, но это соглашение в отличие от соглашения о cc и .c не поддерживает сам yacc.) Грамматика составляет первую половину файла hoc.y:

```
$ cat hoc.y
%{
#define YYSTYPE double  /* data type of yacc stack */
%}
%token  NUMBER
%left   '+' '-'   /* left associative, same precedence */
%left   '*' '/'   /* left assoc., higher precedence */
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */
...
```

Вы видите, как много информации заключено в этих нескольких строках. Поскольку мы не можем вам здесь все объяснить, в частности, как работает синтаксический анализатор, обратитесь к справочному руководству по yacc.

Альтернативные правила разделены символом '|'. С каждым грамматическим правилом может быть связано определенное действие, которое выполняется, когда экземпляр этого правила распознается во входном потоке. Действие описывается последовательностью операторов Си, заключенной в фигурные скобки. Внутри последовательности $n (т. е. $1, $2 и т. д.) определяет значение, вырабатываемое n-м компонентом правила, а $$–значение, вырабатываемое всеми компонентами правила в целом. Так, в правиле

```
expr:     NUMBER        { $$ = $1; }
```

`$1` — значение, вырабатываемое при распознавании `NUMBER`, и оно же является результирующим значением `expr`. В данном случае присваивание `$$ = $1` может быть опущено, так как `$$` всегда принимает значение `$1` (если не устанавливается явно каким–либо иным образом). В следующей строке с правилом

```
expr:     expr '+' expr { $$ = $1 + $3; }
```

результирующее значение выраж является суммой двух компонентов, тоже `expr`. Отметим, что `$2` соответствует '+' т. е. каждый компонент пронумерован.

Строкой выше выражение, за которым следует символ перевода строки ('`\n`'), распознается как список, и печатается его значение. Если за такой конструкцией следует конец входного потока, процесс разбора завершается правильно. Список может быть пустой строкой; так учитываются пустые входные строки.

Формат входного потока для `yacc` — произвольный. Наш формат рекомендуется как стандартный.

В этой реализации процесс распознавания или разбора входного потока приводит к немедленному вычислению выражения. В более сложных решениях (включая `hoc4` и его последующие версии) процесс разбора порождает код для дальнейшего выполнения.

Наглядно представить разбор вам поможет рис. 8.1, где изображено дерево разбора. Кроме того, вы должны знать, как вычисляются значения и как они распространяются от листьев к корню дерева.

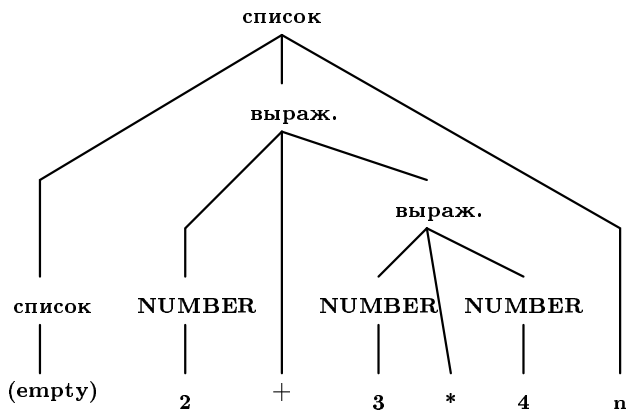


Рис. 8.1: Дерево разбора для $2 + 3 * 4$

Реально значения не полностью разобранных правил хранятся в стеке и через стек передаются от одного правила к следующему. Обычно данные в стеке имеют целый тип, но поскольку мы в своей работе используем числа с плавающей точкой, необходимо переопределить значение по умолчанию. Определение

```
#define YYSTYPE double
```

устанавливает двойную точность для типа данных стека.

Теперь перейдем к описанию синтаксических классов, распознаваемых лексическим анализатором, если только они не являются литералами, состоящими из одного символа вида '+' и '-'. Описание `%token` специфицирует одни или несколько таких объектов. При необходимости можно задать левую или правую ассоциативность, используя `%left` или `%right` вместо `%token`.

(Левая ассоциативность означает, что `a-b-c` будет разбираться как `(a - b) - c`, а не `a - (b - c)`.) Приоритет устанавливается порядком появления операции: лексемы из одного определения имеют один и тот же приоритет, а лексемы, специфицированные позднее, — более высокий. Таким образом, в грамматике может быть неоднозначность (т. е. для некоторых входных потоков существует несколько способов разбора), но дополнительная информация в определениях разрешает эту неоднозначность.

Вторую половину файла `hoc.y` составляют процедуры:

```
/* Продолжение hoc.y */

#include <stdio.h>
#include <ctype.h>
char    *progname;      /* for error messages */
int     lineno = 1;

main(argc, argv)        /* hoc1 */
    char *argv[];
{
    progname = argv[0];
    yyparse();
}
```

Функция `main` обращается к `yyparse` для разбора входного потока. Переход в цикле от одного выражения к другому происходит в рамках грамматики с помощью последовательности правил вывода для списка. Приемлемо также обращаться в цикле к `yyparse` из функции `main`, если действия для списка предполагают печать значения и немедленный возврат.

Функция `yyparse` в свою очередь многократно обращается за лексемами из входного потока к функции `yylex`. Наша функция `yylex` проста: в ее задачи входят пропуск пробелов и символов табуляции, преобразование цифровых строк в числовое значение и подсчет входных строк для вывода сообщений об ошибках. Поскольку грамматика допускает только `+`, `-`, `*`, `/`, `(`, `)` и `\n`, при появлении любого другого символа yyparse выдает сообщение об ошибке. Получение `0` означает для `yyparse` “конец файла”.

```
/* Продолжение hoc.y */

yylex()         /* hoc1 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {   /* number */
        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 стек yacc. Функция yylex возвращает тип лексемы, равно как и ее функциональное значение, и приравнивает yylval значению лексемы (если оно есть). Например, число с плавающей точкой имеет тип NUMBER и значение, скажем, 12.34. Для некоторых лексем, прежде всего состоящих из одного символа, таких, как '+' или '\n', в грамматике используется только тип. В этом случае yylval не нужно определять.

Определение %token NUMBER из входного файла для yacc преобразуется в оператор #define в выходном файле y.tab.c, поэтому NUMBER можно использовать в качестве константы в любом месте Си–программы. Yacc выбирает такие значения, которые не будут смешиваться с символами ASCII.

При наличии синтаксической ошибки yyparse обращается к yyerror со строкой, содержащей загадочное сообщение: "syntax error" ("синтаксическая ошибка"). Предполагается, что функцию yyerror предоставляет пользователь: в нашей функции строка просто передается другой функции — warning, которая выдает некоторую дополнительную информацию. В последующих версиях hoc функция warning будет применяться непосредственно.

```
yyerror(s)      /* called for yacc syntax error */
    char *s;
{
    warning(s, (char *) 0);
}

warning(s, t)   /* print warning message */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

Этим завершаются процедуры файла hoc.y. Трансляция программы для yacc происходит в два этапа:

```
$ yacc hoc.y                  Выходной поток попадает в y.tab.c
$ cc y.tab.c -o hoc1          Выполняемая программа попадает в hoc1
$ hoc1
2/3
0.66666667
-3-4
hoc1: syntax error near line 1
$
```

■ Упражнение: Исследуйте структуру файла `y.tab.c` (для `hoc1` это составляет около 300 строк текста).

Внесение изменений — унарный минус. Ранее мы утверждали, что, работая с `yacc`, легко менять язык. В качестве примера добавим к `hoc1` унарный минус, чтобы выражения типа

```
-3-4
```

вычислялись, а не отвергались как синтаксические ошибки. Всего две строки нужно дополнительно включить в `hoc.y`. Добавляется новая лексема `UNARYMINUS` в ту часть грамматики, где задаются приоритеты, чтобы унарный минус имел наивысший приоритет:

```
%left'+''-'
%left '*' '/'
%left UNARYMINUS /* новая лексема */
```

Грамматика увеличивается на одно правило для `expr`:

```
expr: NUMBER ($$= $1;}
    | '-'expr %prec UNARYMINUS {$$=- $2} /* новое */
```

Определение `%prec` "говорит", что символ унарного минуса (т. е. знак "`-`" перед выражением) имеет тот же приоритет, что и `UNARYMINUS` (наивысший); действие заключается в изменении знака. Приоритет минуса между двумя выражениями устанавливается по умолчанию.

■ Упражнение: Добавьте операции `%` (взятие остатка) и унарный плюс к `hoc1`. Рекомендация: обратитесь к справочному руководству по `frexp(3)`.

Некоторые замечания относительно `make`. Обидно, что приходится вводить две команды для компиляции `hoc1`. Хотя, конечно, нетрудно составить командный файл для такого задания, но есть лучший способ, который позднее можно распространить на тот случай, когда программа состоит из нескольких исходных файлов. Программа `make` читает описания взаимозависимости компонентов программы и позволяет создать ее действующую версию. Она проверяет время последней модификации каждого компонента, выясняет минимальный объем перекомпиляции, которую необходимо выполнить для получения новой действующей версии, и затем запускает процесс. Программа `make` разбирается в запутанных многошаговых процессах, в частности в `yacc`, поэтому ей можно давать задания, не уточняя отдельные шаги.

Особенно полезно обращаться к `make`, когда создаваемая программа настолько велика, что "располагается" в нескольких исходных файлах. Однако она удобна и для таких малых программ, как `hoc1`. Ниже приведены описания команд для `make`, рассчитанные на `hoc1`, которые `make` предполагает найти в файле с именем `makefile`.

```
$ cat makefile
hoc1: hoc.o
    cc hoc.o -o hoc1
$
```

Здесь сообщается, что hoc1 зависит от hoc.o и что hoc1 создается из hoc.o с помощью команды cc, которая запускает компилятор Си, помещая выходной поток в файл hoc1. Программа make уже "знает", как преобразовать входной файл для yacc hoc.y в выходной файл hoc.o:

```
$ make                  Проделаем первый раз получение hoc1 с помощью make
yacc hoc.y
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cc hoc.o -o hoc1
$ make                  Попробуем еще раз
'hoc1' is up to date    make понимает, что это не нужно
$
```

8.2 Этап 2: переменные и восстановление после ошибки

Следующий шаг — переход от hoc1 к hoc2, который сводится к расширению памяти (в памяти хранится 26 переменных с именами от a до z). Это довольно несложный и весьма полезный промежуточный этап. Мы также введем здесь процесс обработки ошибок. Если вы проверите hoc1, то убедитесь, что реакцией на синтаксические ошибки являются вывод сообщения и прекращение работы. Поведение же hoc1 в случае арифметических ошибок типа деления на нуль достойно всяческого порицания:

```
$ hoc1
1/0
Floating exception --- core dump
$
```

Для реализации новых возможностей требуются лишь небольшие изменения: приблизительно 35 строк текста. Лексический анализатор yylex должен распознавать буквы как переменные, а грамматика — содержать правила вывода вида

```
expr: VAR
    | VAR '=' expr
```

Выражение может содержать операцию присваивания; разрешены также многократные присваивания типа

```
x = y = z = 0
```

Простейший способ хранения значений переменных — создать массив из 26 элементов; однобуквенную переменную можно использовать в качестве индекса массива. Однако если анализатору предстоит обрабатывать и имена переменных, и значения в одном стеке, необходимо сообщить yacc, что элемент стека является объединением double и int, а не просто элементом типа double. Это делается с помощью описания %union. Описания #define или typedef подходят для определения стека из базовых типов как double, но для типов объединения требуется описание %union, поскольку yacc осуществляет контроль типов в выражениях вида $$ = $2.

Ниже приведена часть определения грамматики hoc.y для программы hoc2:

```
$ cat hoc.y
%{
double  mem[26];        /* memory for variables 'a'..'z' */
%}
%union {                /* stack type */
        double  val;    /* actual value */
        int     index;  /* index into mem[] */
}
%token  <val>   NUMBER
%token  <index> VAR
%type   <val>   expr
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */
```

Из описания `%union` следует, что элементы стека содержат или число с двойной точностью (обычный случай), или целое, являющееся индексом в массиве `mem`. В описании `%token` дополнительно указывается тип значения. В описании `%type` есть сведения о том, что выраж является элементом объединения `<val>`, т. е. `double`. Информация о типе позволяет `yacc` обращаться к нужному элементу объединения. Обратите внимание: "`=`" представляет собой правоассоциативную операцию, тогда как другие операции — левоассоциативные.

Обработка ошибок происходит в несколько этапов. Прежде всего производится проверка на нулевой делитель: если делитель равен нулю, вызывается процедура обработки ошибок `execerror`. Второй этап заключается в перехвате сигнала "переполнение вещественного" (`"floating point exception"`), который возникает при переполнении вещественного числа. Сигнал устанавливается в функции `main`. Последний шаг восстановления

после ошибки заключается в добавлении к грамматике правила вывода для ошибки. В грамматике yacc слово error зарезервировано; оно дает возможность анализатору осознать синтаксическую ошибку и восстановиться после нее. Если произойдет ошибка, yacc в конце концов использует это правило, распознает ошибку как грамматически "правильную" конструкцию и, таким образом, восстановится. Действие yyerrok заключается в установке признака в анализаторе, который позволяет вернуться ему назад в состояние осмысленного разбора. Восстановление после ошибки — сложная проблема для всех анализаторов. Мы показали вам здесь лишь самые элементарные приемы и только обозначили возможности yacc.

В грамматике hoc2 произошли незначительные изменения. Ниже приведена функция main, дополненная обращением к setjmp. Оно позволяет запомнить то нормальное состояние, которое будет использовано при восстановлении после ошибки. В фукнции execerror происходит соответствующее обращение к longjmp. (Описание setjmp и longjmp см. в разд. 7.5.)

```
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#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) /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
    execerror("floating point exception", (char *) 0);
}
```

В целях отладки мы сочли удобным, чтобы функция execerror вызывала abort (см. справочное руководство по abort(3)), что приведет к распечатке содержимого памяти,

которую затем смогут использовать программы adb и sdb. Когда разработка программы полностью завершится, обращение к abort будет заменено на longjmp.

В программе hoc2 лексический анализатор несколько иной. В нем учтено различие строчных и прописных букв, а поскольку теперь yyval является объединением, нужно выбрать подходящий элемент перед выходом из yylex. Ниже показаны измененные фрагменты:

```
yylex()         /* hoc2 */
{
...
    if (c == '.' || isdigit(c)) {   /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    if (islower(c)) {
        yylval.index = c --- 'a'; /* ASCII only */
        return VAR;
    }
...
```

Еще раз отметим, что тип лексемы (т. е. NUMBER) не совпадает с ее значением (например, 3.1416).

Продемонстрируем новые возможности hoc2 — переменные и способность восстановления после ошибки:

```
$ hoc2
x = 355
355
y = 113
113
p = x/z                                 z не определено, а значит, равно 0
hoc2: division by zero near line 4      Восстановление после ошибки
x/y
3.1415929
1e30 * 1e30                             Переполнение
hoc2: floating point exception near line 5
...
```

В самом деле, для **PDP-11** требуются вполне конкретные меры, чтобы обнаружить переполнение вещественного, но на большинстве других машин hoc2 действует так, как показано выше.

■ Упражнение: Обеспечьте возможность запоминания последнего вычисленного значения, чтобы его не приходилось вводить снова для последовательности связанных вычислений. Одним из решений может быть использование какой-либо переменной, например 'p', в качестве "предыдущего" (previos) значения.

■ Упражнение: Измените программу `hoc` так, чтобы можно было использовать символ `';'` как разделитель выражений наравне с символом перевода строки.

8.3 Этап 3: переменные с произвольными именами; встроенные функции

В версию `hoc3` добавлено несколько новых средств, из-за чего увеличился текст программы. Основное нововведение — возможность обращения к встроенным функциям:

```
sin cos atan exp log  log10 sqrt ing abs
```

Введена также дополнительно операция возведения в степень `'^'` (право–ассоциативная с наивысшим приоритетом).

Поскольку лексический анализатор должен справляться с встроенными именами длиной более чем в один символ, не так уж много усилий придется приложить, чтобы допустить переменные с именами произвольной длины. Для хранения информации об этих переменных нужна довольно сложная таблица имен, но если мы ее создаем, то можно заранее задать в ней вместе с именами значения некоторых полезных констант:

`PI`	3.14159265358979323846	Число пи
`E`	2.71828182845904523536	Основание натурального логарифма
`GAMMA`	0.57721566490153286060	Константа Эйлера–Маскерони
`DEG`	57.2957795130823208768	Отношение градуса к радиану
`PHI`	1.61803398874989484820	Золотое сечение

В результате получим полезный калькулятор:

```
$ hoc3
1.5^2.3
2.5410306
exp(2*3*log(1.5))
2.5410306
sin(PI/2)
1
atan(1)*DEG
45
...
```

Несколько улучшилась и работа распознавателя. В `hoc2` присваивание `x = expr` не только вызывало присваивание, но и приводило к печати значения, поскольку все выражения печатаются:

```
$ hoc2
x=2*3.14159
6.28318           В случае присваивания переменной значение печатается
...
```

В программе hoc3 проводится различие между присваиваниями и выражениями; значения печатаются только для выражений:

```
$ hoc3
x=2*3.14159  Присваивание: значение не печатается
                 Выражение:
6.28318          Значение печатается
```

Получившаяся в результате всех этих изменений программа настолько велика (около 250 строк текста), что для простоты редактирования и ускорения компиляции лучше разбить ее на отдельные файлы. Итак, теперь мы имеем пять файлов вместо одного:

hoc.y	грамматика, main, yylex (как и прежде);
hoc.h	глобальные структуры данных для включения в другие файлы;
symbol.c	функции, работающие с таблицей имен: lookup, install;
unit.c	встроенные функции и константы; init;
math.c	функции для вызова стандартных математических функций: Sqrt, Log и т. д.

Необходимо более детально познакомиться с работой Си–программы, состоящей из нескольких файлов, и программы make, чтобы переложить на нее часть своих обязанностей.

Вернемся снова к программе make и рассмотрим вначале структуру таблицы имен. Поименованный объект имеет имя, тип (VAR или BLTIN) и значение. Так, объект типа VAR имеет значение double; если объект является встроенным, то его значением служит указатель на функцию, возвращающую double. Данная информация используется в hoc.y, symbol.c и init.c. Ее можно размножить в трех экземплярах, но тогда легко ошибиться или забыть исправить один из экземпляров при внесении изменений. Вместо этого мы поместили общую информацию в файл макроопределений hoc.h, который можно включить при необходимости в любой файл. (Окончание .h традиционно, но не контролируется никакими программами.) В файл makefile также добавлены сведения о зависимости исходных файлов от hoc.h, чтобы при изменении hoc.h была проведена требуемая перекомпиляция.

```
$ cat hoc.h
typedef struct Symbol { /* symbol table entry */
    char    *name;
    short   type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;            /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } u;
    struct Symbol   *next;  /* to link to another */
} Symbol;
Symbol  *install(), *lookup();
$
```

Тип UNDEF обозначает VAR, которой пока не присвоили значения. Объекты связаны в список с помощью элемента next в записи Symbol. Сам список является локальным для symbol.c, доступ к нему возможен только посредством функций lookup и install. Это

позволяет в случае необходимости легко менять структуру таблицы имен (что мы уже сделали однажды). Функция `lookup` отыскивает в списке заданное имя и возвращает указатель на `Symbol`, если имя найдено, и `0` — в противном случае. Таблица имен рассчитана на линейный поиск, что вполне допустимо для диалогового калькулятора, поскольку поиск имен выполняется не во время его работы, а в процессе разбора. Функция `install` вносит переменную и связанные с ней тип и значение в начало списка. Функция `emalloc` обращается к стандартной функции размещения `malloc` (см. справочное руководство по `malloc(3)`) и проверяет результат. Указанные три функции составляют содержимое файла `symbol.c`. Файл `y.tab.h` создается при выполнении команды `yacc -d`; он содержит операторы `#define`, порождаемые `yacc` для лексем `NUMBER`, `VAR`, `BLTIN` и т. д.

```
$ cat 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;
}
$
```

Файл `init.c` содержит определения констант (`PI` и т. п.) и указатели на встроенные функции; они заносятся в таблицу имен функцией `init`, находящейся в `main`.

```
$ cat init.c
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Exp(), Sqrt(), integer();

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 argument */
    "log10", Log10, /* checks argument */
    "exp",  Exp,    /* checks argument */
    "sqrt", Sqrt,   /* checks argument */
    "int",  integer,
    "abs",  fabs,
    0,      0
};

init()  /* install constants and built-ins in table */
```

```
{
    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 вводятся в текст программы, чтобы легче было их читать и изменять. Таблицы определены как статические, что обеспечивает их доступность только в данном файле. Мы вскоре вернемся к обсуждению стандартных математических функций типа `Log` и `Sqrt`.

Построив такой базис, можно перейти к изменениям в грамматике, которые осуществляются на его основе.

```
$ cat hoc.y
%{
#include "hoc.h"
extern  double  Pow();
%}
%union {
    double  val;    /* actual value */
    Symbol  *sym;   /* symbol table pointer */
}
%token  <val>   NUMBER
%token  <sym>   VAR BLTIN UNDEF
%type   <val>   expr asgn
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */
...
```

Теперь в грамматике присутствует `asgn` для присваивания, подобно `expr` для выражения. Входная строка, состоящая только из

```
VAR = expr
```

является присваиванием, и, следовательно, ни одно из значений не печатается. Заметьте, кстати, как мы легко добавили к грамматике операцию возведения в степень, являющуюся правоассоциативной.

Для стека `yacc` используется другое определение `%union`: вместо представления переменной как индекса в массиве из 26 элементов введен указатель на объект типа `Symbol`. Файл макроопределений `hoc.h` содержит определение этого типа.

Лексический анализатор распознает имена переменных, находит их в таблице имен и определяет, относятся ли они к переменным (VAR) или к встроенным функциям (BLTIN). Функция `yylex` возвращает один из указанных типов. Заметим, что определенные пользователем переменные и предопределенные переменные типа `PI` относятся к `VAR`.

Одно из свойств переменной состоит в том, что ей может быть присвоено либо не присвоено значение, поэтому обращение к не определенной переменной должно диагностироваться программой `yyparse` как ошибка. Возможность проверки переменной (определена она или нет) должна быть предусмотрена в грамматике, а не в лексическом анализаторе. Когда `VAR` распознается на лексическом уровне, контекст пока еще не известен, но нам не нужны сообщения о том, что `x` не определен, хотя контекст и вполне допустимый, как, например, `x` в присваивании типа `x = 1`.

Ниже приводится измененная 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 функции `main` добавлена еще одна строка, в которой вызывается процедура инициации `init` для занесения в таблицу имен встроенных и предопределенных имен типа `PI`:

```
main(argc, argv)        /* hoc3 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}
```

Теперь остался только файл `math.c`. Для некоторых стандартных математических функций требуется обработка ошибок для диагностики и восстановления, например, стандартная функция по умолчанию возвращает 0, если аргумент отрицателен. Функции из файла `math.c` используют контроль ошибок, описанный в разд. 2 справочного руководства по UNIX (см. гл. 7). Это более надежный и переносимый вариант, чем введение своих проверок, так как, вероятно, конкретные ограничения функций полнее учитываются в "официальной" программе. Файл макроопределений `<math.h>` содержит описания типов для стандартных математических функций, а файл `<errno.h>` — определения фатальных 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)   /* 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;
}
```

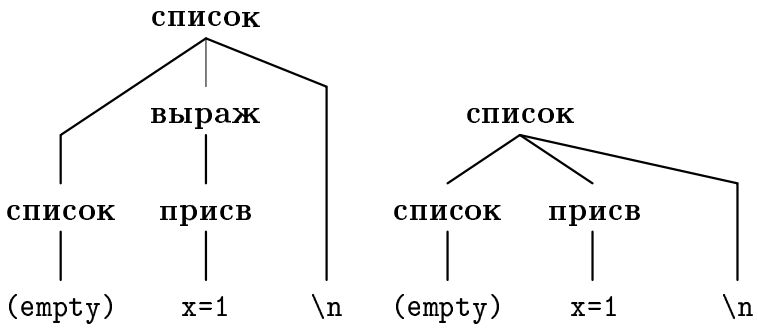
Любопытная, хотя грамматически неясная, диагностики появится при запуске `yacc` с новой грамматикой:

```
$ yacc hoc.y
conflicts: 1 shift/reduce
$
```

Сообщение `shift/reduce` означает, что грамматика `hoc3` неоднозначна: единственная входная строка

```
x=1
```

может быть разобрана двумя способами:



Анализатор может решить, что присв сводится к выраж, а затем к список, как показано в левом дереве разбора, или что нужно применить заключающий символ `\n` сразу (shift — перенос) и преобразовать все в список, не используя промежуточных выводов, как в правом дереве разбора. Встретив неоднозначность, `yacc` выбирает перенос, так как это почти всегда правильное решение для реальных грамматик. Вы должны понимать такие сообщения, чтобы быть уверенным, что `yacc` сделал правильный выбор (Сообщение `yacc`: "shift/reduce conflict" обозначает серьезную проблему и чаще всего является симптомом не неоднозначности, а явной ошибки в грамматике.). Запуск `yacc` с флагом `-v` порождает обширный файл с именем y.output, который поможет вам найти причины конфликтов.

■ Упражнение: В данной версии `hoc3` допустимо присваивание:

```
PI=3
```

Хорошо ли это? Как бы вы изменили `hoc3`, чтобы запретить присваивание "констант"?

■ Упражнение: Добавьте к грамматике встроенную функцию `atan2(x, y)` для вычисления величины угла, тангенс которого равен `x/y`. Добавьте встроенную функцию `rand()`, вырабатывающую случайные вещественные числа, равномерно распределенные на интервале $[0, 1)$. Как бы вам пришлось изменить грамматику, чтобы разрешить встроенные функции с разным числом аргументов?

■ Упражнение: Как ввести дополнительное средство для выполнения команд прямо в `hoc`, подобно операции `!` в программах UNIX?

■ Упражнение: Переработайте текст `math.c` так, чтобы можно было использовать таблицу, а не предложенное выше множество идентичных функций.

Еще одно замечание относительно make. Поскольку теперь программа hoc3 размещается не в одном, а в пяти файлах, makefile становится более сложным:

```
$ cat makefile
YFLAGS = -d             # force creation of y.tab.h
OBJS = hoc.o init.o math.o symbol.o     # abbreviation

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 флаг -d в командную строку запуска yacc, создаваемую make. Этот флаг предписывает yacc создать файл y.tab.h, содержащий операторы #define. Строка OBJS = ... вводит сокращение для записи конструкции, используемой последовательно несколько раз. Синтаксис здесь не такой, как для переменных интерпретатора, — скобки обязательны. Флаг -lm указывает, что математические функции нужно искать в библиотеке libm.a.

Теперь программа hoc3 образуется из четырех файлов .o, причем некоторые из них в свою очередь зависят от файлов .h. "Зная" эти зависимости, make может рассчитать, какая требуется перетрансляция в случае изменения любого из указанных файлов. Если вы хотите выяснить действия make, не запуская процесс, то попробуйте ввести команду

```
$ make -n
```

С другой стороны, если необходимо установить временную согласованность файлов, с помощью флага -t (touch — исправить) вы можете как бы модифицировать файлы, не производя перетрансляции.

Обратите внимание на то, что мы ввели не только множество зависимостей между исходными файлами, но и несколько полезных процедур, сконцентрировав их в одном файле. По умолчанию программа make выполняет первое действие, указанное в файле makefile. Однако если на первом месте окажется элемент, помечающий правило зависимости, такой, как symbol.o или pr, то выполняться будет он. Считается, что в случае "пустой" зависимости элемент всегда берется не из последней версии, поэтому при запросе он обязательно должен изменяться. Итак,

```
$ make pr | lpr
```

инициирует распечатку зависимостей файлов на принтере. (Появление символа @ в "@ pr" подавляет эхо выполняемой команды, запущенной с помощью make.) Команда же

```
make clean
```

удаляет выходные файлы yacc, а также файлы .o.

Такой механизм пустых зависимостей в файле makefile часто оказывается более предпочтительным по сравнению с командным файлом как средство для концентрации в одном файле всех связанных операций. Область применения команды make не ограничивается только разработкой программ, она позволяет сгруппировать в единый набор все операции, имеющие временные зависимости.

Несколько замечаний относительно lex. Программа lex порождает лексические анализаторы аналогично тому, как yacc генерирует программы грамматического разбора: вы создаете описание лексических правил вашего языка с помощью регулярных выражений и фрагментов Си–программ, которые будут выполняться при обнаружении строки, соответствующей шаблону. Программа lex строит по этой информации распознаватель. Программы lex и yacc взаимодействуют таким же образом, как и описанные выше лексические анализаторы. Мы не собираемся здесь детально рассматривать lex; наша цель — заинтересовать вас, а подробности вы найдете в справочном руководстве по UNIX (том 2B).

Вначале приведем lex-программу из файла lex.l, которая заменяет применявшуюся до сих пор функцию yylex:

```
$ cat lex.l
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t]   { ; }   /* skip blanks and tabs */
[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'; }   /* everything else */
.       { return yytext[0]; }
$
```

Каждое “правило” является регулярным выражением, как и те, что использовались в egrep или awk, однако в отличие от них lex распознает комбинации в стиле Си типа \t и \n. Действие заключено в фигурные скобки. Правила проверяются по порядку, а конструкции с символами * и + задают сколь угодно длинную строку. Если правило применимо к текущей части входного потока, то выполняется действие. Совпавшая с правилом входная строка доступна в lex-программе под именем yytext.

Чтобы работать в lex, нужно изменить файл makefile: Программа make

```
$ 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
$
```

"знает", как получить из файла `.l` настоящий файл `.o`; все, что требуется от нас, — дать ей сведения о зависимостях. (Нужно добавить библиотеку `lex -ll` к списку каталогов, в которых ведет поиск команда cc, поскольку распознаватель, создаваемый `lex`, нуждается в дополнительных функциях.) Эффект получается весьма ощутимым, причем совершенно автоматически:

```
$ make
yacc -d hoc.y
conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
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 один файл изменится, достаточно единственной команды `make` для получения действующей версии:

```
$ touch lex.l              Смена времени модификации файла 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
$
```

Некоторое время мы дебатировали о том, следует ли считать обсуждение программы `lex` отступлением от нашей темы и поэтому показать ее кратко, а затем перейти к другим вопросам или рассматривать ее как основное средство для лексического анализа, когда язык становится слишком сложным. У нас были аргументы "за" и "против". Затруднения

в работе с lex (помимо того, что пользователь должен изучить еще один язык) связаны с тем, что замедляется выполнение программы, а распознаватели оказываются более объемными и медленными, чем эквивалентные версии на языке Си. К тому же возникают трудности с механизмом ввода в некоторых особых случаях, таких, как восстановление после ошибки, а также с вводом из файла. Ни одна из перечисленных проблем не является существенной для hoc. К сожалению, из-за ограниченного объема книги мы вынуждены вернуться в последующих лексических анализаторах к Си. Однако создание версии с lex будет для вас хорошей практикой.

■ Упражнение: Сравните размеры двух версий hoc3. Подсказка: обратитесь к справочному руководству по size(1).

8.4 Этап 4: компиляция на машину

Мы постепенно приближаемся к созданию hoc5–интерпретатора языка со структурами управления. Программа hoc4 является промежуточным звеном: она имеет те же операции, что и hoc3, но реализуется на базе интерпретатора, как hoc5. Мы действительно написали такую программу hoc4 и в результате получили две программы с одинаковыми возможностями, что ценно для отладки. По мере разбора входного потока hoc4 порождает код, рассчитанный на простую машину, а не выдает сразу результат. При определении конца оператора будет выполнен код, порожденный для вычисления нужного результата (т. е. произойдет "интерпретация").

Под простой машиной здесь подразумевается стековая машина: когда появляется операнд, он заносится в стек, точнее, создаются команды, заносящие операнд в стек). Большинство операций над операндами выполняется в вершине стека. Например, при обработке присваивания

```
x=2*y
```

создаются следующие команды:

`constpush`	Записать в стек: константа . . . константа2
`2`	
`varpush`	Записать указатель на таблицу имен в стек
`y`	. . . для переменной `y`
`eval`	Вычислить: заменить указатель значением
`mul`	Перемножить два верхних элемента; результат заменяет их
`varpush`	Записать указатель на таблицу имен в стек
`x`	. . . для переменной `x`
`assign`	Записать значение в переменную, убрать указатель
`pop`	Убрать верхний элемент из стека
`STOP`	Конец последовательности команд

Когда выполняются команды, выражение вычисляется и результат записывается в x, как и указано в примечаниях. Последняя команда pop удаляет из стека верхний элемент, поскольку он больше не нужен.

Стековые машины обычно реализуются с помощью простых интерпретаторов, и наш интерпретатор тоже не является исключением: это просто массив, содержащий операции

и операнды. Операции представляют собой машинные команды: каждая из них суть обращение к функции с параметрами, которые следуют за командой. Некоторые операнды могут уже находиться в стеке, как было показано в приведенном выше примере.

Структура таблицы имен для `hoc4` совпадает с таковой для `hoc3`: инициация проводится в `init.c`, и математические функции, находящиеся в `math.c`, одни и те же. Грамматика `hoc4` идентична грамматике `hoc3`, но действия совершенно иные. Вообще, каждое действие порождает машинные команды и все необходимые для них аргументы. Например, в случае появления `VAR` в выражении создаются три команды: команда `varpush`, указатель на таблицу имен для переменной и команда `eval`, которая заменяет при вычислении указатель на таблицу имен соответствующим значением. Код для '`*`' содержит одну команду `mul`, поскольку операнды для нее 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;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
}
%token  <sym>   NUMBER VAR BLTIN UNDEF
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
        | 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); }
```

```
        ;
%%
        /* end of grammar */
...
```

`Inst` является типом данных машинной команды (указатель на функцию, возвращающую `int`), к обсуждению которого мы вскоре вернемся. Обратите внимание на то, что аргументами для программы `code` служат имена функций, т. е. указатели на функции или другие совместимые с ними величины.

Мы несколько изменили процедуру `main`. Теперь происходит возврат из анализатора после выполнения каждого оператора или выражения, и порожденный код выполняется. При обнаружении файла `yyparse` возвращает нуль.

```
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 отличается мало — в основном тем, что числа следует сохранять, а не использовать немедленно. Для этого достаточно занести их в таблицу имен вместе с переменными. Ниже приведена измененная часть `yylex`:

```
yylex()         /* hoc4 */
...
    if (c == '.' || isdigit(c)) {   /* number */
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
...
```

Каждый элемент стека интерпретатора является вещественным значением или указателем на запись в таблице имен; тип данных стека — объединение всех элементов. Сама машина реализуется как массив указателей на процедуры, выполняющие операции типа `mul`, или на данные в таблице имен. Файл макроопределений `hoc.h` увеличивается, поскольку он должен включить эти структуры данных и описания функций для интерпретатора, чтобы они были доступны программе в целом. (Кстати, мы предпочли поместить всю информацию в один файл, а не в два, хотя для больших программ ее целесообразно разделить на несколько файлов с тем, чтобы включать каждый из них только там, где он действительно нужен.)

```
$ cat hoc.h
typedef struct Symbol { /* symbol table entry */
    char    *name;
    short   type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;        /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } 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();

typedef int (*Inst)();  /* machine instruction */
#define STOP    (Inst) 0

extern  Inst prog[];
extern  eval(), add(), sub(), mul(), div(), negate(), power();
extern  assign(), bltin(), varpush(), constpush(), print();
$
```

Процедуры, выполняющие машинные команды и управляющие стеком, хранятся в файле с именем `code.c`. Поскольку содержимое файла составляет около 150 строк, мы покажем его по частям:

```
$ cat code.c
#include "hoc.h"
#include "y.tab.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 */

initcode()      /* initialize for code generation */
{
    stackp = stack;
    progp = prog;
}
...
```

Управление стеком осуществляется путем обращений к двум процедурам push и pop:

```
push(d)         /* push d onto stack */
    Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack overflow", (char *) 0);
    *stackp++ = d;
}

Datum pop()     /* pop and return top elem from stack */
{
    if (stackp <= stack)
        execerror("stack underflow", (char *) 0);
    return *--stackp;
}
```

Машинные команды создаются в процессе разбора при обращении к функции code, которая просто вносит команду на первое свободное место массива prog. Она возвращает адрес команды (который не используется в hoc4):

```
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;
}
```

Выполнение машинной команды фантастически тривиально, а как мала процедура, которая "выполняет" машинные команды, когда уже определены все программы!

```
execute(p)      /* run the machine */
    Inst *p;
{
    for (pc = p; *pc != STOP; )
        (*(*pc++))();
}
```

В цикле выполняется функция, указываемая командой, на которую в свою очередь указывает счетчик команд pc. Значение pc увеличивается, что делает возможным выбор очередной команды. Команда с кодом операции STOP завершает цикл. Некоторые команды, например constpush и varpush, сами увеличивают pc, чтобы "перескочить" через любые аргументы, следующие за командой.

```
constpush()     /* push constant onto stack */
{
```

```
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()       /* push variable onto stack */
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push(d);
}
```

Оставшаяся часть описания машины проста. Так, арифметические операции в основном те же, и создаются они редактированием одного образца. Ниже показана операция add:

```
add()           /* add top two elems on stack */
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}
```

Другие процедуры также просты:

```
eval()          /* evaluate variable on stack */
{
    Datum d;
    d = pop();
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}

assign()        /* assign top value to next value */
{
    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);
}

bltin()         /* evaluate built-in on top of stack */
{
    Datum d;
    d = pop();
    d.val = (*(double (*)())(*pc++))(d.val);
    push(d);
}
```

Самый сложный момент здесь — операция приведения в функции, которая требует, чтобы `*pc` рассматривался как указатель на функцию, возвращающую `double`, и эта функция выполняется с `d.val` в качестве аргумента.

Диагностические сообщения от функций `eval` и `assign` никогда не появятся, если программа работает нормально. Мы оставили их на случай возникновения недоразумений из-за какой-нибудь ошибки программы. Потери за счет увеличения времени выполнения и размера кода даже не так важны, как обнаружение ошибки при внесении необдуманных изменений (что мы и наблюдали несколько раз).

Использование языка Си дает возможность работать с указателем на функцию, что позволяет писать компактные и эффективные программы.

Альтернативное решение состоит в том, чтобы сделать операторы константами и сгруппировать семантические функции в большой переключатель в функции `execute`. Попытайтесь реализовать его в качестве упражнения.

И снова о `make`. По мере увеличения исходного текста программы `hoc` возрастает необходимость механически отслеживать изменения и взаимозависимости. Неоценимую помощь здесь может оказать команда `make`: она автоматизирует процесс, который иначе пришлось бы выполнять вручную (и иногда с ошибками) или создавать для этого специальный командный файл.

Мы сделаем еще две модификации в файле `makefile`. Первая связана с тем, что хотя несколько файлов и зависят от констант, определенных в `yacc`–программе файла `y.tab.h`, нет нужды их перетранслировать, если не изменились сами константы, а изменение в тексте Си–программы из файла `hoc.y` не влияет на другие файлы. В новой версии `makefile` файлы .o зависят от нового файла `x.tab.h`, который изменяется только при замене содержимого файла `y.tab.h`. Вторая модификация основана на том, что правило для `pr` (печать исходных файлов) зависит лишь от самих исходных файлов, а именно, печатаются только измененные файлы.

Первая модификация позволяет существенно экономить время в случае больших программ, когда грамматика постоянна, а семантические действия меняются (обычная ситуация). Второе изменение обеспечивает экономию бумаги.

Приведем `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]
```

Символ '-' перед командой `cmp` дает указание `make` продолжать выполнение даже в случае неудачи `cmp`; это позволяет не останавливать работу и при несуществующем файле `x.tab.h` (флаг `-s` предписывает команде `cmp` не производить вывод, но установить код завершения). Комбинация `$?` раскрывается как список элементов из правила с устаревшей версией. К сожалению, форма записи в `makefile` слабо связана с обозначениями в интерпретаторе.

Проиллюстрируем изложенное выше на примере (в предположении, что все файлы — последней версии):

```
$ touch hoc.y                   Изменим время для файла 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                    Печать измененных файлов
pr hoc.y
touch pr
$
```

Отметим, что, кроме `hoc.y`, файлы не перетранслировались, поскольку файл `y.tab.h` остался тем же.

■ Упражнение: Сделайте размеры стека и массива `prog` динамическими, чтобы для `hoc4` всегда хватало объема памяти, если только ее можно получить, обращаясь к функции `malloc`.

■ Упражнение: Измените `hoc4` так, чтобы использовать в функции `execute` вместо вызова функций переключатель по виду операции `+`. Каково соотношение версий по размеру исходного текста и по времени выполнения? Как приблизительно их сопоставить по сложности развития и поддержания?

8.5 Этап 5: структуры управления и операции отношений

Версия `hoc5` оправдывает все затраты, связанные с созданием интерпретатора. В нее допустимо включать операторы `if-else` и `while`, аналогичные операторам языка Си, группировать операторы с помощью `{` и `}` и использовать оператор `print`. Она содержит полный набор операций отношений (`>`, `>=`, и т. д.), а также операций `AND`, `OR`, `&&` и `||`. (Две последние операции не гарантируют вычисления слева — направо, хотя такой подход принят в Си; вычисляются оба условия, даже если в этом нет необходимости.)

Грамматику `hoc5` дополняют лексемы, нетерминальные символы и правила для `if`, `while`, фигурных скобок и операций отношений. Поэтому она получилась несколько больше, но не намного Сложнее предыдущих версий (возможно, за исключением правил для `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;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
}
%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:     /* nothing */
        | 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;     /* 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); }
        ;
end:      /* nothing */         { code(STOP); $$ = progp; }
        ;
stmtlist: /* nothing */         { $$ = 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); }
        ;
%%
        /* end of grammar */
```

В грамматике есть пять случаев неоднозначности типа сдвиг/свертка, подобных упомянутой в грамматике для `hoc3`.

Обратите внимание на то, что команды `STOP`, завершающие последовательность операторов, теперь порождаются в нескольких местах. Как и прежде, `progp` здесь представляет собой адрес очередной создаваемой команды. При выполнении все команды `STOP` служат для завершения цикла в функции `execute`. Правило для понятия все по сути является подпрограммой, вызываемой из нескольких мест, именно оно порождает команду `STOP` и возвращает адрес следующей за ней команды.

Команды, создаваемые для операторов `if` и `while`, требуют особого рассмотрения. Когда встречается ключевое слово `while`, порождается операция `whilecode`, и адрес этой команды возвращается в качестве значения правила

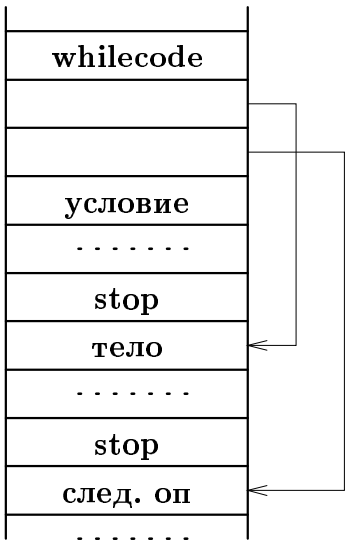
```
пока: WHILE
```

Но в то же самое время резервируются два следующих машинных слова, которые будут определены ниже. Далее создаются команды для выражения, которое образует условие в операторе `while`. Значение, возвращаемое правилом `cond`, является адресом начала команд для условия. После распознавания всего оператора `while` в два зарезервированных слова, вводимых за командой `whilecode`, заносится адрес начала тела цикла и адрес оператора, следующего за циклом. (Команда по этому адресу будет создана позднее.)

```
| пока усл опер все {
    ($1) [1] = (Inst) $ 3 /* тело цикла */
    ($1) [2] = (Inst) $ 4 /* все, если условие неверно */
```

Здесь `$1` обозначает адрес команды `whilecode`, таким образом, ($1) [1] и ($1) [2] обозначают два следующих слова.

Рисунок, приведенный ниже, может прояснить ситуацию:



С оператором `if` дело обстоит аналогично, но резервируются три слова: для частей `then` и `else`, а также для оператора, следующего за `if`. Мы вскоре рассмотрим этот случай.

Лексический анализ теперь несколько удлиняется — в основном из-за необходимости распознавания дополнительных операций:

```
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 возвращает символ назад во входной поток с помощью `ungetc`, если он оказался не тем, который требовался:

```
follow(expect, ifyes, ifno)  /* look ahead for >=, etc. */
{
    int c = getchar();

    if (c == expect)
        return ifyes;
    ungetc(c, stdin);
    return ifno;
}
```

В файле `hoc.h` стало больше описаний функций, например всех отношений, но в общем его структура такая же, как и в `hoc4`. Ниже приведено несколько последних строк грамматики:

```
$ cat hoc.h
...
typedef int (*Inst)();  /* machine instruction */
#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 файла code.c также не изменилась, хотя, очевидно, здесь появилось много новых процедур для обработки операций отношений. Типичным примером может служить функция le ("less than equal to" — меньше или равно).

```
le()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}
```

Не совсем тривиальными являются функции whilecode и ifcode. Чтобы понять их, необходимо уяснить, что функция execute последовательно обрабатывает команды до тех пор, пока не будет найдена команда STOP, после чего происходит возврат из execute. Процесс разбора построен таким образом, что команда STOP завершает каждую последовательность команд, которую нужно обработать за одно обращение к execute. Тело цикла while, а также условие и фрагменты оператора if после then и else обрабатываются с помощью рекурсивных обращений к execute, возврат из которых по завершении обработки осуществляется в функцию execute на один уровень вложенности выше. Управление этими рекурсивными обращениями обеспечивается в whilecode и ifcode. Последние и предназначены для обработки соответствующих операторов.

```
whilecode()
{
    Datum d;
    Inst *savepc = pc;      /* loop body */

    execute(savepc+2);      /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc)));  /* body */
        execute(savepc+2);
        d = pop();
    }
    pc = *((Inst **)(savepc+1));  /* next statement */
}
```

Как уже отмечалось ранее, после операции whilecode размещается указатель на тело цикла, затем указатель на следующий оператор, а за ним команды условия. Когда вызывается whilecode, значение указателя pc уже увеличено, так что он содержит указатель на тело цикла. Таким образом, pc+1 настроен на следующий оператор, а pc+2 — на команды условия.

Функция ifcode аналогична предыдущей: при входе в нее pc ссылается на фрагмент после then, pc+1 — на фрагмент после else, pc+2 — на следующий оператор, а pc+3 — на условие.

```
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)));
    pc = *((Inst **)(savepc+2));     /* next stmt */
}
```

Программа в файле `init.c` также немного увеличится за счет введения в нее таблицы ключевых слов, хранимых в таблице имен вместе с остальной информацией:

```
$ cat init.c
...
static struct {         /* Keywords */
    char    *name;
    int     kval;
} keywords[] = {
    "if",       IF,
    "else",     ELSE,
    "while",    WHILE,
    "print",    PRINT,
    0,          0,
};
...
```

Для занесения в таблицу имен ключевых слов нужно организовать еще один цикл в функции `init`:

```
...
    for (i = 0; keywords[i].name; i++)
        install(keywords[i].name, keywords[i].kval, 0.0);
...
```

Изменения в функциях, управляющих таблицей имен, не требуются; в файле `code.c` есть функция `prexpr`, которая вызывается при выполнении оператора вида `print` выраж.

```
prexpr()        /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g\n", d.val);
}
```

Это не та функция печати, которая автоматически вызывается для вывода окончательного результата вычислений. Здесь выбирается число из стека и добавляется символ перевода строки к выходному потоку.

Теперь `hoc5` представляет собой вполне полезный калькулятор, хотя для серьезного программирования необходимы дополнительные средства. В приведенных ниже упражнениях предлагаются возможные решения.

■ Упражнение: Добавьте для отладки к `hoc5` средство печати создаваемых машинных команд в понятной форме.

■ Упражнение: Введите в свою программу операции присваивания из языка Си вида `+=`, `*=` и т. п., а также операции инкремента и декремента `++` и `--`. Измените операции `&&` и `||` так, чтобы обеспечить вычисление слева направо и условное вычисление, как в Си–программах.

■ Упражнение: Введите в `hoc5` оператор `for`, как в Си–программах. Добавьте операторы `break` и `continue`.

■ Упражнение: Как бы вы изменили грамматику или лексический анализатор `hoc5` (или и то, и другое), чтобы сделать программу более "терпимой" к использованию символов перевода строки? Каким образом можно ввести символ в качестве синонима символа перевода строки? Как ввести в язык примечания? Какой синтаксис, по вашему мнению, нужно использовать?

■ Упражнение: Добавьте к `hoc5` средства обработки прерываний, чтобы некорректные вычисления можно было остановить без потери значений уже вычисленных переменных.

■ Упражнение: Неудобно создавать программный файл, запускать его на выполнение, а затем редактировать с целью внесения небольших изменений. Как бы вы изменили `hoc5`, чтобы создать команду редактирования, которая автоматически вызывала бы редактор с уже считанной копией вашей `hoc`–программы? Подсказка: изучите текст функции.

8.6 Этап 6: функции и процедуры; ввод–вывод

На последнем из описываемых здесь этапе развития программа значительно разрастается: в нее добавляются процедуры и функции, средства печати строк символов наряду с числами и чтения чисел из стандартного входного потока. Кроме того, в язык `hoc6` вводятся аргументы имен файлов, включая имя "`-`", обозначающее стандартный входной поток. Все эти изменения увеличивают программу на 235 строк, доводя ее общий размер до 810 строк. В результате `hoc` преобразуется из калькулятора в интерпретатор языка программирования. Полностью программа приводится в приложении 3.

В грамматике вызовы функции определяются как выражения, а вызовы процедур — как операторы. И то, и другое детально поясняется в приложении 2, где дается еще несколько примеров. Так, определение и использование процедуры печати всех чисел Фибоначчи, меньших заданного параметра, происходят следующим образом:

```
$ 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
...
```

Здесь также показано использование файлов: имя файла "`-`" задает стандартный входной поток.

Ниже приведена функция "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 процедуры или функции к параметрам можно обращаться с помощью `$1` и т. д., как в командных файлах, но, кроме того, допустимо присваивание параметрам. Функции и процедуры рекурсивны, но в качестве локальных переменных можно использовать только параметры; остальные переменные являются глобальными, т. е. доступными во всей программе.

В языке `hoc` функции и процедуры различаются, что дает возможность проверки, ценной для освобождения стека. (Ведь так легко забыть выполнить возврат или записать лишнее выражение и получить несбалансированный стек!)

Требуется значительное число изменений для преобразования грамматики при переходе от `hoc5` к `hoc6`, но все они локальные. Нужны новые лексемы и нетерминальные символы, а в описание `%union` необходимо ввести новый элемент для хранения числа аргументов:

```
$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;   /* 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 */
...
```

С помощью правила для аргсписок (список агументов) подсчитывается число аргументов. На первый взгляд может показаться, что нужно каким-то образом собирать аргументы, но это не так, поскольку каждое выражение (выраж) из списка аргументов вырабатывает значение в стеке как раз там, где оно необходимо.

Правило для опред вводит новое свойство языка `yacc`: встроенное действие. Оказывается, можно поместить действие посредине правила, так, чтобы оно выполнялось в процессе распознавания последнего. Мы воспользовались этой возможностью, чтобы запомнить, что сейчас распознается: определение функции или процедуры. (В качестве альтернативного решения можно было бы ввести новый символ типа `begin`, который распознавался бы в соответствующее время.) Функция `defnonly` печатает предупреждающее сообщение, если вопреки синтаксису какая-либо конструкция окажется вне определения функции или процедуры. Обычно вам предоставляется выбор: обнаруживать ошибку синтаксически или семантически. Перед нами уже стояла такая задача ранее, при диагностике неопределенных переменных. Функция `defnonly` хорошо иллюстрирует ситуацию, когда семантическая проверка легче синтаксической.

```
defnonly(s)     /* warn if illegal definition */
    char *s;
{
    if (!indef)
        execerror(s, "used outside definition");
}
```

Переменная `indef` определена в `hoc.y` и принимает значения в действиях для опред.

К лексическому анализатору добавлены средства проверки аргументов: символ `$`, за которым следует число для строки в кавычках. Последовательности в строках, начинающиеся с обратной дробной черты, например `\n`, обрабатываются функцией `backslash`:

```
yylex()         /* hoc6 */
...
    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;
    }
...

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;
}
```

Лексический анализатор является примером конечного автомата независимо от того, написан ли он на Си или получен с помощью порождающей программы типа `lex`. Наша первоначальная версия Си–программы стала весьма сложной, и поэтому для всех программ, больших ее по объему, лучше использовать `lex`, чтобы максимально упростить внесение изменений.

Остальные изменения сосредоточены главным образом в файле code.c, хотя несколько имен функций добавляется к файлу hoc.h. Машина остается той же, но с дополнительным стеком для хранения последовательности вложенных вызовов функций и процедур (проще ввести второй стек, чем загружать больший объем информации в существующий). Начало файла code.c выглядит так:

```
$ cat 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;
}
...
$
```

Поскольку теперь в таблице имен хранятся указатели на функции и процедуры, а также на строки для печати, необходимо расширить определение типа объединения в файле hoc.h:

```
$ cat 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;
...
$
```

В процессе трансляции функция `define` заносит запись о функции в таблицу имен, сохраняет указатель на нее и изменяет в случае успешной компиляции адрес следующего после созданных команд свободного слова:

```
define(sp)      /* put func/proc in symbol table */
    Symbol *sp;
{
    sp->u.defn = (Inst)progbase;    /* start of code */
    progbase = progp;       /* next code starts here */
}
```

Когда в процессе выполнения вызывается функция или процедура, все аргументы уже вычислены и помещены в стек (первый аргумент находится на наибольшем уровне). Код операции вызова (`call`) сопровождается указателем на таблицу имен и числом аргументов. Сохраняется образ стека, в котором содержится вся существенная информация о программе: запись в таблице имен, место возврата после вызова, место хранения аргументов в стеке выражений, а также число аргументов, сопровождающих вызов. Образ стека создается функцией `call`, которая затем выполняет тело программы.

```
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;
}
```

Создаваемая структура показана на рис. 8.2.

В конце концов произойдет возврат из вызываемой программы при выполнении `procret` или `funcret`:

```
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();
}
```

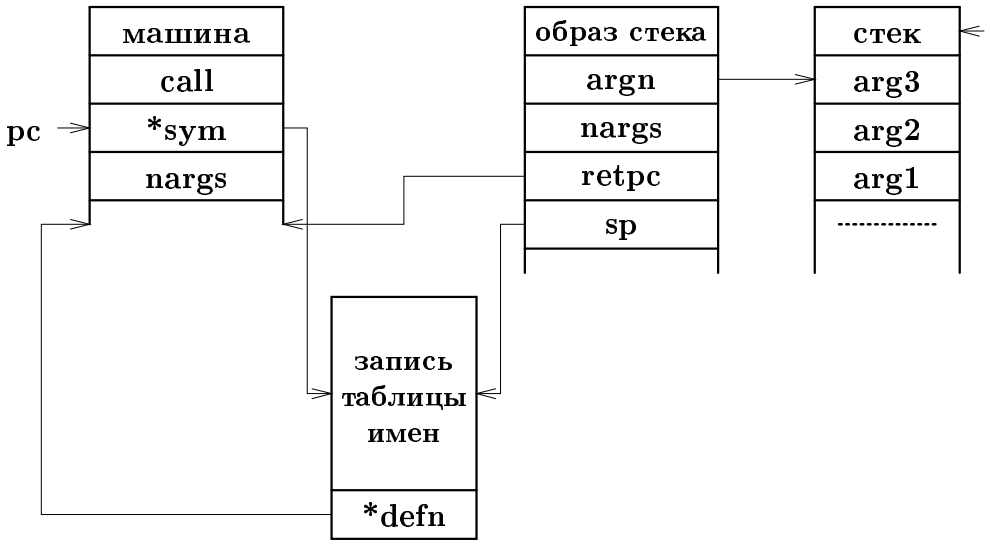


Рис. 8.2: Структуры данных для вызова процедуры

Функция `ret` удаляет аргументы из стека, сохраняет указатель на образ стека `fp` и устанавливает счетчик команд:

```
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;
}
```

Некоторые программы интерпретатора нуждаются в небольших поправках для учета ситуаций, когда происходит возврат во вложенных операторах. Решение не элегантно, но

верно и состоит во введении признака с именем `returning`, который принимает значение 1 при обнаружении оператора `return`. Выполнение, организуемое функциями `ifcode`, `whilecode`, `execute`, завершается раньше, если установлен признак `returning`; в функции `call` он обнуляется.

```
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 */
}

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 */
}

execute(p)
    Inst *p;
{
    for (pc = p; *pc != STOP && !returning; )
        (*((++pc)[-1]))();
}
```

Аргументы выбираются для получения значения или присваивания с помощью функции `getarg`, которая следит за, сбалансированностью стека:

```
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;
}
```

Функции `prstr` и `prexpr` печатают строки и числа:

```
prstr()         /* print string value */
{
    printf("%s", (char *) *pc++);
}

prexpr()        /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g ", d.val);
}
```

Функция `varread` читает переменные. Она возвращает 0 при обнаружении конца файла и 1 — в противном случае, а также устанавливает значение указанной переменной:

```
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);
}
```

Обнаружив конец файла для текущего входного потока, функция `varread` обратится к `moreinput`, которая откроет следующий файл, заданный в качестве аргумента (если он есть). В функции `moreinput` обработка входной информации имеет некоторые нюансы, здесь не рассматриваемые; речь о них идет в приложении 3.

Итак, мы завершили разработку программы `hoc`. Для сравнения приведем число непустых строк в каждой версии:

`hoc1`	59
`hoc2`	94
hoc3	248 (для версии с `lex` 229)
`hoc4`	396
`hoc5`	574
`hoc6`	809

Конечно, эти значения были вычислены программным способом:

```
$ sed '/^$/d' `pick *.[chyl` | wc -l
```

Безусловно, развитие языка может быть продолжено, и вам предоставляется такая возможность в приведенных ниже упражнениях.

■ Упражнение: Измените `hoc6` так, чтобы можно было использовать поименованные формальные параметры в подпрограммах вместо `$1` и т. д.

■ Упражнение: Сейчас все переменные глобальны, за исключением параметров. Уже есть большая часть механизма для введения локальных переменных, хранимых в стеке. Одно из решений заключается во введении описания `auto`, которое резервирует место в стеке для перечисленных переменных; не перечисленные переменные считаются глобальными. Кроме того, придется расширить таблицу имен так, чтобы поиск в ней осуществлялся вначале для локальных, а затем для глобальных переменных. Как это связано с поименованными аргументами?

■ Упражнение: Как бы вы ввели массивы в язык `hoc`? Как следует передавать их функциям и процедурам? Как возвращать их?

■ Упражнение: Обобщите работу со строками так, чтобы переменные могли хранить строки, а не только числа. Какие операции потребуются для этого? Самая трудная часть — управление памятью — добейтесь динамичного хранения строк: память должна освобождаться, когда строки перестают быть нужными. В качестве промежуточного шага добавьте более развитые форматы печати, например, обеспечьте возможность использования некоторых форм стандартной Си–функции `printf`.

8.7 Оценка времени выполнения

Мы сравнивали `hoc` с другими программами-калькуляторами UNIX, чтобы приблизительно оценить, насколько хорошо он работает. К таблице, представленной ниже (табл. 8.1), можно, конечно, отнестись скептически, но она показывает "разумность" нашей реализации. Все приведенные в ней величины даны в секундах. Работа велась на **PDP-11/70**. Было выполнено два теста. Первый, вычисление функции Аккерманна `ack(3,3)`, — хороший тест для отработки механизма вызова функций. Здесь происходят 2432 вызова, причем некоторые из них достаточно глубоко вложены.

```
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)
```

Второй тест — стократное вычисление чисел Фибоначчи со значениями, меньшими 1000. В этом случае выполнялись в основном арифметические операции с периодическим вызовом функций:

```
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
}
```

Тест выполнялся на четырех языках: `hoc`, `bc(1)`, `bas` (древний диалект Бейсика, который существует только на **PDP-11**) и Си (использовался тип **PDP-11** для всех переменных).

Числа, приведенные в табл. 8.1, являются суммой пользовательского и системного времени процессора и вычислены с помощью функции `time`.

Программа	**(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

Таблица 8.1: Время работы на **PDP-11/70** (в секундах)

Можно также приспособить Си–программу для определения количества времени, используемого каждой функцией. Программу нужно перетранслировать в режиме профилирования, введя флаг `-p` в каждой единице трансляции Си и при режиме загрузки. Если изменить файл `makefile` для чтения:

```
hoc6: $(OBJS)
    cc $(CFLAGS) $(OBJS) -lm -o hoc6
```

чтобы команда cc задействовала переменную `CFLAGS`, а затем задать

```
$ make clean; make CFLAGS=-p
```

то результирующая программа будет выполняться с профилированием. После выполнения программы остается файл `mon.out`, который интерпретируется программой–профилировщиком `prof`.

Для иллюстрации изложенного мы протестировали `hoc6` на приведенной выше программе Фибоначчи:

```
$ hoc6 <fibtest                          Запуск теста
$ prof hoc6 | sed 15q                    Анализ
name     %time cumsec #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
$
```

Результаты, полученные с помощью профилировщика, также подвержены случайным вариациям, как и те, что получены с помощью функции `time`, поэтому их следует считать лишь указанием настоящих значений, а не принимать за абсолютную истину. Тем не менее при необходимости приведенные значения могут помочь повысить быстродействие программы `hoc`. Приблизительно третья часть времени тратится на запись и чтение из стека. Накладные расходы еще более возрастут, если мы будем учитывать время выполнения функций связи `csv` и `cret` между программами Си (функция `mcount` представляет собой часть программы с профилированием, полученную с помощью команды ее `-p`.). Замена вызовов функций на макрообращения даст заметную разницу во времени выполнения.

Для проверки этого предположения мы изменили `code.c`, заменив вызовы `push` и `pop` на макрокоманды, управляющие стеком:

```
#define push(d) *stackp++ = (d)
#define popm()  *--stackp = pop() /* функция все-таки нужна */
```

(Функция `pop` все-таки нужна в качестве кода операции нашей машины, поэтому нельзя заменить все обращения к ней.) Новая версия выполняется на 35% быстрее; время в табл. 8.1 сокращается от 5.5 до 3.7 с и от 5.0 до 3.1 с.

■ Упражнение: В макрокомандах `push` и `popm` не предусмотрен контроль ошибок. Прокомментируйте разумность такого решения. Как бы вы обеспечили контроль ошибок, производимый в версии с функциями, не снижая быстродействия макрокоманд?

8.8 Заключение

Ознакомившись с материалом этой главы, мы можем сделать важные выводы. Во-первых, средства для развития языков очень нужны, так как позволяют сконцентрировать внимание на интересной работе — проектировании языка (с ним легко экспериментировать). Грамматика является организующей структурой при реализации: программы привязываются к грамматике и вызываются в подходящий момент в процессе разбора.

Во-вторых, и это уже философский аспект, ценна сама постановка задачи — речь идет о разработке языка, а не просто о написании программы. Построение программы как языкового процессора обеспечивает регулярность синтаксиса (т. е. взаимодействие с пользователем), делает более структурированной реализацию. Кроме того, мы получаем гарантию, что новые средства будут хорошо согласовываться с уже реализованными. Под "языками", конечно, следует понимать не только традиционные языки программирования, но и уже упоминавшиеся выше в примерах языки `eqn` и `pick`, а также `yacc`, `lex`, и `make`.

Рассмотрены здесь и вопросы использования программных средств. В частности, показана роль программы `make`, которая предотвращает целый класс ошибок (например, вы забыли перетранслировать какую-то подпрограмму). Она позволяет избавиться от лишней работы и предоставляет удобный способ сгруппировать в одном файле большое число связанных и, возможно, зависимых операций.

С помощью файлов макроопределений вы можете координировать описания данных, доступных более чем в одном файле. Проводя централизацию информации, они исключают ошибки, вызванные несогласованностью применяемых версий, особенно если

действуют совместно с программой make. Весьма важно разбить данные и подпрограммы на файлы таким образом, чтобы они не были видимы, если в этом нет необходимости.

Хотелось бы отметить, что из-за ограниченного объема книги мы мало внимания уделили тем средствам UNIX, которые применяются при разработке семейства программ hoc. Каждая версия программы находится в отдельном каталоге, для идентичных файлов используются связи; постоянно вызываются команды ls и du, чтобы следить за тем, какие файлы где находятся. На многие вопросы ответы дают сами программы. Например, на вопрос: "Где описана данная переменная?" отвечает программа grep. "Как мы внесли изменения в данную версию?" — отвечает idiff. "Насколько велик файл?" — отвечает wc. Пора делать копию файла — обратитесь к команде cp. Нужно скопировать только те файлы, которые изменились со времени последнего копирования? Вам поможет в этом деле программа make.

Такой общий подход типичен для повседневной разработки программ в системе UNIX: множество небольших программных средств, каждое в отдельности или их различные сочетания, позволяет автоматизировать работу, которую иначе пришлось бы выполнять вручную.

Историческая и библиографическая справка. Программа yacc создана С. Джонсоном. Класс языков, для которых yacc может создавать программу разбора, называется LALR-(1): разбор здесь ведется слева направо и входной поток просматривается не более чем на одну лексему вперед. Понятие раздельных описаний для задания приоритетов и разрешения неоднозначностей в грамматике появилось вместе с yacc. Этот вопрос рассматривается в статье А. В. Ахо, С. К. Джонсона и Д. Д. Ульмана "Determenistic parsing of ambiguous grammars" (CACM, August, 1975). Там же приведены новые алгоритмы и структуры данных для создания и хранения таблиц разбора.

Основы теории, на базе которой построены yacc и другие программы–анализаторы, излагаются в книге А. В. Ахо и Д. Д. Ульмана "Principles of Compiler Design" (Addison — Wesley, 1977). Сама программа yacc описана в справочном руководстве по UNIX (том 2B). В этом разделе также представлен калькулятор, сравнимый с hoc2: для вас такое сравнение может оказаться полезным.

Программа lex первоначально было написана М. Леском. Теория lex освещена в книге Ахо и Ульмана, а сам язык lex описан в справочном руководстве по UNIX.

Программы yacc и в меньшей степени lex использовались для реализации многих языковых процессоров, включая переносимый компилятор Си, процессоры на Паскале, Фортране 77, Ратфоре, awk, be, egn и pick.

Программа make создана С. Фельдманом и описана в статье "MAKE — a program for maintaining computer programs" (Software-Practice & Experience, April, 1979).

В книге Д. Бентли "Writing Efficient Programs" (Prentice-Hall, 1982) обсуждается техника ускорения выполнения программ. Акцент в ней делается на создание подходящего алгоритма, а также на улучшение кода, если в этом есть необходимость.

Глава 9

Подготовка документации

Средства редактирования и форматирования документации стали одним из ранних приложений системы UNIX. Администрация Bell Labs, приобретая первую установку **PDP-11**, хотела иметь систему подготовки документации, а не операционную систему. (К счастью, заказчики получили больше, чем ожидали.)

Первая форматирующая программа `roff` — небольшая, быстрая программа, с которой легко было работать до тех пор, пока выпускались простые документы с помощью печатающего устройства. Следующая форматирующая программа `nroff` обладала большими возможностями. Ее автор Дж. Осанна вместо того, чтобы обеспечивать каждый вид документации по желанию пользователя, сделал свою программу программируемой, т. е. многие задачи форматирования решались путем программирования на языке `nroff`.

После приобретения в 1973 г. небольшого наборного устройства `nroff` была расширена и смогла поддерживать множество символов, а также шрифты разных кеглей и гарнитур. Новая программа получила название `troff` (по аналогии с “en–roff”, а произносили это как “tee–roff”.) Программы `nroff` и `troff` имели одну и ту же основу и один и тот же входной язык; `nroff` игнорировала команды типа смены размера шрифта, которые она не могла выполнить. В настоящей главе речь пойдет главным образом о `troff`, но большинство наших пояснений можно отнести и к `nroff` с учетом ограничений, налагаемых выводными устройствами.

Самое большое достоинство `troff` — гибкость базового языка и ее возможности программирования. Она может выполнить почти любую задачу форматирования. Но гибкость обходится недешево, использовать `troff` часто бывает весьма трудно. Следует отметить, что почти все программное обеспечение UNIX для подготовки документации сконструировано таким образом, чтобы заместить некоторые фрагменты “чистой `troff`”. Одним из примеров может служить разметка страницы: общий вид документа, как выглядят названия, заголовки и абзацы, где появляются номера страниц, как велики размеры страницы и т. д. Все соответствующие средства не встроены и должны программироваться. Однако пользователю не нужно специфицировать перечисленные выше детали для каждого документа, так как есть стандартный пакет форматирующих команд. Пользователь пакета не говорит: “Следующая строка должна центрироваться, состоять из прописных букв, выделенных жирным шрифтом”. Вместо этого он должен сказать: “Следующая строка - заголовок”, и тогда используется определение вида названия из пакета. Пользователи имеют дело с логическими компонентами документа — названиями, заголовками, абзацами, сносками и т. п., а не с размерами, шрифтами и позициями.

К сожалению, планируемый как “стандартный” пакет команд форматирования не является более стандартным: распространение получили несколько пакетов плюс многие

их варианты. Мы рассмотрим здесь два пакета общего назначения: `ms`, первоначальный "стандарт", и `mm`, более новую версию, стандартную в System V, а также пакет `man` для печати справочников. Основной акцент сделан на `ms` как на стандарт в седьмой версии; он может служить хорошим примером и является достаточно мощным для выполнения работы (с его помощью подготовлена наша книга, правда, пришлось добавить команду для задания требуемого нам шрифта).

Этот опыт типичен — пакеты макроопределений подходят для выполнения многих задач форматирования, но иногда необходимо вернуться к основным командам `troff`. В данной главе мы опишем лишь небольшую часть `troff`.

Хотя `troff` обеспечивает возможность полностью управлять выходным форматом, ее непросто использовать для набора более сложного материала, такого, как формулы, таблицы и рисунки. Их разметка так же трудна, как разметка страницы. Решить упомянутые выше проблемы можно по-разному. Существуют языки специального назначения для формул, таблиц и рисунков, которые заменяют пакеты команд форматирования. Каждый язык поддерживается отдельной пограммой, транслирующей его в команды `troff`. Программы взаимодействуют через программные каналы. Эти препроцессоры служат хорошим примером подхода к проблемам в UNIX: вместо расширения и усложнения `troff` с ней взаимодействуют отдельные программы. (Конечно, средства развития языка, описанные в гл. 8, были использованы для облегчения реализации.) Ниже описываются две программы: `tbl`, форматирующая таблицы, и `eqn`, форматирующая математические выражения.

Мы постараемся дать вам советы по подготовке документации и применению поддерживающих инструментов. Приводимые здесь примеры войдут в документ, описывающий язык `hoc`, из гл. 8 и справочного руководства. Сам документ приведен в приложении 2.

9.1 Пакет макроопределений `ms`

Основная идея, заложенная в пакет макроопределений, состоит в том, что документ описывается в терминах его логических частей — названия, заголовков разделов, абзацев, а не в деталях: расстановка пробелов, выбор шрифтов, определение размеров букв. Это спасает вас от рутинной работы и освобождает документ от несущественных подробностей. Фактически вы можете сделать свой документ совершенно иным, использовав другое множество макроопределений с теми же логическими именами. Так, один и тот же документ мог бы последовательно превращаться в технический отчет, доклад на конференции, журнальную статью и главу из книги с помощью одинаковых команд форматирования, интерпретируемых четырьмя разными пакетами макроопределений.

Входной поток `troff` представляет собой обычный текст, размеченный командами форматирования, независимо от использования пакета макроопределений. Существуют два вида команд. Команда первого вида состоит из точки в начале строки и следующей за ней одной (двух) буквы либо цифры и, возможно, параметра, как показано ниже:

```
.PP
.ft B
Это небольшой абзац, выделенный жирным шрифтом.
```

Все встроенные команды `troff` имеют имена, образованные строчными буквами, поэтому по соглашению командам в пакетах даются имена из прописных букв. В нашем примере `.PP` есть команда `ms` для абзаца, а `.ff` — команда `troff`, вызывающая замену

обычного шрифта жирным. (Имена команд, управляющих шрифтами, состоят из прописных букв; шрифты могут быть разными на разных наборных устройствах.)

Команда второго вида — это последовательность символов, начинающаяся с обратной дробной черты \, которая может оказаться в любом месте входного потока. Например, команда \fB также вызывает переключение на жирный шрифт. Она является примером чистой troff; ниже мы коротко ее рассмотрим.

Для форматирования достаточно вводить команду .PP перед каждым абзацем. В принципе при подготовке большинства документов можно обойтись дюжиной разных команд ms. Так, приложение 2, описывающее hoc, имеет название, имена авторов, резюме, автоматически нумеруемые заголовки разделов и абзацы. Здесь используется всего 14 различных команд, причем некоторые из них — парами. В ms документ принимает следующую общую форму:

```
.TL
Название документа (одна или более строк)
.AU
Имена авторов, одно на строке
.AB
Резюме, оканчивающееся .AE
.AE
.NH
Нумеруемые заголовки (автоматически)
.PP
Абзац...
.PP
Другой абзац...
.SH
Подзаголовок (нет нумерации)
.PP
...
```

Команды форматирования могут появиться в начале строки. Входной текст между командами размещается в произвольной форме: расположение концов строк в нем не существенно, так как troff переносит слова с одной строки на другую, чтобы сделать строки более полными (процесс, называемый заполнением), и расставляет дополнительные пробелы между словами, чтобы выравнять границы (выравнивание). Тем не менее начинать каждое предложение с новой строки считается хорошим стилем; это облегчает последующее редактирование.

Ниже показано начало непосредственного документа hoc:

```
.TL
Hoc - диалоговый язык для арифметики
с плавающей точкой
.AU
Брайан Керниган
Роб Пайк
.AB
.I Hoc
это простой программируемый интерпретатор
```

```
для выражений с плавающей точкой.
Он обеспечивает поток управления в стиле Си,
определения функций и обычные
числовые встроенные функции,
такие как косинус и логарифм.
.AE
.NH
Выражения
.PP
.I Hoc
Это язык выражений,
во многом подобный Си:
хотя он содержит несколько управляющих операторов,
большинство операторов, таких как присваивания,
это выражения, чьи значения не принимаются
во внимание.
...
```

Команда `.I` выделяет свой аргумент курсивом или переключает шрифт на курсив, если аргумента нет.

При использовании пакета макроопределений он специфицируется как аргумент `troff`:

```
$ troff -ms hoc.ms
```

Символы после `-m` определяют пакет макроопределений [1]. После форматирования с помощью `ms` документ `hoc` выглядит так:

```
Hoc - диалоговый язык для арифметики с плавающей точкой

                              Брайан Керниган
                                 Роб Пайк
                                  РЕЗЮМЕ

       Hoc - это простой программируемый интерпретатор для выражений с
       плавающей точкой. Он обеспечивает поток управления в стиле Си,
       определения функций и обычные числовые встроенные функции, такие
       как косинус и логарифм.

1. Выражения

    Hoc - это язык выражений, во многом подобный Си; хотя он содержит
несколько управляющих операторов, большинство операторов, таких как
присваивания, это выражения, чьи значения не принимаются во внимание.
```

Отображения. Хотя нам и удобно, что `troff` осуществляет заполнение и выравнивание текста, иногда нежелательно, в частности для программ, выравнивать их границы. Такое

[1] Макроопределения `ms` находятся в файле `/usr/lib/tmac/tmac.s`, а макроопределения `man` — в файле `/usr/lib/tmac/tmac.an`

форматированное представление называется отображением текста. Команды ms .DS (начало отображения) и .DE (конец отображения) ограничивают текст, благодаря чему он выводится с отступами, но без реорганизации. Посмотрите на следующий фрагмент руководства по hoc, включающий короткое отображение:

```
.pp
.I Hoc
Это язык выражений, во многом подобный Си:
хотя он содержит несколько управляющих операторов,
большинство операторов, таких как присваивания,
это выражения, чьи значения не принимаются
во внимание.
Например, оператор присваивания
= присваивает значение его правой части
его левому операнду и вырабатывает значение,
используемое в многократном присваивании.
Грамматика выражений такова:
.DS
.I
выражение: число
    | переменная
    | (выражение)
    | выражение бинарная-операция выражение
    | унарная-операция выражение
    | функция(аргументы)
.R
.DE
Числа представляются с плавающей точкой.
```

Данный фрагмент печатается так:

```
    Hoc - это язык выражений, во многом подобный Си; хотя он содержит
несколько управляющих операторов, большинство операторов, таких как
присваивания, - это выражения, чьи значения не принимаются во внимание.
Например, оператор присваивания = присваивает значение своей правой части
левому операнду и вырабатывает значение, используемое в многократном
присваивании. Грамматика выражений такова:

    выражение: число
        |переменная
        |(выражение)
        |выражение бинарная_операция выражение
        |унарная_операция выражение
        |функция (аргументы)

Числа представляются с плавающей точкой.
```

Текст внутри отображения не является ни нормально заполненным, ни выравненным. Далее, если места на текущей странице не хватает, отображаемый материал (и все, что за ним следует) переносится на следующую страницу. Команда .DS обеспечивает несколько

флагов, включая L для левого выравнивания, C для индивидуальной центровки каждой строки и B для центровки всего отображения.

Фрагменты текста в приведенном выше отображении разделены символами табуляции. По умолчанию символы табуляции troff ставятся через каждые полдюйма, а не через восемь пробелов, как обычно. Даже если бы эти символы вставлялись через каждые восемь пробелов, символы табуляции, обрабатываемые troff, не всегда бы появлялись там, где нужно из-за их переменной ширины.

Смена шрифта. Макроопределения ms обеспечивают три команды смены шрифта. Команда .R меняет шрифт на латинский (обычный), .I устанавливает курсив, а .B — жирный шрифт:

```
Этот текст обычный,
.I
это курсив
.R
это снова обычный, а
.B
это жирный шрифт.
```

При выполнении команд приведенный выше текст превращается в следующий:

Этот текст обычный, *это курсив*, это снова обычный, **а это жирный шрифт.**

Команды .I и .B воспринимают возможные аргументы, причем смена шрифта относится только к аргументу. В troff аргументы, содержащие пробелы, должны быть экранированы, и единственным символом для такой операции служит двойная кавычка ".

```
Это обычный текст, но
.I - это
курсив, а
.B "эти слова"
напечатаны жирным шрифтом.
```

Данный текст печатается так:

Это обычный текст, но — это курсив, а **эти слова** напечатаны жирным шрифтом.

В конечном счете второй аргумент для .I или .B, напечатанный обычным шрифтом, добавляется без пробелов к первому аргументу. Это средство широко используется при выборе шрифта для пунктуации. Обратите внимание на последнюю скобку фразы

```
(взятые в скобки
.I "слова курсивом)"
```

которая печатается неверно в виде

(взятые в скобки *слова курсивом)*

и сравните ее с фразой

```
(взятые в скобки
.I "слова курсивом")
```

которая печатается верно как

(взятые в скобки *слова курсивом*)

Различные шрифты распознаются программой `nroff`, но результат оставляет желать лучшего. Символы курсива подчеркнуты и нет жирных литер, хотя некоторые версии `nroff` изображают жирный шрифт двойной печатью.

Смешанные команды. Сноски вводятся с помощью `.FS` и заканчиваются `.FE`. Ваше дело — определить метку (сноску) в виде звездочки `*` или крестика `+`. Такая сноска создается следующим образом:

```
определяющая метка типа звездочки или крестика. \(dg
.FS
\(dg подобно этому
.FE
Эта сноска была создана с помощью ...
```

Выделенные отступом абзацы, возможно с использованием номера или другой пометки на границе, создаются командой `.IP`. Сделаем следующее:

1. Первый небольшой абзац.
2. Второй абзац, который мы удлиняем, чтобы показать, что отступ во второй строке будет таким же, как в первой.

Для этого нужен такой входной текст:

```
.IP(1)
Первый небольшой абзац.
.IP(2)
Второй абзац ...
```

Команды `.PP` или `.LP` (выравненный слева абзац) завершают дело, начатое командой `.IP`. Аргументом `.IP` может быть любая строка: введите кавычки, а при необходимости и пробелы. Второй аргумент можно использовать, чтобы определить значение отступа.

Когда вы работаете с парой команд `.KS` и `.KE`, текст должен быть размещен в одном месте; текст, заключенный между этими командами, будет перенесен на новую страницу, если он не разместится весь на текущей странице. Заменив `.KS` на `.KF`, вы можете передвинуть текст за последующий текст в верхнюю часть следующей страницы (если его необходимо поместить на одной странице). Все таблицы в книге построены с помощью `.KF`.

Можно изменить большинство значений ms, принятых по умолчанию, путем установки некоторого числа регистров, являющихся переменными `troff` и используемых `ms`. Наиболее часто применяются регистры, управляющие размером текста и интервалом между строками. Нормальным размером текста считается размер в “10 точек”, где точка составляет 1/72 дюйма (единица, заимствованная из полиграфии). Обычно строки печатаются с 12–точечным разделением (интервалом). Чтобы изменить интервал, например на 9 или 11 точек (как сделано в наших отображениях), присвойте указанные числа регистрам `PS` и `VS`:

```
.nr PS 9
.nr VS 11
```

.AB	Печатать резюме; оканчивается .AE
.AU	Ввести в следующей строке имя автора; разрешены многократные .AU
.B	Начать печатать "жирный" текст либо выделить жирным шрифтом аргумент, если он есть
.DS t	Начать отображать (незаполненный) текст, оканчивающийся .DE t=L (выравнивание по левому краю), C (центрирование), B (центрирование блока)
.EQ s	Начать выравнивание s (входной поток eqn); оканчивается .EN
.FS	Начать печатать сноску; оканчивается .FE
.I	Начать печатать текст, выделенный курсивом, или выделить курсивом аргумент, если он есть
.IP s	Сделать абзац с отступом, помеченный s
.KF	Печатать слитно часть текста, если необходимо целиком передвинуть на следующую страницу; конец ее .KE
.KS	Печатать слитно часть текста на странице; заканчивается .KE
.LP	Печатать новый выравненный слева абзац
.NH n	Ввести числовой заголовок n–го уровня; затем сам заголовок до .PP или .LP
.PP	Сделать новый абзац
.R	Вернуться к обычному шрифту
.SH	Ввести подзаголовок; заголовок следует далее до .PP
.TL	Далее печатать название до следующей команды ms
.TS	Начать печатать таблицу (входной поток tbi); оканчивается .TE

Таблица 9.1: Распространенные команды форматирования ms (см. также справочное руководство по ms(7))

Другие числовые регистры включают LL для установки длины строки, PI — для определения отступов абзацев и PD — для отделения последних. Это влияет на следующие .PP или .LP.

Пакет макроопределений mm. Мы не будем подробно рассматривать этот пакет макроопределений, поскольку в целом, а зачастую и в деталях он похож на ms. Пакет mm обеспечивает контроль параметров в расширенном по сравнению с ms диапазоне, обладает большими возможностями (например, автоматически нумеруемые списки) и выдает лучшие сообщения об ошибках. В табл. 9.2 показаны команды mm, эквивалентные командам ms из табл. 9.1.

■ Упражнение: Пропуск завершающей команды типа .AE или .DE обычно ведет к неприятностям. Напишите программу mscheck для обнаружения ошибок во входном потоке ms (или в предпочитаемом вами пакете). Совет: воспользуйтесь awk.

9.2 Уровень troff

На практике приходится иногда выходить за пределы возможностей ms, mm или других пакетов, чтобы реализовать некоторые свойства "чистой" troff. Однако, как и к программированию на языке Ассемблера, прибегать к этому следует в крайних случаях.

.AS	Начать печатать резюме; оканчивается .AE
.AU	Задать имя автора
.B	Начать печатать “жирный” текст либо выделить жирным шрифтом аргумент, если он есть
.DF	Задать слитную часть текста, если необходимо ее целиком передвинуть на следующую страницу; оканчивается .DE
.DS	Начать отображать текст; оканчивается .DE
.EQ	Начать выравнивание (входной поток eqn); оканчивается .EN
.FS	Начать печатать сноску; оканчивается .FE
.I	Начать печатать текст, выделенный курсивом, или выделить курсивом аргумент, если он есть
.Hn "..."	Задать нумерованный заголовок n–го уровня "..."
.HU "..."	Задать ненумерованный заголовок "..."
.P	Сделать абзац. Используйте .nr Pt 1 один раз для создания абзаца с отступом
.R	Вернуться к обычному шрифту
.TL	Задать заголовок до следующей команды mm
.TS	Начать печатать таблицу (tbl входной поток); оканчивается .TE

Таблица 9.2: Распространенные команды форматирования mm

Вероятны три ситуации: доступ к специальным символам, использование встроенных команд замены шрифта и введение нескольких базовых функций форматирования.

Имена символов. Доступ к необходимым символам (греческим буквам, например, π, графике вида $\bullet$ и $\star$, разнообразным штрихам и пробелам) несложен, хотя и не вполне систематизирован. Каждый такой символ имеет имя \c, где c — одиночный символ, или \(cd, где cd — пара символов.

Программа troff печатает минус в коде ASCII как дефис, а не как '—'. Настоящий минус должен обозначаться через \-, а тире — через \(em, называемое “em пунктир”, символ “тире”.

В табл. 9.3 перечислены наиболее часто используемые специальные символы; в справочном руководстве по troff их число намного больше (в вашей системе перечень специальных символов может быть иным).

В ряде случаев требуется, чтобы troff не интерпретировала символ, особенно обратную дробную черту или точку в начале строки. Два наиболее часто применяемых “отменяющих” символа - \e и \&. Последовательность \e гарантированно печатается как обратная дробная черта, не интерпретируется и используется для получения такого символа в выходном потоке. С другой стороны, \& не несет никакой смысловой нагрузки: это пробел нулевой ширины. Главное назначение этой комбинации - заставить troff не интерпретировать точки в начале строк. Мы задействовали \e и \& здесь несколько раз. Например, фрагмент ms в начале главы был напечатан как

```
\&.TL
.I "Название документа"
\&.AU
.I "Имя автора"
\&.AB \&...
...
```

`-`	Дефис
`\(hy`	Дефис, аналогичный предыдущему
`\-`	Знак “минус”, набираемый текущим шрифтом
`\(mi`	Знак “минус”, набираемый математическим шрифтом
`\(em`	em тире
`\&`	Ничего; защищает точку в начале строки
`\blank`	Неразмножаемый пробел
`\\|`	Неразмножаемый полупробел
`\e`	Символ экранирования, обычно \
`\(bu`	Жирная точка
`\(dg`	Крестик +
`\(*a`	α, `\(*b`=β, `\(*c`=ξ, `\(*p`=π и т. д.
`\fX`	Символ смены шрифта на `X`; `X=P` — предыдущий (шрифт)
`\f(XX`	Символ смены шрифта на `XX`
`\sn`	Символ смены размера шрифта на `n`; `n=0` — предыдущий
`\s+-n`	Относительная замена размера шрифта

Таблица 9.3: Некоторые последовательности специальных символов `troff`

Конечно, этот фрагмент был напечатан следующим образом:

```
\e&.TL
\&.I "Название документа"
\e& .AU
...
```

и вы можете себе представить, как в свою очередь был напечатан последний фрагмент.

Другой специальный символ, “неразмножаемый” пробел, появляется изредка: это символ \, за которым следует пробел. Как правило, `troff` размножает обычный пробел, чтобы выравнять границы, но неразмножаемый пробел не позволяет “растягивать” строку. Он подобен любому другому символу и имеет фиксированную ширину. Его также можно использовать для передачи нескольких слов единым аргументом:

```
.I Название\ документа
```

Смена шрифта и размера символов текста. В большинстве случаев замена шрифтов и форматов может быть сделана с помощью начинающей строку макрокоманды типа `.I`, но иногда их замена должна осуществляться и в строке. В частности, символ конца строки разделяет слова, поэтому если требуется сменить шрифт в середине слова, макрокоманду нельзя использовать. С помощью `troff` можно решить эту проблему (отметим, что именно `troff`, а не пакет `ms` обеспечивает такую возможность).

Встроенные (in-line) команды `troff` вводит с использованием символа \. Наиболее часто применяются команды `\f` для смены шрифта и `\s` — для смены формата.

Шрифт, заменяемый командой `\f`, определяется символом, следующим непосредственно за `f`:

```
a\fBпро\fIизвольное \fR \fI мно\fBжество \fIшрифтов\fP
```

Это выводится как

про*извольное мно***жество** *шрифтов*

При смене шрифта `\fP` возвращает нас к предыдущему шрифту — тому, который был до последнего переключения. (Есть только один предыдущий шрифт, т. е. стека нет.)

Некоторые шрифты имеют двухсимвольные имена. Они специфицируются форматом `\f` (`XX`, где `XX` — имя шрифта. Например, шрифт, которым напечатаны программы в нашей книге, называется CW (курьер постоянной ширины), поэтому `keyword` пишется так:

```
\f\(CWkeyword \fP
```

Очевидно, печатать это довольно неудобно, поэтому мы ввели расширение `ms` — макрокоманду `.CW`, так что теперь нет необходимости печатать или читать символы `\`.

Воспользуемся указанным расширением, чтобы набирать слова внутри строки, такие, как `troff`, следующим образом:

```
The
.CW troff
formatter ...
```

Решения о форматировании, определяемые макрокомандами, также легко потом поменять.

Смена размера шрифта осуществляется последовательностью `\sn`, где `n` — одна или две цифры, определяющие новый размер: `\s8` переключает на восьмиточечный размер. В принципе можно выполнять относительные замены, предпосылая размеру плюс или минус. Например, слова можно напечатать в SMALL CAPS, введя

```
\s-2SMALL CAPS\s0
```

Комбинация `\s0` предписывает возвратить размер к его предыдущему значению. Это аналог `\fP`, но, следуя традиции `troff`, она не записывается как `\sP`. Наши расширения ms включают макрокоманду `.UC` (прописные буквы) для такого рода работы.

Основные команды `troff`. Даже располагая хорошим пакетом макрокоманд, мы должны знать достаточно много команд `troff` для управления пробелами и заполнением, для установки позиций табуляции и т. п. Команда `.br` вызывает “обрыв”, т. е. следующий вводимый за `.br` текст окажется на новой выходной строке. Это явление можно использовать, например, чтобы расщепить длинное название в подходящем месте:

```
.TL
Нос - Диалоговый язык
.BR
для арифметики с плавающей точкой
...
```

Команда `.nf` отключает нормальное заполнение выходных строк; каждая строка ввода переходит непосредственно в одну выходную строку. Команда `.fi` снова включает процедуру заполнения. Команда `.ce` центрирует следующую строку.

Команда `.bp` начинает новую страницу. Команда `.sp` вызывает появление на выходе одной пустой строки. За командой `.sp` может следовать аргумент, чтобы задать число пустых строк или число пробелов:

`.sp 3`	Оставить 3 пустых строки
`.sp .5`	Оставить пустые полстроки
`.sp 1.5i`	Отступить на 1,5 дюйма
`.sp 3p`	Вставить 3 позиции
`.sp 3.1c`	Оставить 3,1 сантиметра

Лишнее пространство в нижней части страницы не играет роли, поэтому `.sp` с большим аргументом эквивалентно `bp`.

Команда `.ta` устанавливает позиции табуляции (которые инициализируются каждые полдюйма). Команда

```
.ta n n n ...
```

расставляет позиции табуляции на определяемых расстояниях от левой границы. Как и для `.sp`, каждое число `n` измеряется в дюймах, если за ним следует '`i`'. Позиция табуляции с суффиксом `R` вызывает правое выравнивание текста на очередной позиции табуляции, `C` вызывает центрированную табуляцию.

Команда `.ps n` устанавливает значение `n` размера шрифта; команда `.ft X` устанавливает шрифт `X`. Правила для увеличения размеров и возвращения к предыдущему значению те же самые, что и для `\s` и `\f`.

Определение макрокоманд. Определение макрокоманд во всей полноте потребовало бы от нас более детального изучения `troff`, чем это необходимо. Поэтому мы ограничимся здесь рассмотрением нескольких примеров. В частности, определение для `.CW` имеет вид

```
.de CW                  Начать определение
\&\f (CW\\$1\fP\\$2     Смена шрифта для первого аргумента
..                      Конец определения
```

Комбинация `\$n` дает значение `n`–го аргумента при вызове макрокоманды: оно пусто в случае отсутствия `n`–го аргумента. Двойной символ `\\` откладывает вычисление `\$n` на время определения макрокоманды. Комбинация `\&` не позволяет интерпретировать аргумент как команду `troff` в том случае, если он начинается с точки, как показано ниже:

```
.CW .sp
```

9.3 Препроцессоры `tbl` и `eqn`

Программа `troff` — большая и сложная, и поэтому модифицировать ее для того, чтобы она выполняла новую задачу, нелегко. Соответственно разработка программ для набора математических выражений и таблиц требует другого подхода, а именно создания специальных языков, реализованных отдельными программами `eqn` и `tbl`, действующих как процессоры для `troff`. Программа `troff` по существу представляет язык Ассемблера для наборной машины, а `eqn` и `tbl` компилируют для нее код.

Вначале появилась `eqn`. Это было первое применение `yacc` не для целей программирования (Программа `eqn` вряд ли смогла бы появиться, если бы уже не существовала `yacc`.). Программа `tbl`, разработанная позднее, аналогична `eqn`, хотя и имеет независимый синтаксис; `tbl` не использует `yacc`, так как ее грамматика достаточно проста.

Средства программных каналов UNIX предполагают строгое разделение на отдельные программы. Кроме разбиения на части (что так или иначе необходимо, поскольку `troff` уже достигла максимального размера для **PDP–11**), программные каналы сужают круг взаимодействия между частями программы и между разрабатывающими их программистами. Последнее особенно важно: чтобы сделать препроцессор, не нужно "залезать" в исходную программу. Далее программные каналы позволяют не создавать большие промежуточные файлы при условии, что компоненты намеренно запускаются отдельно с целью отладки.

Однако организация взаимодействия программ через конвейеры связана с некоторыми проблемами. Отчасти снижается быстродействие, поскольку увеличивается объем ввода и вывода: обычно и `eqn`, и `tbl` дают расширение выходного потока по отношению к входному в отношении 8:1. Еще более существенно, что информация идет только в одном направлении. Например, нет способа определения текущего размера шрифта, что создаст неудобства в пользовании языком. И, наконец, трудно обеспечить сообщения об ошибках, так как иногда трудно связать диагностику из `troff` с `eqn` и `tbl`. Тем не менее преимущества разделения значительно перекрывают недостатки, поэтому было написано несколько препроцессоров, основанных на этой модели.

Таблицы. Обсудим кратко работу `tbl` и прежде всего таблицу операций по документации к `hoc`. `tbl` читает свои входные файлы или стандартный входной поток и преобразует текст между командами `.TS` (начало таблицы) и `.TE` (конец таблицы) в команды `troff`, печатающие таблицу, выравнивающие столбцы и обеспечивающие все типографские атрибуты. Строки `.TS` и `.TE` тоже копируются, поэтому пакет макроопределений выдает для них подходящие определения с тем, например, чтобы можно было помещать таблицу на одной странице и отделять ее от окружающего текста.

При формировании сложных таблиц вам, конечно, придется обращаться к справочному руководству по `tbl`. Хотя для уяснения основных принципов работы вполне достаточно приведенного ниже примера (из документации по `hoc`).

```
.TS
center, box;
c s
lfCW 1
\fBТаблица 1:\fP Операции по порядку уменьшения
            приоритета
.sp.5
^       возведение в степень (\s-1FORTRAN\s0 **) правоассоциативна
!\-     одноместные логическое и арифметическое отрицания
* /     умножение, деление
+\-     сложение, вычитание
> >=    операции отношения: больше, больше или равно
< <=    меньше, меньше или равно
\&== !=  равно, не равно (все отношения одинакового приоритета)
&&      логическое И (оба операнда всегда вычисляются)
||      логическое ИЛИ (оба операнда всегда вычисляются)
\&=     присваивание, правоассоциативна
.TE
```

В результате мы получаем следующую таблицу:

Таблица 1: Операции по порядку уменьшения приоритета

```
^       возведение в степень (FORTRAN **) правоассоциативна
! -     одноместные логическое и арифметическое отрицания
* /     умножение, деление
+ -     сложение, вычитание
> >=   операции отношения: больше, больше или равно
< <=   меньше, меньше или равно
== !=  равно, не равно (все отношения одинакового приоритета)
&&      логическое И (оба операнда всегда вычисляются)
!!      логическое ИЛИ (оба операнда всегда вычисляются)
=       присваивание, правоассоциативна
```

Слова перед точкой с запятой описывают глобальные свойства таблицы: центрировать по горизонтали на странице и заключить в рамку. Другие средства включают `doublebox` (сделать двойную рамку), `allbox` (включить каждый элемент в рамку) и `expand` (расширить таблицу на формат страницы).

Следующие строки до точки описывают формат различных секций таблицы. Первая спецификация служит для первой строки таблицы, вторая — для второй, последняя — для всех остальных строк. В табл. 1 вы видите только две строки спецификаций, поэтому вторая спецификация применяется к каждой строке таблицы после первой. Символы формата для элементов центрированных в столбце, — `c`, `r` и `l` для правого и левого выравнивания и `n` — для выравнивания чисел по десятичной точке. Символ `S` определяет столбец с промежутком; в нашем случае '`c S`' означает центровку названия над всей таблицей путем задания размера второго столбца так же, как и первого. Для столбца можно определить шрифт. Спецификация `tbl lfCW` позволяет печатать выравненный по левому краю столбец шрифтом `CW`.

Текст таблицы следует за информацией для форматирования. Символы табуляции разделяют столбцы и некоторые команды `troff`, например `.sp`, которые уместны внутри таблиц. (Отметим пару вхождений `\&`: незащищенный передний символ `-` и знак `=` в столбцах указывают `tbl` на необходимость располагать строки таблицы в этой точке.)

Программа `tbl` строит более широкий набор таблиц, чем показано в примере: текст может помещаться в рамки, могут вертикально выравниваться заголовки столбцов и т. д. Самый легкий способ использовать `tbl` для создания сложных таблиц — обратиться к справочному руководству по UNIX (том 2A) и применить перечисленные в нем команды.

Математические выражения. Второй препроцессор `eqn` превращает язык, описывающий математические выражения, в команды `troff`, чтобы эти выражения печатать. Препроцессор автоматически обрабатывает смены шрифта и формата и, кроме того, предоставляет имена для стандартных математических символов. Входной текст для `eqn` обычно находится между строками `.EQ` и `.EN`, аналогично командам `tbl .TS` и `.TE`. Например,

```
.EQ
x sub i
.EN
```

выдает x_i. Если используется пакет `ms`, уравнение печатается как "отображение", а возможный аргумент `.EQ` определяет номер уравнения. Например, формула интеграла Коши

$$f(\xi) = \frac{1}{2\pi i} \int_c \frac{f(z)}{z - \xi} dz$$

записывается как

```
.EQ (9.1)
f( zeta ) ~=~ 1 over {2 pi i} int from C
    f(z) over {z - zeta} dz
.EN
```

В основу языка `eqn` и положен способ чтения вслух математических формул. Единственное различие между "разговорной" математикой и входным текстом `eqn` состоит в том, что скобки `{ }` отменяют заданные по умолчанию правила предшествования языка, однако обычные скобки специального смысла не имеют. Пробелы тем не менее важны. Заметим, что первое вхождение zeta в примере, приведенном выше, окружено пробелами: ключевые слова, такие, как `zeta` и `over`, распознаются только тогда, когда они окружены пробелами или скобками, но ни те, ни другие в выходной текст не попадают. Чтобы обеспечить пробелы в выходном потоке, используйте символ `~`, как показано в примере (`≅~`). Для получения скобок используйте "`{`" и "`}`".

Существует несколько классов ключевых слов `eqn`. Греческие буквы записываются прописными и строчными: `lambda` и `LAMBDA` (λ и Λ). Другие математические символы имеют имена, такие как `sum`, `int`, `infty`, `grad`: $\sum$, $\int$, ∞, $\bigtriangledown$. Есть знаки позиции, например sub, sup, from, to, and, over:

Эта формула выводится так:

$$\sum_{i=1}^{\infty} x_i^2 \to \frac{1}{2n}$$

```
sum from i=0 to infinity x sub i sup 2~->~1 over {2pi}
```

Существуют знаки операций типа `sqrt`, расширяющие скобки, фигурные скобки и т. д. Программа `eqn`, кроме того, позволяет создавать из объектов столбцы и матрицы. Предусмотрены команды для управления шрифтами и позициями, если те, которые установлены по умолчанию, не подходят.

Часто приходится помещать небольшие математические выражения, такие, как `log-10(x)`, в тело текста, а не в отображение. Ключевое слово `eqn delim` определяет пару символов для выделения подобных выстроенных выражений. Символы, задаваемые в качестве левого и правого ограничителей, обычно одинаковы: часто применяется знак доллара `$`. Но поскольку `hoc` использует `$` для аргументов, в нашем примере мы употребили `@`. Символ `%` тоже удобен как ограничитель, но других символов избегайте: многие из них имеют специальные назначения в различных программах, поэтому вы можете спровоцировать непредсказуемое поведение `eqn` (именно так у нас и получилось с этим разделом).

Итак, после обозначения

```
.EQ
delim @ @
.EN
```

можно напечатать встроенное выражение $\sum_{i=0}^{\infty} x_i$ в виде

```
@ sum from i == 0 to infinity x sub i @ can be printed.
```

Встроенные выражения используются для вывода формул в таблице (см. пример из документации по hoc):

```
.TS
center,box
css
lfCWn1.
\fBТаблица3:\fPВстроенныеконстанты
.sp.5
DEG57.295779513082320876800@180/pi@,градусыирадианы
E2.7182818284590452353600@e@,основаниенатуральногологарифма
GAMMA0.5772156649015328606000@gamma@,константаЭйлера-Масчерони
PHI1.6180339887498948482000@(sqrt5+1)/2@,золотоесечение
PI3.1415926535897932384600@pi@,круговоетрансцендентноечисло
.TE
```

Из этой таблицы, кроме того, видно, как строки tbl помещают десятичные точки в числовых (n) столбцах. Результат показан ниже.

```
Таблица 3 : Встроенные константы

DEG   57.29577951308232087680 $180/\pi$, градусы на радианы
E      2.71828182845904523536  $e$, основание натуральных логарифмов
GAMMA 0.57721566490153286060  $\gamma$, константа Эйлера-Масчерони
PHI   1.61803398874989484820  $\frac{(\sqrt 5 + 1)}{2}$, золотое сечение
PI    3.14159265358979323846  $\pi$, круговое трансцендентное число
```

И, наконец, поскольку eqn выделяет курсивом любую строку букв, которые она не распознает, довольно просто выделять обычные слова курсивом. Последовательность @Word@ например, печатается как *Word*. Но будьте внимательны: eqn распознает некоторые обычные символы (такие, как from и to) и специальным образом их рассматривает: она "глотает" пробелы, поэтому указанный прием следует применять с осторожностью.

Получение выходного потока. Как только ваш документ готов, вы должны соединить все препроцессоры и troff в цепочку, чтобы получить выходной поток. Порядок команд следующий: tbl, eqn, troff. Если вы просто используете troff, то печатайте

```
$ troff -ms имена_файлов                    (или -mm)
```

Иначе вам придется задать аргумент имена_файлов первой команде в цепочке и дать остальным командам читать их стандартный входной поток, как показано ниже:

```
$ eqn имена_файлов | troff -ms
```

или

```
$ tbl имена_файлов | eqn | troff -ms
```

Неудобно следить за тем из препроцессоров, который действительно должен печатать какой-то отдельный документ. Мы сочли уместным написать программу doctype, обеспечивающую вывод соответствующей последовательности 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 с помощью инструментов, рассмотренных в гл. 4. В частности, программа `awk` отыскивает последовательность команд, используемую препроцессорами, и печатает строку команд, которые нужно вызвать, чтобы отформатировать документ. Она также находит команду `.PP` (абзац) для форматирования пакетом запросов `ms`.

```
$ cat doctype
# doctype:  synthesize proper command line for 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"
} '
$
```

(Флаг `-h` заставляет ее подавлять заголовки имен файлов на каждой строке: к сожалению, этот аргумент есть не во всех версиях системы.) При сканировании входного потока собирается информация о том, какие компоненты используются. После просмотра входной поток обрабатывается в требуемой последовательности для печати выходного текста. В формировании документов `troff` со стандартными препроцессорами есть специфика, и главная задача состоит в том, чтобы заставить "думать" об этом саму машину.

Программа `doctype` в нашем примере подобна `bundle`–программе, которая создает программу. Однако в таком виде она требует от пользователя вновь вводить строку для `shell`. В одном из приводимых ниже упражнений вам предлагается это исправить.

Когда дело дойдет до запуска реальных команд `troff`, не забывайте, что поведение программы зависит от системы: на некоторых установках она управляет наборным устройством непосредственно, в то время как на других выдает в стандартный выходной

поток информацию, которая должна быть послана на наборное устройство отдельной программой.

Между прочим, в новой версии этой программы не предусмотрена программа `egrep` или `sort`; `awk` сама просматривает весь входной поток. Для больших документов такой вариант оказывается слишком медленным, поэтому для ускорения поиска мы добавили `egrep` и затем `sort -u`, чтобы избавиться от дублирования. При построении типичных документов накладные расходы по созданию двух дополнительных разбирающих данные процессов меньше, чем запуск `awk` в тех же целях с большим объемом входного текста.

В качестве иллюстрации сравним `doctype` с версией, только запускающей `awk` применительно к содержимому данной главы (около 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
$
```

Сравнение, очевидно, в пользу версии с тремя процессами. (Работа была выполнена в однопользовательском режиме; соотношение значений времени показало бы даже более значительное преимущество версии `egrep` и при повышенной нагрузке на систему.) Отметим, что, прежде чем начать оптимизацию, мы получили сначала простую работающую версию.

■ Упражнение: Как мы сформатировали эту главу?

■ Упражнение: Если вашим ограничителем для `eqn` является знак доллара, то как вы получите этот знак в выходном потоке? Подсказка: исследуйте кавычки и предопределенные слова `eqn`.

■ Упражнение: Почему команда

```
$ doctype имена_файлов
```

не выполняется? Модифицируйте `doctype` так, чтобы запускать команду, полученную в результате, вместо того, чтобы ее печатать.

■ Упражнение: Важны ли накладные расходы на добавочную команду `cat` в `doctype`? Перепишите `doctype`, чтобы избавиться от дополнительного процесса. Какая версия проще?

■ Упражнение: Что лучше: использовать `doctype` или писать файл `shell`, содержащий команды, для форматирования конкретного документа?

■ Упражнение: Поэкспериментируйте с различными комбинациями `grep`, `egrep`, `fgrep`, `sed`, `awk` и `sort`, чтобы повысить быстродействие `doctype`.

9.4 Справочник

Основной документацией для команды является обычно справочная страничка (называемая далее справочником) — одностраничное описание в справочном руководстве по UNIX (см. рис. 9.2). Справочник хранится в стандартном каталоге, как правило, в `/usr/man`, в подкаталоге, нумерованном в соответствии с разделом руководства. Например, наш справочник для `hoc` хранится в `/usr/man/manl/hoc.1`.

Справочники печатаются с помощью команды `man(1)`, т. е. файла `shell`, который запускает `nroff -man`, поэтому `man hoc` печатает справочник `hoc`. Если одно и то же имя появляется в нескольких разделах, как само man (раздел 1 описывает команду, тогда как раздел 7 описывает макрокоманды), то раздел можно определить для man как

```
$ man 7 man
```

В результате печатается только описание макрокоманд пакета `man`. По умолчанию принято печатать все справочники с определенным именем, использующим `nroff`, но `man -t` порождает справочники для наборного устройства с помощью `troff`.

Автор справочника создает файл в соответствующем подкаталоге `/usr/man` Чтобы печатать справочник, команда `man` вызывает `nroff` или `troff` с пакетом макроопределений; это можно увидеть, отыскав команду man для вызовов форматирующей программы. Мы получили бы такой результат:

```
$ 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 ;;
$
```

Разнообразие достигается применением флагов: `nroff` или `troff`, запускается или нет `eqn` и т. д. Справочник по макрокомандам, вызываемый `troff -man`, определяет команды `troff`, форматирующие в стиле данного руководства. В принципе они аналогичны макрокомандам `ms`, но есть и различия, особенно в установке названия и командах смены шрифта. Макрокоманды кратко документированы в `man(7)`, но основные из них легко запоминаются. Разметка справочника такова:

```
.TH COMMAND номер раздела
.SH NAME
```

```
команда \- краткое описание функций
.B команда
возможные аргументы
.SH DESCRIPTION
Подробное объяснение команд и флагов.
Абзацы вводятся .PP.
.PP
Это новый абзац.
.SH FILES
Файлы, используемые командой, например, passwd(1)
упоминает /etc/passwd
.SH "SEE ALSO."
Ссылки к связанным документам, включая другие
справочники
.SH DIAGNOSTICS
Описание некоторого необычного выходного потока
(например, см. cmp(1))
.SH BUGS
Неожиданные черты (не всегда ошибки; см. ниже)
```

Если какой-то раздел пуст, его заголовок опускается. Строка. `.TH` и разделы NAME, SYNOPSIS, DESCRIPTION не обязательны.

Строка

```
.TH COMMAND номер раздела
```

называет команду и определяет номер раздела. Различные строки `.SH` идентифицируют разделы справочника. Разделы NAME и SYNOPSIS являются специальными; остальные содержат обычный текст. Раздел NAME называет команду (на этот раз строчными буквами) и дает ее описание в одной строке, а раздел SYNOPSIS называет флаги, но не описывает их. Как и в любом разделе, входной текст имеет произвольную форму, поэтому смену шрифта можно определять с помощью макрокоманд `.B`, `.I`, `.R`. В разделе SYNOPSIS и имя, и флаги выделены жирным шрифтом; прочая информация печатается обычным шрифтом. Разделы `ed(1)` NAME и SYNOPSIS, например, имеют вид и выводятся как

```
.SH NAME
ed \- text editor
.SH SYNOPSIS
.B ed
[
.B \-
][
.B \-x
][ name ]
```

```
NAME
    ed - text editor
SYNOPSIS
    ed [ - ][ -x ][ name ]
```

Заметьте, что используется \-, а не просто -.

Раздел DESCRIPTION описывает команду и ее флаги. В большинстве случаев это описание команды, а не языка, определяемого командой. Справочник `cc(1)` не определяет язык Си: он указывает, как запустить команду ее, чтобы компилировать программы на Си, как вызвать оптимизатор, где оставлен результат и т. п. Язык описывается в руководстве для пользователя по Си, на которое есть ссылки в разделе `cc(1)` SEE ALSO. С другой стороны, разделение не абсолютно: `man(7)` есть описание языка макрокоманд руководства.

По соглашению имена команд и метки флагов (такие, как "name" в справочнике `ed`) печатаются курсивом с помощью макрокоманды `.I` (первый аргумент печатается курсивом, второй — обычным шрифтом). Макрокоманда `.IR` используется здесь потому, что макрокоманда `.I` в пакете не обеспечивает недокументированного, но удобного применения второго флага в `ms`.

Раздел FILES упоминает любые файлы, неявно используемые командой. DIAGNOSTICS следует включать только в том случае, если команда вырабатывает необычный выходной поток. Это могут быть диагностические сообщения, сведения о состоянии выхода или информация о неожиданных отклонениях от стандартного выполнения команды. Раздел BUGS тоже назван отчасти неверно. Дефекты, о которых здесь сообщается, представляют собой не столько ошибки, сколько недостатки - просто ошибки должны быть исправлены прежде, чем команда будет введена в систему. Чтобы понять, для чего нужны разделы DIAGNOSTICS и BUGS, вам следует пролистать стандартное руководство.

Поясним на примере, как писать справочник. Источник для `hoc(1)`, `/usr/man/man1-/hoc.1`, показан на рис. 9.1, а на рис. 9.2 представлен выходной текст после вызова.

Упражнение 9.8. Напишите справочник для doctype. Напишите версию команды man, которая отыскивает документацию по вашим личным программам в вашем собственном каталоге man.

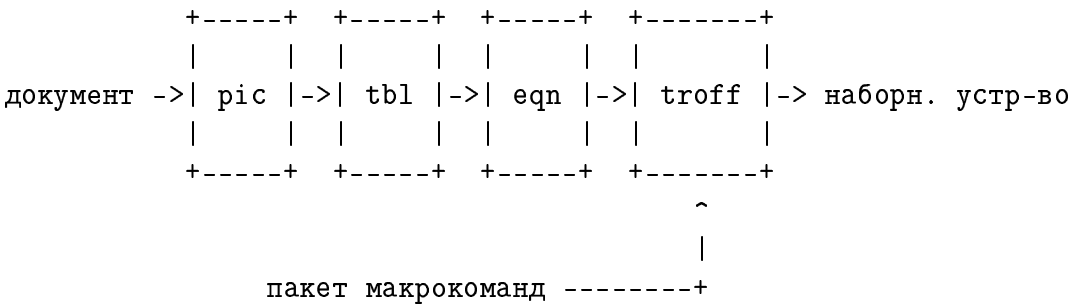
9.5 Дополнительные средства для подготовки документации

Для подготовки документации существует несколько дополнительных программ. Команда `refer(1)` отыскивает ссылки на ключевые слова, вставляет эти ссылки в строки вашего документа и помещает раздел ссылок в его конце. Определив соответствующую макрокоманду, вы можете добиться, чтобы `refer` печатала ссылки в том виде, в каком они вам нужны. Имеются определения для многих журналов по вычислительным наукам. Команда `refer` является частью седьмой версии, но не включена в некоторые другие версии.

Программы `pic(1)` и `ideal(1)` предназначены для подготовки рисунков, так же как `eqn` для уравнений. Подготовить рисунки значительно сложнее, чем уравнения (по крайней мере для набора), а поскольку традиции здесь отсутствуют, оба языка отчасти облегчают знакомство с этой техникой и ее использование. В качестве иллюстрации приведем простой рисунок и его выражение на `pic`.

```
.PS
.ps -1
box invis "document"; arrow
```

```
box dashed "pie"; 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 книге сделаны с помощью `pic`. Программы `pic` и `ideal` не являются частью седьмой версии, но сейчас они в нее включены.

Программы `refer`, `pic` и `ideal` представляют собой препроцессоры `troff`. Кроме того, в вашей документации есть программы для просмотра и комментирования текста. Наилучшая из известных программ - `spell(1)`, которая выдает сообщения о возможных ошибках написания в файлах; мы ее здесь применяли. Программы `style(1)` и `diction(1)` анализируют пунктуацию, грамматику и использование языка. Со временем все они были превращены в "Рабочее место писателя" — набор программ, помогающих улучшить стиль изложения. Эти программы полезны для обнаружения клише и слов, не являющихся необходимыми, а также некорректных фраз.

Программа `spell` считается стандартной. В вашей системе могут быть и другие программы, вы легко обнаружите их с помощью man:

```
$ man style diction wwb
```

или путем просмотра `/bin` и `/usr/bin`.

Историческая и библиографическая справка. Программа `troff` (ее автор — Дж. Осанна), предназначенная для графических систем **CAT–4**, имеет свою историю, восходя к **RUNOFF**, созданной Д. Е. Зальтцером для **CTSS** в MIT в начале 60-х годов. Обе программы имеют общие цели и основной синтаксис команд, хотя `troff`, конечно, более сложная и мощная программа, а наличие `eqn` и других препроцессоров значительно повышает ее эффективность. Существует несколько новых программ для наборных устройств с более "цивилизованным" форматом входного текста; наиболее известны из них TEX Д. Кнута ("TEX and Metafont: New Direction in Typesetting", Digital Press, 1979) и Scribe Б. Рейда ("Scribe: a high–level approach to computer document formatting". 7th Symposium on the Principles of Programming Languages, 1980).

Статья Фурута, Дж. Скофилда и А. Шоу "Document Formatting Systems: Survey, Concepts and Issues" (Computing Surveys, 1982) дает хороший обзор таких систем.

Представляет интерес оригинальная работа по `eqn` Б. Кернигана и Л. Черри "A system for typesetting mathematics" (CACM, March 1975). Пакеты макрокоманд `ms`, `tbl` и `refer`

принадлежат М. Леску; они документированы только в справочном руководстве по UNIX (том 2A).

Препроцессор `pic` описан в статье Б. Кернигана "PIC — a language for typesetting graphics" (Software–Practice and Experience, January, 1982), препроцессор `ideal` — в статье К. Ван Вика "A high–level language for describing pictures" (ACM Translation on Graphics, April, 1982).

Команда `spell` из файла `shell`, написанного С. Джонсоном, превратилась в Си — программу Д. МакИлроя. Программа `spell` из седьмого издания для быстрого поиска использует механизм хеширования и правила для автоматического отделения суффиксов и префиксов, чтобы уменьшить занимаемое словарем место. (См.: Mcllroy. M. D. "Development of a spelling list". IEEE Transaction on Communications, Jaruary, 1982).

Программы `style` и `diction` описаны в работе Л. Черри "Computer aids for writers" (SYGPLAN Symposium on Text Manipulation, Portland, Oregon, June, 1981).

```
$ man -t hoc

.TH HOC 1
.SH NAME
hoc \- диалоговый язык для арифметики с плавающей точкой
.SH SYNOPSYS
.B hoc
[ файл ... ]
.SH DESCRIPTION
.I Hoc
интерпретирует простой язык для арифметики с плавающей точкой,
примерно уровня Бейсика, с синтаксисом, подобным Си,
и с процедурами и функциями с аргументами, а также с рекурсией.
.PP
Поименованные
.IR файлы
читаются и интерпретируются по порядку. Если
.I файл
не указан или если
.I файл это '\-'
I1 hoc
интерпретирует стандартный входной поток.
.PP
Входной поток
.I Hoc
состоит из
.I выражений и
.IR операторов.
Выражения вычисляются и их результаты печатаются. Операторы, обычно
присваивания и определения функций или процедур, не вырабатывают выходного
результата, если они явно не вызывают
.IR print.
.SH "SEE ALSO"
.I
Hoc \- Диалоговый язык для арифметики с плавающей точкой Брайена Кернигана и
Роба Пайка.
.br
.IR bas(1),
.IR bc(1)
and
.IR dc(1).
.SH BUGS
Восстанавление после ошибок в определениях функции и процедур несовершенно.
.br
Обработка концов строк не совсем удобна для пользователя.
```

Рис. 9.1: /usr/man/manl/hoc.1

```
HOC(1)                                                    HOC(1)

NAME

hoc - диалоговый язык для арифметики с плавающей точкой

SYNOPSYS

hoc [ файл ... ]

DESCRIPTION

Нос интерпретирует простой язык для арифметики с плавающей точкой, примерно
уровня Бейсика, с синтаксисом, подобным Си, и с процедурами и функциями с
аргументами, а также с рекурсией. Поименованные файлы читаются и
интерпретируются по порядку. Если файл не указан или если файл это '-' hoc
интерпретирует стандартный входной поток. Входной поток Нос состоит из
выражений и операторов. Выражения вычисляются и их результаты печатаются.
Операторы, обычно присваивания и определения функций, или процедур, не
вырабатывают выходного результата, если они явно не вызывают print.

"SEE ALSO"

Нос - Диалоговый язык для арифметики с плавающей точкой Брайена Кернигана и
Роба Пайка.
bas(1), bc(1) and dc(1).

BUGS

Восстанавление после ошибок в определениях функции и процедур несовершенно.
Обработка концов строк не совсем удобна для пользователя.

8-я версия                                                        1
```

Рис. 9.2: hoc(1)

Глава 10

Эпилог

Операционной системе UNIX уже более десяти лет, а число использующих ее вычислительных машин растет сейчас быстрее, чем когда-либо. Для системы, сконструированной без коммерческих целей или даже намерений, это уникальный успех.

Главная причина популярности системы, вероятно, в ее переносимости — свойстве, благодаря которому все, кроме небольших частей компиляторов и ядра, выполняется на любом компьютере без изменений. Поэтому разработчикам, запускающим программное обеспечение UNIX на своих машинах, достаточно лишь небольших усилий на ее адаптацию, чтобы успешно пользоваться все расширяющимся рынком программ для UNIX.

Система UNIX стала популярной задолго до того, как она приобрела коммерческое значение, и даже до того, как начала применяться не только на **PDP–11**, но и на других машинах. Статья Ритчи и Томпсона в CACM в 1974 г. вызвала интерес в академических кругах, а в 1975 г. шестая версия получила распространение и в университетах. В середине 70-х годов о UNIX заговорили: хотя система не поддерживалась и не имела гарантий, нашлись энтузиасты, которые работали с ней сами и рекомендовали ее другим. Еще одна причина успеха системы заключалась в том, что поколение программистов, использовавших академические системы UNIX, ныне ожидают найти среду UNIX в сфере своей деятельности.

Чем объясняется такой интерес к системе? Дело в том, что она была сконструирована и построена двумя очень талантливыми людьми, единственная цель которых состояла в создании среды, удобной для разработки программ. Свободные от давления рынка ранние системы были достаточно малы, чтобы в них мог разобраться один человек. Дж. Лайонс излагал шестую версию ядра в рамках курса по операционным системам для выпускников университета Нового Южного Уэльса в Австралии. В своих заметках он писал, что вся документация умещается в "студенческом портфеле" (это зафиксировано в недавних версиях).

Та ранняя версия базировалась на изобретениях в сфере приложений вычислительной науки, включающих обработку потоков (программные каналы), регулярные выражения, теорию языков (`yacc`, `lex` и т. п.) и ряд специальных вопросов типа алгоритма `diff`. Объединенные в одной системе, они стали ядром с такими возможностями, которыми не всегда располагают даже большие операционные системы. В качестве примера можно привести систему ввода-вывода, состоящую из иерархической файловой системы, редкой по тем временам, устройств, которым поставлены в соответствие имена в файловой системе, так что они не требовали применения специальных утилит, и дюжины основных системных вызовов, подобных примитивам `open` с двумя аргументами. Программное обеспечение, написанное на языке высокого уровня, поставлялось вместе с системой, так

что его можно было изучать и модифицировать. На компьютерном рынке UNIX известна как одна из стандартных операционных систем. Размер ее ядра за последнее десятилетие вырос в 10 раз, хотя, к сожалению, качественно оно улучшается существенно медленнее. Увеличилось число труднопонимаемых программ, которые не созданы в существующей среде. Создаваемые средства обрастают командами с флагами, которые затемняют первоначальный замысел программ. Так как исходные тексты программ зачастую не распространяются вместе с системой, образцы хорошего стиля программирования становятся менее доступными.

Тем не менее последние версии все еще насыщены идеями,, сделавшими ранние версии столь популярными. Принципы, на которых основана UNIX — простота структуры, отсутствие непропорциональных средств, использование существующих программ (вместо того, чтобы создавать их заново), программируемость командного языка, древовидная структура файловой системы и т. д. — продолжают доминировать и вытесняют идеи "монолитных" предшествующих систем. Система UNIX, конечно, не может стать навсегда непревзойденной; однако те системы, которые рассчитывают ее "обойти", должны будут заимствовать многие из ее фундаментальных идей.

В предисловии отмечалось, что существует подход, или философия, UNIX, т. е. особый стиль программирования. Теперь, прочитав всю книгу, вы, должно быть, сможете оценить элементы этого стиля, проиллюстрированного многочисленными примерами.

Во–первых, пусть машина работает. Используйте программы типа `grep`, `wc` и `awk`, чтобы автоматизировать задачи, которые бы вам пришлось делать вручную в других системах.

Во–вторых, пусть работает и человек. Используйте программы, которые уже существуют как "строительные блоки" в ваших программах, с `shell` и программируемыми фильтрами, чтобы соединить их воедино. Пишите небольшие программы, обеспечивающие взаимодействие с имеющимися, которые решают реальные задачи, как мы сделали это в `diff`. Среда UNIX богата средствами, пригодными для самых разнообразных комбинаций, — выберите из них единственно верную.

В–третьих, выполняйте работу поэтапно. Постройте какую–нибудь простую полезную программу и, исходя из своего опыта, определите, что (если вообще что–либо нужно) следует делать далее. Не добавляйте к ней средства, пока вы на практике не выясните, что именно необходимо.

В-четвертых, создавайте инструментарий. Пишите программы, которые "стыкуются" с существующей средой, усиливая их, а не просто расширяя их возможности. Хорошо построенные, такие программы сами становятся частью чьего–либо инструментария.

Разумеется, система не вполне совершенна. Читая книгу, вы сталкивались со многими условностями, бессмысленными различиями в программах и произвольными ограничениями. Однако, несмотря на все свои недостатки, UNIX, в самом деле, хороша в том, для чего она предназначена: в обеспечении удобной среды программирования. И хотя система уже начала обнаруживать признаки "среднего возраста", она все еще жизнеспособна и все еще популярна. Это заслуга нескольких одаренных человек, которые в 1969 г. набросали на доске проект удобной среды программирования, получившей впоследствии высокую оценку целого поколения программистов.

Глава 11

Дополнение: Краткое описание редактора

Стандартный текстовый редактор UNIX создан К. Томпсоном в начале 70-х годов для вычислительной среды на малых машинах (первая система UNIX ограничивала предельный размер программ пользователя до 8К байт) с терминалами "твердой копии", работавшими при очень низких скоростях (10–15 символов в секунду). Этот редактор написан на базе более ранней версии `qed`, которая была в то время популярна.

С развитием технологии `ed` постигла та же судьба. Почти наверняка вы найдете в своей системе другие редакторы с интересными свойствами, в частности с возможностью "визуального", или "экранного", редактирования, при котором на экране терминала отражаются все вносимые вами коррективы.

Почему же тогда мы тратим время на, казалось бы, устаревшую программу? Дело в том, что `ed` выполняет некоторые операции весьма успешно, несмотря на свой возраст. Эта программа есть на всех установках UNIX, и вы всегда найдете ее при переходе с одной системы на другую. Она работает хорошо и с низкоскоростными телефонными линиями, и с любыми терминалами. Кроме того, `ed` легко запускать из командного файла, в то время как большинство экранных редакторов управляются с терминала и не могут должным образом получить входной поток из файла.

Редактор `ed` предоставляет регулярные выражения для поиска по образцу. Регулярные выражения, на которых основан `ed`, присутствуют во всех частях системы: `grep` и `sed` применяют почти такие же, а `egrep`, `awk`, `lex` расширяют их. `Shell` использует для сравнения имен файлов иной синтаксис, но те же самые идеи. Некоторые экранные редакторы имеют "строчный режим", который предусматривает обращение к регулярным выражениям `ed`.

И, наконец, `ed` обладает высоким быстродействием. Вполне возможно вызвать `ed`, заменить в файле одну строку, записать новую версию и вернуться из него, причем все это происходит быстрее, чем один только запуск большого и более сложного экранного редактора.

Основные сведения. Программа `ed` редактирует один файл за один раз. Она работает с копией файла. Чтобы внести исправления в первоначальный файл, вы должны дать явную команду. Редактор предоставляет команды для манипуляций с последовательными строками или строками, соответствующими образцу, а также команды для внесения в строки изменений.

Каждая команда `ed` представляет собой символ (обычно букву). Большинству команд может предшествовать один или два номера строки, которые указывают, на какую стр-

оку или строки должна воздействовать команда: в противном случае подразумевается номер, принятый по умолчанию. Номер строки можно специфицировать абсолютной позицией в файле (1,2 ...), символами $ для последней строки и "." для текущей, процедурой поиска по образцу, использующей регулярные выражения, и их аддитивными комбинациями.

Рассмотрим, как с помощью ed можно создавать файлы, используя стихи Де Моргана из первой главы.

```
$ ed poem
? poem                                  Предупреждение: файл poem не существует
a                                       Начать добавление строк
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
.                                       Печатаем '.' чтобы закончить ввод
w poem                                  Пишем строки в файл poem
121                                     ed сообщает, что записан 121 символ
q                                       Выход
```

Команда a добавляет или присоединяет строки. Режим добавления заканчивается строкой, состоящей из одной точки. Из-за отсутствия индикации режима, в котором вы работаете, возможны две распространенные ошибки: ввод текста без команды a и ввод команды до ввода '.'.

Редактор ed никогда не будет писать ваш текст в файл автоматически; вы должны задать это с помощью команды w. Однако, если вы пытаетесь закончить редактирование без записи ваших изменений, ed выдает '?' как предупреждение. Есть другая команда q, позволяющая завершить работу независимо от внесения исправлений.

```
$ ed poem
121                                     Файл существует и имеет 121 символ
a Добавить еще строки в его конец
And the great fleas themselfs, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.                                       Печатаем '.' для завершения
q                                       Пытаемся выйти
?                                       Предупреждение: не было записи
w                                       Нет имени файла; подразумевается poem
263
q                                       Теперь можно выходить
$ wc poem                               Проверьте для уверенности
8 46 263 poem
$
```

Временная передача управления shell с помощью '!'. Если вы запустили ed, то можете временно выйти из него, чтобы запустить другую команду shell. В этом случае нет необходимости прекращать работу — достаточно ввести команду ed '!'

```
$ ed poem
263
! wc poem                          Запуск wc без выхода из ed
8 46 263 poem
!                                  Вернулись из команды
q                                  Выход без w годится: не было исправлений
$
```

Печать. Строки файла нумеруются как `1,2 ...` Вы можете печатать `n`–ю строку, дав команду `np` или просто номер `n`, и строки с `m` по `n`, используя `m,np`. "Номером строки" `$` обозначается последняя строка, так что строки можно не считать.

`1`	Печатать первую строку; `1p` — то же самое
`$`	Печатать последнюю строку; `$p` — то же самое
`1,$p`	Печатать строки с первой по последнюю

Печатать файл по одной строке проще всего; нажимая клавишу *RETURN*, вы можете вернуться на одну строку назад с помощью '`-`'. Можно комбинировать номера строк с '`+`' и '`-`':

`$-2,$p`	Печатать последние три строки
`1,2+3p`	Печатать строки с первой по пятую

Однако нельзя печатать после конца файла или в обратном порядке; команды типа `$,$+1p` и `$,1p` считаются незаконными.

Команда `list 1` выводит текст в формате с видимыми символами. Это удобно при поиске в файлах управляющих символов, при различении пробелов, табуляции и т. п. (см. `vis` в гл. 6).

Образцы. Как только размер начинает превышать две строки, становится неудобным печатать его весь целиком, чтобы отыскать нужную строку. Редактор `ed` предлагает способ поиска строк, совпадающих с некоторым образцом, шаблоном: `/pattern/` обнаруживает очередное вхождение `pattern`.

```
$ ed poem
263
/flea/                         Ищет очередную строку, содержащую flea
Great fleas have little fleas
/flea/                         Ищет еще одну
And little fleas have lesser fleas,
//                             Ищет следующую по тому же образцу
And the great fleas themselves, in turn,
??                             Поиск в обратном направлении по тому же образцу
And little fleas have lesser fleas,
```

Редактор запоминает образец, применявшийся вами в последний раз, так что можно повторить поиск просто с помощью `//`. Для поиска в обратном направлении воспользуйтесь `?pattern?` и `??`.

Поиск с помощью `/.../` и `?...?` — циклический, т. е. продолжается в обратном направлении после достижения одного из концов текста:

```
$p                          Печатать последнюю строку ('p' необязательна)
and greater still, and so on.
/flea/                      Следующее flea вблизи начала
Great fleas have little fleas
??                          От начала идет в обратном направлении
have greater fleas to go on;
```

Результатом поиска по образцу типа `/flea/` является номер строки, например 1 или `$`, который может использоваться в том же контексте, что и такие номера:

`1,/flea/p`	Печатать от единицы до следующего `flea`
`?flea?+1,$p`	Печатать от предыдущего `flea + 1` до конца

Текущая редактируемая строка. Редактор `ed` отслеживает последнюю строку, с которой имели дело: печатали или вводили текст, читали из файла. Это текущая строка с именем '.'. Каждая команда определенным образом влияет на текущую строку, обычно настраивая ее на ту, с которой она последний раз работала. Вы можете использовать текущую строку так же, как `$` или номер строки типа 1:

```
$ ed poem
263
.               Печатает текущую строку; после чтения файла это то же, что $
and greater still, and so on.
.-1,.p          Печатает предыдущую строку и еще одну
While these again have greater still,
and greater still, and so on.
```

Выражения для номера строки могут быть сокращены:

Сокращение	**Эквивалент**	**Сокращение**	**Эквивалент**
`-1`	`.-1`	`+`	`.+1`
`--` или `-2`	`.-2`	`++` или `+2`	`.+2`
`-n`	`.-n`	`+n`	`.+n`
`$-`	`$-1`	`.3`	`.+3`

Добавление, замена, исключение, вставка. Команда `a` (добавить) добавляет строки после определенной строки, команда `d` (удалить) вычеркивает строки, команда `i` (вставить) вставляет строки перед определенной строкой, команда `c` (заменить) заменяет строки, действуя как комбинация команд "удалить" и "вставить".

`na`	Добавить текст после строки `n`
`ni`	Вставить текст перед строкой `n`
`m,nd`	Удалить строки с `m` по `n`
`m,nc`	Заменить строки с `m` по `n`

Если номера строк не указаны, используется текущая строка. Новый текст для команд `a`, `c` и `i` оканчивается строкой '.'; точка, введенная в последней строке, оставляется. Текущая строка настраивается на следующую строку после последней удаленной, за исключением случая, когда удалена последняя строка, т. е. `$`.

`0a`	Добавить текст в начало (то же, что `1i`)
`dp`	Удалить текущую строку, печатать следующую (или последнюю, если `$`)
`.,$dp`	Удалить отсюда до конца, печатать новую последнюю
`1,$d`	Удалить все
`?pat?,.-1d`	Удалить от предыдущей, совпадающей с '`pat`' до той, что перед текущей
`$dp`	Удалить последнюю строку, печатать новую последнюю
`$c`	Заменить последнюю строку (`$a` добавляет после последней строки)
`1,$c`	Заменить все строки

Подстановка, аннулирование. Нет необходимости перепечатывать целую строку, если в ней нужно заменить лишь несколько символов. Команда подстановки s заменяет одну последовательность символов другой:

`s/old/new/`	Заменить первую `old` на `new` в текущей строке
`s/old/new/p`	Заменить первую `old` на `new` и печатать строку
`s/old/new/g`	Заменить каждую `old` на `new` в текущей строке
`s/old/new/gp`	Заменить каждую `old` на `new` и печатать строку

Заменяется только самое левое вхождение образца в строке, если не написана буква '`g`'. Команда `s` выводит измененную строку только в том случае, когда она оканчивается буквой '`p`'. Фактически большинство команд `ed` выполняет свою работу "молча", но почти любая команда может быть завершена буквой `p` для вывода результата.

Если подстановкой вы не добились того, что хотели, с помощью команды `u` (аннулировать) можно уничтожить последнюю подстановку. Текущая строка должна быть настроена на преобразованную строку:

`u`	Аннулировать последнюю сделанную подстановку
`up`	Аннулировать последнюю подстановку и напечатать

Как вам уже известно, командам `p` и `d` могут предшествовать один или два номера, указывающие строки, на которые нужно воздействовать. Этот же принцип используется и для команды `s`.

`/old/s/old/new/`	Найти следующую `old`; заменить на `new`
`/old/s//new`	Найти следующую `old`; заменить на `new` (образец запоминается)
`1,$s/old/new/p`	Заменить первую `old` на `new` в каждой строке; печатать последнюю измененную строку
`1,$s/old/new/gp`	Заменить каждую `old` на `new` в каждой строке; печатать последнюю измененную строку

Отметим, что `1,$s` вызывает команду для обработки каждой строки, но это означает лишь самое левое вхождение образца в каждой строке; нужна заключительная команда '`g`', чтобы заместить все вхождения во всех строках. Кроме того, `p` выдает только последнюю измененную строку. Для вывода всех измененных строк необходима глобальная команда, которую мы вскоре рассмотрим.

Символ `&` означает сокращение; оказавшись где–либо справа от команды `s`, он заменяется образцом из левой части:

`s/big/very &/`	Заменить `big` на `very big`
`s/big/& &/`	Заменить `big` на `big big`
`s/.*/(&)/`	Взять в скобки целую строку (см. `.*` ниже)
`s/and/\&/`	Заменить `and` на `&` (\ — отключает специальное значение символа)

Метасимволы и регулярные выражения. Как и символы `*`, `>`, `:`, имеющие специальный смысл в `shell`, некоторые символы имеют специальный смысл для `ed`, если они появляются в образце для поиска или в левой части команды `s`. Эти символы называют метасимволами, а использующие их образцы — регулярными выражениями. В табл. 11.1 перечислены все символы и их значения. Примеры, приведенные ниже, следует читать в соответствии с таблицей. Специальный смысл любого символа может быть отменен предшествующей ему обратной дробной чертой '\'.

`c`	Любой специальный символ задает совпадение с таким же символом
`\c`	Отменяет специальный смысл символа `c`
`^`	Соответствует началу строки, когда `^` начинает образец
`$`	Соответствует концу строки, когда `$` заканчивает образец
`.`	Совпадает с любым одиночным символом
`[...]`	Соответствует одному любому символу в `...`; допустимы диапазоны типа `a-z`
`[^...]`	Соответствует любому одиночному символу, не входящему в `...`; допустимы диапазоны
`r*`	Соответствует нулевому или более числу вхождений `r`, где `r` символ, или `[...]`
`&`	Используется только в правой части `s`; вставляет фрагмент, совпавший с образом
`\(...\)`	Помечает регулярное выражение; найденная строка доступна как `\1`, и т. д. в левой и правой частях выражения

Таблица 11.1: Регулярные выражения редактора

Символу перевода строки не соответствует ни одно регулярное выражение.

Образец	**Соответствие**
`/^$/`	пустая строка, т.е. только конец строки
`/./`	непустая, т.е. по крайней мере один символ
`/^/`	все строки
`/thing/`	`thing` где–либо в строке
`/^thing/`	`thing` в начале строки
`/thing$/`	`thing` в конце строки
`/^thing$/`	строка, состоящая лишь из `thing`
`/thing.$/`	`thing` плюс любой символ в конце строки
`/thing\.$/`	`thing.` в конце строки
`/\/thing\//`	`/thing/` где–либо в строке
`/[tT]hing/`	`thing` или `Thing` где–либо в строке
`/thing[0-9]/`	`thing`, за которой одна цифра
`/thing[^0-9]/`	`thing`, за которой не цифра
`/thing[0-9][^0-9]/`	`thing`, за которой цифра и не цифра
`/thing1.*thing2/`	`thing1`, затем любая строка, затем `thing2`
`/^thing1.*thing2$/`	`thing1` в начале и `thing2` в конце

Регулярные выражения, использующие *, выбирают самое левое совпадение с образцом до тех пор, пока это возможно. Отметим, что x* может соответствовать нулю, а xx* одному или более символу.

Глобальные команды. Глобальные команды g и v управляют вызовом одной или большего числа других команд, выполняющих преобразования в множестве строк, выбранных регулярным выражением. Команда g наиболее часто используется для печати, подстановки или удаления множества строк:

m,ng/re/cmd	Для всех строк между m и n, которые соответствуют re выполнить cmd
m,nv/re/cmd	Для всех строк между m и n, которые не соответствуют re выполнить cmd

Командам g и v могут предшествовать номера строк, ограничивающие диапазон; по умолчанию принимается диапазон 1,$:

g/.../p	Печатать все строки, соответствующие регулярному выражению ...
g/.../d	Убрать все строки соответствующие ...
g/.../s//repl/p	Заменить первое вхождение ... в каждой строке на 'repl', печатать измененные строки
g/.../s//repl/gp	Заменить каждое ... на 'repl' , печатать измененные строки
g/.../s/pat/repl/	В строках, соответствующих ..., заменить первую 'pat' на 'repl'
g/.../s/pat/repl/p	В строках, соответствующих ..., заменить первую 'pat' на 'repl' и печатать
g/.../s/pat/repl/gp	В строках, соответствующих ..., заменить все 'pat' на 'repl' и печатать
v/.../s/pat/repl/gp	В строках, не соответствующих ..., заменить все 'pat' на 'repl' и печатать
v/^$/p	Печатать все непустые строки
g/.../cmd1\ cmd2\ cmd3	Выполнять составные команды с единственной g, присоединить \ к каждой cmd кроме последней

Команды, управляемые командами g или v, также могут использовать номера строк, текущая строка настраивается по очереди на каждую выбранную строку:

g/thing/.,.+1p	Печатать каждую строку с thing и следующую
g/^\.EQ/.1,/^\.EN/s/alpha/beta/gp	Заменять alpha на beta только между .EQ и .EN и печатать измененные строки

Перемещение и копирование строк. Команда m перемещает группу смежных строк, а команда t копирует группу строк:

m,n,md	Переместить строки m по n за строку d
m,n,td	Скопировать строки m по n за строку d

Если исходные строки не определены, используется текущая строка. Строка назначения `d` не может быть в диапазоне `m,n-1`. Ниже приведено несколько общих идиом, включающих `m` и `t`.

`m+`	Поместить текущую строку после следующей (переставить)
`m-2`	Поместить текущую строку перед предыдущей
`m--`	То же самое: — это то же, что -2
`m-`	Ничего не делать
`m$`	Поместить текущую строку в конец (`m0` — поместить в начало)
`t.`	Дублировать текущую строку (`t$` дублирует в конце)
`-,.t.`	Дублировать предыдущую и текущую строки
`1,$t$`	Дублировать все множество строк
`g/^/m0`	Инвертировать порядок строк

Метки и номера строк. Команда `=` печатает номер строки `$` (слабое умолчание), `.=` печатает номер текущей строки и т. д. Положение текущей строки не изменяется.

Команда `kc` метит нужную строку буквой `c`; впоследствии на эту строку можно ссылаться с помощью `'c`. Команда `k` не меняет положение текущей строки. Метки удобны при перемещении больших фрагментов текста, поскольку они остаются привязанными к строкам, как показано в приведенной ниже последовательности:

`/.../ka`	Найти строку `...` и пометить буквой `a`
`/.../kb`	Найти строку `...` и пометить буквой `b`
`'a,'bp`	Печатать целый диапазон, чтобы быть уверенным
`/.../`	Найти нужную строку
`'a,'bm`	Поместить выбранные строки после нее

Объединение, расщепление и реорганизация строк. Строки могут быть объединены с помощью команды `j` (пробелы не добавляются):

`m,nj`	объединяет строки с `m` по `n` в одну

По умолчанию принимается диапазон `.,.+1`;

`jp`	Объединить текущую строку со следующей и печатать
`-,.jp`	Объединить предыдущую строку с текущей и печатать

Строки можно расщепить командой подстановки, отделив новую строку:

`s/part1part2/part1\` `part2/`	Расщепить строку на две части
`s/ /\` `/g`	Расщепить по каждому пробелу Оставить одно слово на строку

Текущей становится последняя созданная строка.

Чтобы манипулировать не только целыми фрагментами, выбираемыми регулярными выражениями, но и их соответствующими частями, используйте помеченные регулярные выражения: если конструкция `\(...\)` появляется в регулярном выражении, то часть соответствующего ей фрагмента доступна как `\1`. Возможно до девяти помеченных выражений, на которые ссылаются с помощью `\1`, `\2` и т. д.

`s/\(...\)\(.*\)/\2\1/`	Поместить 3 первых символа в конец
`/\(..*\)\1/`	Найти строки, содержащие повторяющиеся смежные цепочки символов

Команды, работающие с файлами.

Командам `r` и `w` (читать и писать) могут предшествовать номера строк:

`nr file`	Читать `file`; добавить его после строки `n`; текущей становится последняя прочитанная строка
`m,nw file`	Писать строки `m-n` в `file`; положение текущей строки не изменяется
`m,nw file`	Добавить строки `m-n` к `file` положение текущей строки не изменяется

По умолчанию диапазон для `w` и `W` (команда `W` приведена,ниже в табл. 11.2) — это целый файл. Значение `n` по умолчанию для `r` равно `$`, что представляется не очень удачным. Будьте внимательны.

Редактор `ed` запоминает первое использованное имя файла из командной строки или из команд `r`, `w`. Команда `f` (файл) печатает или заменяет имя запомненного файла:

`f`	Печатать имя запомненного файла
`f file`	Установить запомненное имя на '`file`'

Команда `e` (редактировать) вновь вызывает `ed` с запомненным или новым файлом:

`e`	Начать редактировать запомненный файл
`e file`	Начать редактировать '`file`'

Команда `e` защищена тем же способом, что и `q`: если вы не записали измененную версию, первая команда `e` выдает сообщение об ошибке; `e` вновь инициализирует редактор независимо от внесения изменений. В некоторых системах `ed` связан с `e`, так что одна и та же команда (`e filename`) может использоваться внутри и вне редактора.

Шифрование. Файлы могут быть зашифрованы по записи и дешифрованы при чтении с помощью команды `x`; пароль будет запрошен. Шифрование происходит тем же способом, что и в `crypt(1)`. В некоторых системах команда `x` заменена на `X` (прописную букву) во избежание случайностей.

Сводка команд. В табл. 11.2 перечислены команды редактора, а в табл. 11.3 — допустимые номера строк. Каждой команде предшествует нуль, один или два номера строк, указывающие число используемых строк, если их нет, принимается соглашение по умолчанию. За большинством команд может следовать буква p для вывода последней обработанной строки или `l` для формата списка. Текущей обычно становится последняя обработанная строка; настройка не меняется командами `f`, `k`, `w`, `x`, `=`, `!`.

```
%Упражнение.
  Если вы думаете, что знаете \cmd{ed}, попробуйте выполнить текст
  (см.  справочное руководство по \code{quiz(6)}).
```

`.a`	Выполнять ввод до тех пор, пока не напечатана строка, содержащая только '.'
`.,.c`	Заменить строки, новый текст заканчивается так же, как для команды `a`
`.,.d`	Исключить строки
`e file`	Вновь начать редактировать `file`. Редактирование начинается даже в том случае, если исправления не записаны
`f file`	Запомнить имя файла как `file`
`1,$g/re/cmds`	Выполнить `cmds` для каждой строки, соответствующей регулярному выражению `re`; отдельные команды в `cmds` разделены `\newline` (`\+"перевод строки"`)
`.i`	Вставить текст перед строкой; он заканчивается так же, как для команды `a`
`.,.+1j`	Соединить строки в одну
`.kc`	Пометить строку буквой `c`
`...l`	Перечислить строки, делая невидимые символы видимыми
`.,.m line`	Переместить строки после строки `line`
`.,.p`	Печатать строки
`q`	Выйти. `Q` выходит, даже если исправления не записаны
`$r file`	Читать `file`
`.,.s/re/new/`	Заменить `new` на то, что соответствует `re`
`.,. t line`	Скопировать строки после `line`
`.u`	Аннулировать последнюю подстановку в строке (только одну)
`1,$v/re/cmds`	Выполнить команды `ed cmds` для каждой строки, не соответствующей re
`1,$w file`	Записать строки в файл; `W` добавляет (строки к файлу) вместо того чтобы записывать (как новый файл)
`x`	Войти в режим шифрования (или `ed -x имя_файла`)
`$=`	Печатать номер строки
`! cmdline`	Выполнить команду UNIX `cmdline`
`(.+1) newline`	Печатать строку

Таблица 11.2: Сводка команд `ed`

`n`	Абсолютный номер строки `n`, $n = 0, 1, 2, \ldots$
`.`	Текущая строка
`$`	Последняя строка текста
`/re/`	Следующая строка, соответствующая `re`; после последней `$` циклическое движение к первой строке
`?re?`	Предыдущая строка, соответствующая `re`; после первой циклическое движение к последней `$`
`'c`	Строка с меткой `c`
`N1+/-n`	Строка `N1+/-n` (аддитивная комбинация)
`N1,N2`	Строки с `N1` по `N2`
`N1;N2`	Команда: сделать строку `N1` текущей, затем вычислить `N2`. `N1` и `N2` могут быть определены любым из перечисленных выше способов

Таблица 11.3: Номера строк в `ed`

Глава 12

Дополнение: Справочное руководство по `hoc`

`Hoc` — диалоговый язык для арифметики с плавающей точкой

Б. Керниган Р. Пайк

РЕЗЮМЕ

`Hoc` – это простой программируемый интерпретатор для выражений с плавающей точкой. Он обеспечивает поток управления в стиле Си, определения функций и обычные числовые встроенные функции, такие, как косинус и логарифм.

Выражения

Hoc представляет язык выражений, во многом подобный Си: хотя он и содержит несколько управляющих операторов, большинство операторов, например присваивания, суть выражения, значения которых не принимаются во внимание. Так, оператор присваивания `=` присваивает значение своей правой части левому операнду и вырабатывает значение, используемое в многократном присваивании. Грамматика выражений имеет вид:

```
выражение  :  число
           |  переменная
           |  (выражение)
           |  выражение бинарная_операция выражение
           |  унарная_операция выражение
           |  функция(аргументы)
```

Числа представляются с плавающей точкой. Формат ввода распознается с помощью `scanf(3)`: цифры, десятичная точка, цифры, `e` или `E`, показатель степени со знаком. Должна присутствовать по крайней мере одна цифра или десятичная точка; другие компоненты являются необязательными.

Имена переменных формируются из букв, за которыми следуют строки букв и цифр. Здесь `бинарная_операция` означает двуместные операции, такие, как сложение или логическое сравнение, а `унарная_операция` — две операции отрицания: '`!`' (логическое отрицание НЕ) и '`-`' (арифметическое отрицание, перемена знака). Все операции перечислены в табл. 12.1.

Функции, как описывается ниже, могут быть определены пользователем. Аргументы функций - это выражения, разделяемые запятыми. В табл. 12.2 перечислено несколько встроенных функций, имеющих по одному аргументу.

`^`	Возведение в степень (**FORTRAN** `**`), правоассоциативна
`! -`	Одноместные логическое и арифметическое отрицания
`* /`	Умножение, деление
`+ -`	Сложение, вычитание
`> >=`	Операции отношения: больше, больше или равно
`< <=`	Меньше, меньше или равно
`== !=`	Равно, не равно (все отношения одинакового приоритета)
`&&`	Логическое И (оба операнда всегда вычисляются)
`||`	Логическое ИЛИ (оба операнда всегда вычисляются)
`=`	Присваивание, правоассоциативна

Таблица 12.1: Операции по порядку уменьшения приоритета

`abs(x)`	Абсолютная величина `x`
`atan(x)`	Арктангенс `x`
`cos(x)`	Косинус `x`
`exp(x)`	Экспонента `x`
`int(x)`	Целая часть `x`, усеченная в сторону нуля
`log(x)`	Натуральный логарифм `x`
`log10(x)`	Десятичный логарифм `x`
`sin(x)`	Синус `x`
`sqrt(x)`	Корень квадратный из `x`

Таблица 12.2: Встроенные функции

Логические выражения имеют значения `1` (истина) и `0` (ложь). Как и в Си, любое ненулевое значение означает истину. При всех операциях над числами с плавающей точкой сравнения на равенство могут быть неточными. Кроме того, `hoc` имеет несколько встроенных констант, приведенных в табл. 12.

Таблица П.2.3 ВСТРОЕННЫЕ КОНСТАНТЫ

`DEG`	57.2957795130823208768	$180/\pi$, градусы на радианы
`E`	2.71828182845904523536	e, основание натуральных логарифмов
`GAMMA`	0.57721566490153286060	γ , константа Эйлера–Масчерони
`PHI`	1.61803398874989484820	$\frac{\sqrt{5}+1}{2}$, золотое сечение
`PI`	3.14159265358979323846	π, круговое трансцендентное число

Операторы и поток управления

Операторы `hoc` имеют следующую грамматику:

```
оператор  :  выражение
          |  переменная = выражение
          |  процедура (список, аргументов)
          |  while (выражение) оператор
          |  if (выражение) оператор
          |  if (выражение) оператор else оператор
          |  список_операторов
          |  print список_выражений
          |  return возможное_выражение

список_операторов  :  (пусто)
                   |  список_операторов оператор
```

Присваивание распознается по умолчанию как оператор, а не как выражение, поэтому после ввода в диалоге присваивании их значения не печатаются.

Отметим, что символ `;` не является для `hoc` специальным: оператор оканчивается символом перевода строки. Это обусловливает некоторые особенности. Ниже показан допустимый оператор `if`:

```
if (x < 0 ) print(y) else print (z)
if (x < 0 ) {
    print(y)
} else {
    print(z)
}
```

Во втором примере скобки не обязательны: символ перевода строки после `if` оканчивал бы оператор и вызывал бы синтаксическую ошибку там, где опущены скобки.

Синтаксис и семантика средств управления в `hoc` в основном те же, что и в Си. Одинаковы также `while` и `if`, однако в `hoc` нет операторов `break` и `continue`.

Ввод и вывод: `read` и `print`

Функция ввода `read` (читать) имеет, подобно другим встроенным функциям, один аргумент: однако он не является выражением: это имя переменной. Следующее число, как определено выше, читается из стандартного входного потока и присваивается поименованной переменной. Функция `read` возвращает значения 1 (истина), если величина была прочитана, и 0 (ложь), если `read` встретила конец файла либо ошибку.

Выходной поток порождается оператором `print`. Аргументы `print` составляют разделяемый запятыми список выражений и строк, взятых в кавычки, как в Си. Символы перевода строки должны добавляться: `print` их никогда автоматически не вводит.

Отметим, что `read` есть специальная встроенная функция и поэтому получает один аргумент в скобках, тогда как `print` — оператор, получающий список, разделяемый запятыми без скобок:

```
while (read (x)) {
    print "value is", x, "\n"
}
```

Функции и процедуры

Функции и процедуры в `hoc` различаются, хотя и определены одним и тем же механизмом. Это различие введено просто для контроля ошибок во время исполнения: возврат значения является ошибкой для процедуры, для функции же ошибочно не возвращать значения.

Синтаксис определения таков:

```
function: func имя () оператор
procedure: proc имя() оператор
```

Здесь имя может быть именем некоторой переменной — встроенные функции исключаются. Определение, вплоть до открывающейся скобки оператора, должно помещаться на одной строке, как в приведенном выше операторе `if`.

В отличие от Си тело функции или процедуры может быть любым оператором, не обязательно составным (в скобках). Поскольку символ `;` не имеет своего значения в `hoc`, пустое тело процедуры формируется пустой парой скобок.

Функции и процедуры при вызовах могут получать аргументы, отделенные запятыми. На аргументы ссылаются так же, как в `shell`: `$3` относится к третьему, индексируемому, начиная с единицы, аргументу. Они передаются значениями и внутри функций семантически эквивалентны переменным. Ссылка на аргумент с помощью числа, превышающего число аргументов, переданных процедуре, считается ошибкой. Контроль ошибок — динамический, поскольку подпрограмма может иметь переменное число параметров, если ее начальные аргументы влияют на это число (см. функцию `printf` в Си).

Функции и процедуры могут быть рекурсивными, но стек имеет ограниченную глубину (около сотни вызовов).

Примеры.

- Ниже показано определение функции Аккерманна в `hoc`:

```
$ hoc
time 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: стэк слишком велик (строка 8)

...
```

- Формула Стирлинга:

$$n! \sim \sqrt{2n\pi}(\frac{n}{e})^n(1+\frac{1}{12n})$$

```
$ hoc
func stirl() {
    return sqrt(2 * $1 * PI) * ($1 / E) ^ $1 * (1 + 1 / (12 * $1))
}
stirl(10)
3628684.7
stirl(20)
2.4328818e+18
```

- Функция факториал $n!$

```
func fac() if ($1 <= 0) return 1 else return $1 * fac($1 - 1)
```

Отношение факториала к приближению 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
```

Глава 13

Дополнение: Исходные тексты калькулятора `hoc`

These files contain all the code from "The Unix Programming Environment", by Brian Kernighan and Rob Pike (Prentice–Hall, 1984, ISBN 0-13-937681-X). A separate hoc6 distribution contains any fixes that we have applied to that; the version in this file is from the book.

13.1 `hoc1`

13.1.1 `makefile`

```
hoc1: hoc.o
        cc hoc.o -o hoc1
```

13.1.2 `hoc.y`

```
%{
```

```
#define YYSTYPE double  /* data type of yacc stack */
%}
%token  NUMBER
%left   '+' '-'   /* left associative, same precedence */
%left   '*' '/'   /* left assoc., higher precedence */
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */

#include <stdio.h>
#include <ctype.h>
char    *progname;      /* for error messages */
int     lineno = 1;

main(argc, argv)        /* hoc1 */
    char *argv[];
{
    progname = argv[0];
    yyparse();
}

yylex()         /* hoc1 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {       /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    if (c == '\n')
        lineno++;
    return c;
```

```
}

yyerror(s)      /* called for yacc syntax error */
    char *s;
{
    warning(s, (char *) 0);
}

warning(s, t)   /* print warning message */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

13.1.3 hoc1.y, версия 1.5

```
%{
#define YYSTYPE double  /* data type of yacc stack */
%}
%token  NUMBER
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%%

list:     /* nothing */
        | 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 { $$ = $1 / $3; }
        | '-' expr  %prec UNARYMINUS { $$ = -$2; } /* new */
        | '(' expr ')'  { $$ = $2; }
        ;
%%
        /* end of grammar */

#include <stdio.h>
#include <ctype.h>
char    *progname;      /* for error messages */
int     lineno = 1;
```

```
main(argc, argv)        /* hoc1 */
    char *argv[];
{
    progname = argv[0];
    yyparse();
}

yylex()         /* hoc1 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {    /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    if (c == '\n')
        lineno++;
    return c;
}

yyerror(s)
    char *s;
{
    warning(s, (char *)0);
}

warning(s, t)
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

13.2 hoc2

13.2.1 hok.y

```
%{
double  mem[26];        /* memory for variables 'a'..'z' */
%}
%union {                /* stack type */
        double  val;    /* actual value */
        int     index;  /* index into mem[] */
}
%token  <val>   NUMBER
%token  <index> VAR
%type   <val>   expr
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */

#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#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();
}

yylex()         /* hoc2 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {       /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    if (islower(c)) {
        yylval.index = c - 'a';    /* ASCII only */
        return VAR;
    }
    if (c == '\n')
        lineno++;
    return c;
}

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);
    longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
```

```
    execerror("floating point exception", (char *) 0);
}

warning(s, t)   /* print warning message */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

13.2.2 makefile

```
hoc2:   hoc.o
        cc hoc.o -o hoc2
```

13.3 hoc3

13.3.1 makefile

```
YFLAGS = -d             # force creation of y.tab.h
OBJS = hoc.o init.o math.o symbol.o     # abbreviation

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]
```

13.3.2 hoc.h

```
typedef struct Symbol { /* symbol table entry */
    char        *name;
    short       type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;            /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } u;
    struct Symbol    *next;     /* to link to another */
} Symbol;
Symbol    *install(), *lookup();
```

13.3.3 hoc.y

```
%{
#include "hoc.h"
extern  double  Pow();
%}
%union {
        double  val;    /* actual value */
        Symbol  *sym;   /* symbol table pointer */
}
%token  <val>   NUMBER
%token  <sym>   VAR BLTIN UNDEF
%type   <val>   expr asgn
%right  '='
%left   '+' '-'
```

```
%left   '*' '/'
%left   UNARYMINUS
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
        | 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; }
        ;
%%
        /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

main(argc, argv)        /* hoc3 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
```

```
    yyparse();
}

yylex()         /* hoc3 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {       /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    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;
    }
    if (c == '\n')
        lineno++;
    return c;
}

yyerror(s)
    char *s;
{
    warning(s, (char *)0);
}

execerror(s, t) /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
```

```
{
    execerror("floating point exception", (char *) 0);
}

warning(s, t)
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

13.3.4 init.c

```
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Exp(), Sqrt(), integer();

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 argument */
    "log10",    Log10,  /* checks argument */
    "exp",      Exp,    /* checks argument */
    "sqrt",     Sqrt,   /* checks argument */
    "int",      integer,
    "abs",      fabs,
    0,  0
};
```

```
init()  /* install constants and built-ins in table */
{
    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;
    }
}
```

13.3.5 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;
}
```

13.3.6 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;
}
```

13.4 hoc3 с lex

13.4.1 hoc.h

```
typedef struct Symbol { /* symbol table entry */
    char        *name;
    short       type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;            /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } u;
    struct Symbol       *next;  /* to link to another */
} Symbol;
Symbol  *install(), *lookup();
```

13.4.2 hoc.y

```
%{
#include "hoc.h"
extern  double  Pow();
%}
%union {
    double      val;    /* actual value */
    Symbol      *sym;   /* symbol table pointer */
}
%token  <val>   NUMBER
```

```
%token  <sym>   VAR BLTIN UNDEF
%type   <val>   expr asgn
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
    | 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; }
    ;
%%
    /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <setjmp.h>
jmp_buf begin;

main(argc, argv)        /* hoc3 */
    char *argv[];
{
    progname = argv[0];
    init();
    setjmp(begin);
```

```
    yyparse();
}

yyerror(s)
    char *s;
{
    warning(s, (char *)0);
}
execerror(s, t)
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}
warning(s, t)
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

13.4.3 init.c

```
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Exp(), Sqrt(), integer();

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 argument */
    "log10", Log10,     /* checks argument */
    "exp",      Exp,    /* checks argument */
    "sqrt",     Sqrt,   /* checks argument */
    "int",      integer,
    "abs",      fabs,
    0,  0
};

init()  /* install constants and built-ins in table */
{
    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;
    }
}
```

13.4.4 lex.l

```
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t]   { ; }   /* skip blanks and tabs */
[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'; }   /* everything else */
.       { return yytext[0]; }
```

13.4.5 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
```

13.4.6 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;
}
```

13.4.7 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;
}
```

13.5 hoc4

13.5.1 code.c

```
#include "hoc.h"
#include "y.tab.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 */

initcode()      /* initialize for code generation */
{
    stackp = stack;
    progp = prog;
}

push(d)         /* push d onto stack */
    Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack overflow", (char *) 0);
    *stackp++ = d;
}

Datum pop()     /* pop and return top elem from stack */
{
    if (stackp <= stack)
        execerror("stack underflow", (char *) 0);
    return *--stackp;
}

constpush()     /* push constant onto stack */
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()       /* push variable onto stack */
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
```

```
    push(d);
}

bltin()         /* evaluate built-in on top of stack */
{
    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 == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}

add()           /* add top two elems on stack */
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}

sub()   /* subtract top of stack from next */
{
    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);
}

power()
{
    Datum d1, d2;
    extern double Pow();
    d2 = pop();
    d1 = pop();
    d1.val = Pow(d1.val, d2.val);
    push(d1);
}
assign()        /* assign top value to next value */
{
    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);
}
```

```
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)      /* run the machine */
    Inst *p;
{
    for (pc = p; *pc != STOP; )
        (*(*pc++))();
}
```

13.5.2 hoc.h

```
typedef struct Symbol { /* symbol table entry */
    char        *name;
    short       type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;            /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } 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();

typedef int (*Inst)();  /* machine instruction */
#define STOP    (Inst) 0

extern  Inst prog[];
extern  eval(), add(), sub(), mul(), div(), negate(), power();
extern  assign(), bltin(), varpush(), constpush(), print();
```

13.5.3 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 */
}
%token  <sym>   NUMBER VAR BLTIN UNDEF
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
    | 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); }
    ;
%%
    /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

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;
}

static int c;   /* global for use by warning() */
yylex()         /* hoc4 */
{
    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {       /* number */
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
    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;
    }
    if (c == '\n')
        lineno++;
    return c;
}
yyerror(s)
    char *s;
{
    warning(s, (char *)0);
}

execerror(s, t) /* recover from run-time error */
    char *s, *t;
```

```
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
    execerror("floating point exception", (char *) 0);
}

warning(s, t)
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
    while (c != '\n' && c != EOF)
        c = getchar();          /* flush rest of input line */
}
```

13.5.4 init.c

```
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Exp(), Sqrt(), integer();

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 argument */
    "log10", Log10,     /* checks argument */
    "exp",      Exp,    /* checks argument */
    "sqrt",     Sqrt,   /* checks argument */
    "int",      integer,
    "abs",      fabs,
    0,  0
};

init()  /* install constants and built-ins in table */
{
    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;
    }
}
```

13.5.5 makefile

```
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]
```

13.5.6 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;
}
```

13.5.7 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;
}
```

13.6 hoc5

13.6.1 code.c

```
#include "hoc.h"
#include "y.tab.h"

#define NSTACK  256

static Datum stack[NSTACK];
static Datum *stackp;

#define NPROG   2000
Inst prog[NPROG];
static Inst *pc;
Inst *progp;

initcode()
{
    progp = prog;
    stackp = stack;
}

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;  /* loop body */

    execute(savepc+2);  /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc)));  /* body */
        execute(savepc+2);
        d = pop();
    }
    pc = *((Inst **)(savepc+1));  /* next statement */
}

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)));
    pc = *((Inst **)(savepc+2));         /* next stmt */
}

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()
{
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}

prexpr()        /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g\n", d.val);
}

Inst *code(f)   /* install one instruction or operand */
    Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("expression too complicated", (char *)0);
    *progp++ = f;
    return oprogp;
}

execute(p)
    Inst *p;
{
    for (pc = p; *pc != STOP; )
        (*(*pc++))();
}
```

13.6.2 fib

```
{
a=0
b=1
```

```
while(b<1000){
    c=b
    b=a+b
    a=c
    print(c)
}
}
```

13.6.3 fib2

```
{
n=0
a=0
b=1
while(b<10000000){
    n=n+1
    c=b
    b=a+b
    a=c
    print(b)
}
print(n)
}
```

13.6.4 hoc.h

```
typedef struct Symbol { /* symbol table entry */
    char        *name;
    short       type;   /* VAR, BLTIN, UNDEF */
    union {
        double  val;            /* if VAR */
        double  (*ptr)();       /* if BLTIN */
    } 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();

typedef int (*Inst)();  /* machine instruction */
#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();
```

13.6.5 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 */
}
%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:     /* nothing */
    | 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;     /* 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); }
    ;
end:      /* nothing */         { code(STOP); $$ = progp; }
    ;
stmtlist: /* nothing */         { $$ = 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); }
    ;
%%
    /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int     defining;

int c;  /* global for use by warning() */
```

```
yylex()         /* hoc5 */
{
    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {       /* number */
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
    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;
    }
    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;
    }
}
follow(expect, ifyes, ifno)  /* look ahead for >=, etc. */
{
    int c = getchar();

    if (c == expect)
        return ifyes;
    ungetc(c, stdin);
    return ifno;
}
yyerror(s)
    char *s;
```

```
{
    warning(s, (char *)0);
}

execerror(s, t) /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
    execerror("floating point exception", (char *) 0);
}
main(argc, argv)
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); yyparse(); initcode())
        execute(prog);
    return 0;
}
warning(s, t)
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
    while (c != '\n' && c != EOF)
        c = getchar();  /* flush rest of input line */
    fseek(stdin, 0L, 2);                /* flush rest of file */
    longjmp(begin, 0);
}
```

13.6.6 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[] = {
    "if",               IF,
    "else",             ELSE,
    "while",    WHILE,
    "print",    PRINT,
    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 argument */
    "log10", Log10,     /* checks argument */
    "exp",      exp,
    "sqrt",     Sqrt,   /* checks argument */
    "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;
    }
}
```

13.6.7 makefile

```
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc5:   $(OBJS)
    cc $(OBJS) -lm -o hoc5

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]
```

13.6.8 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;
}
```

13.6.9 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;
}
```

13.7 hoc6

13.7.1 ack

```
func ack() {
    n = n+1
    if($1 == 0) return ($2+1)
    if($2 == 0)          return (ack($1 - 1, 1))
    return (ack($1 - 1, ack($1, $2 - 1)))
}
n=0
ack(3,3)
print n, "calls\n"
```

13.7.2 ack1

```
func ack() {
    n = n+1
    if($1 == 0) return ($2+1)
    if($2 == 0)          return (ack($1 - 1, 1))
    return (ack($1 - 1, ack($1, $2 - 1)))
}
n=0
while (read(x)) {
    read(y)
    print ack(x,y), "\n"
}
print n,"\n"
```

13.7.3 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]))();
}
```

13.7.4 double

```
proc double(){
    if($1 > 1){
        double($1/2)
```

```
    }
    print($1)
}
double(1024)
```

13.7.5 fac

```
func fac() {
    if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
```

13.7.6 fac1

```
func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
fac(0)
fac(7)
fac(10)
```

13.7.7 fac2

```
func fac() {
    if ($1 <= 0) {
        return 1
    }
    return $1 * fac($1-1)
}
i=0
while(i<=20){
    print "factorial of ", i, "is ", fac(i), "\n"
    i=i+1
}
```

13.7.8 fib

```
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        print b
        c = b
        b = a+b
        a = c
    }
    print "\n"
}
```

13.7.9 fib2

```
{
```

```
n=0
a=0
b=1
while(b<10000000){
    n=n+1
    c=b
    b=a+b
    a=c
    print(b)
}
print(n)
}
```

13.7.10 fibsum

```
proc fib(){
    a=1
    b=1
    c=2
    d=3
    sum = a+b+c+d
    while(d<$1){
        e=d+c
        print(e)
        a=b
        b=c
        c=d
        d=e
        sum=sum+e
    }
    print(sum)
}

fib(1000)
```

13.7.11 fibtest

```
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        c = b
        b = a+b
        a = c
    }
}

i = 1
```

```
while (i < 1000) {
    fib(1000)
    i = i + 1
}
```

13.7.12 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();
```

13.7.13 hoc.ms

```
.EQ
delim @@
.EN
.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.
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.
The input format is
that recognized by @scanf@(3):
.ix [scanf]
digits, decimal point, digits,
.ix [hoc] manual
.ix assignment expression
.ix multiple assignment
@e@ or @E@, signed exponent.
At least one digit or a decimal point
must be present;
the other components are optional.
.PP
Variable names are formed from a letter
followed by a string of letters and numbers.
@binop@ refers to binary operators such
```

```
as addition or logical comparison;
@unop@ refers to the two negation operators,
‘!’ (logical negation, ‘not’)
and ‘\-’ (arithmetic negation, sign change).
Table 1 lists the operators.
.TS
center, box;
c s
lfCW 1.
\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
.ix table~of [hoc] operators
.PP
Functions, as described later, may be defined by the user.
Function arguments are expressions separated by commas.
There are also a number of built-in functions,
all of which take a single argument,
described in Table 2.
.TS
center, box;
c s
lfCW 1.
\fBTable 2:\fP  Built-in Functions
.sp .5
abs(x)  @| x |@, absolute value of @x@
atan(x) arc tangent of @x@
cos(x)  @cos (x)@, cosine of @x@
exp(x)  @e sup x@, exponential of @x@
int(x)  integer part of @x@, truncated towards zero
log(x)  @log (x)@, logarithm base @e@ of @x@
log10(x)        @log sub 10 (x)@, logarithm base 10 of @x@
sin(x)  @sin (x)@, sine of @x@
sqrt(x) @sqrt x@, @x sup half@
.TE
.ix table~of [hoc] functions
.PP
Logical expressions have value 1.0 (true) and 0.0 (false).
```

```
As in C,
any non-zero value is taken to be true.
As is always the case with floating point numbers,
equality comparisons are inherently suspect.
.PP
.I Hoc
also has a few built-in constants, shown in Table 3.
.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
.ix table~of [hoc] constants
.NH
Statements and Control Flow
.PP
.I Hoc
statements have the following grammar:
.DS
.I
stmt:           expr
    |   variable = expr
    |   procedure ( arglist )
    |   while ( expr ) stmt
    |   if ( expr ) stmt
    |   if ( expr ) stmt else stmt
    |   { stmtlist }
    |   print expr-list
    |   return optional-expr

stmtlist:               \fR(nothing)\fI
    |   stmlist stmt
.R
.DE
An assignment is parsed by default as a statement rather than
an expression, so assignments typed interactively do not print their value.
.PP
Note that semicolons are not special to
.ix [hoc] input~format
@hoc@: statements are terminated by newlines.
This causes some peculiar behavior.
```

```
The following are legal
.IT if
statements:
.DS
.ft CW
if (x < 0) print(y) else print(z)

if (x < 0) {
    print(y)
} else {
    print(z)
}
.ft
.DE
In the second example, the braces are mandatory:
the newline after the
.I if
would terminate the statement and produce a syntax error were
the brace omitted.
.PP
The syntax and semantics of @hoc@
control flow facilities are basically the same as in C.
The
.I while
and
.I if
statements are just as in C, except there are no @break@ or
@continue@ statements.
.NH
Input and Output: @read@ and @print@
.PP
.ix [hoc] [read]~statement
.ix [hoc] [print]~statement
The input function @read@, like the other built-ins,
takes a single argument.  Unlike the built-ins, though, the argument
is not an expression: it is the name of a variable.
The next number (as defined above) is read from the standard input
and assigned to the named variable.
The return value of @read@ is 1 (true) if a value was read, and 0 (false)
if @read@ encountered end of file or an error.
.PP
Output is generated with the @print@ statement.
The arguments to @print@ are a comma-separated list of expressions
and strings in double quotes, as in C.
Newlines must be supplied; they are never provided automatically by @print@.
.PP
Note that @read@ is a special built-in function, and therefore takes a
single parenthesized argument, while @print@ is a statement that takes
```

```
a comma-separated, unparenthesized list:
.DS
.ft CW
while (read(x)) {
    print "value is ", x, "\en"
}
.ft
.DE
.NH
Functions and Procedures
.PP
Functions and procedures are distinct in @hoc@,
although they are defined by the same mechanism.
This distinction is simply for run-time error checking:
it is an error for a procedure to return a value,
and for a function @not@ to return one.
.PP
The definition syntax is:
.ix [hoc] function~definition
.ix [hoc] procedure~definition
.DS
.I
.ta 1i
function:       func name() stmt

procedure:      proc name() stmt
.R
.DE
.I name
may be the name of any variable \(em built-in functions are excluded.
The definition, up to the opening brace or statement,
must be on one line, as with the
.I if
statements above.
.PP
Unlike C,
the body of a function or procedure may be any statement, not
necessarily a compound (brace-enclosed) statement.
Since semicolons have no meaning in @hoc@,
a null procedure body is formed by an empty pair of braces.
.PP
Functions and procedures may take arguments, separated by commas,
when invoked.  Arguments are referred to as in the shell:
.ix [hoc] arguments
.IT $3
refers to the third (1-indexed) argument.
They are passed by value and within functions
are semantically equivalent to variables.
```

```
It is an error to refer to an argument numbered greater than the
number of arguments passed to the routine. The error checking
is done dynamically, however, so a routine may have variable numbers
of arguments if initial arguments affect the number of arguments
to be referenced (as in C's @printf@).
.PP
Functions and procedures may recurse, but the stack has limited depth
(about a hundred calls).
The following shows a
.I hoc
definition of Ackermann's function:
.ix Ackermann's~function
.DS
.ft CW
.ix [ack]~function
.S $ "hoc
.S "func ack() {
.S "        if ($1 == 0) return $2+1
.S "        if ($2 == 0) return ack($1-1, 1)
.S "        return ack($1-1, ack($1, $2-1))
.S "}
.S "ack(3, 2)
        29
.S "ack(3, 3)
        61
.S "ack(3, 4)
hoc: stack too deep near line 8
\&...
.ft
.DE
.bp
.NH
Examples
.PP
Stirling's formula:
.ix Stirling's~formula
.EQ
n! ~\(ap~ sqrt {2n pi} (n/e) sup n (1+ 1 over 12n )
.EN
.DS
.ft CW
.S $ hoc
.S "func stirl() {
.S "    return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
.S "}
.S "stirl(10)
        3628684.7
.S stirl(20)
```

```
        2.4328818e+18
.ft R
.DE
.PP
Factorial function, @n!@:
.ix [fac]~function
.DS
.S "func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
.ft R
.DE
.PP
Ratio of factorial to Stirling approximation:
.DS
.S "i = 9
.S "while ((i = i+1) <= 20) {
.S \ \ \ \ \ \ \ \ print\ i,\ "\ \ ",\ fac(i)/stirl(i),\ "\en"
.S "}
.ft CW
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
.ft
.DE
```

13.7.14 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++;
}
```

13.7.15 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;
    }
}
```

13.7.16 makeapp

```
#!/bin/sh

cd hoc6
for i in hoc.y hoc.h symbol.c code.c init.c math.c makefile
do
    echo "
*****   $i   ***********************************************************
"
sed 's/\\/\\e/g
     s/^$/.sp .5/'  $i |
awk '
    { print }
/(^     ;$)|(^})|(^%%)/ { print ".P3" }
'
done
```

13.7.17 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]
```

13.7.18 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;
}
```

13.7.19 mbox

```
From: Polyhedron Software Ltd <100013.461@CompuServe.COM>
To: ">INTERNET:bwk@research.att.com" <bwk@research.att.com>
Subject: Message from Internet
Date: 10 May 91 04:07:07 EDT
Message-Id: <"910510080707 100013.461 CHE27-1"@CompuServe.COM>

Got your message. I'll pass it on to Tony. We haven't noticed any
errors at all in CompuServe mail, so far.

Regards


Graham Wood


From kam Thu May  9 10:58:06 EDT 1991
tony fritzpatrick called from england.  he had spoken to you
last week about compuserve.
the number is:
100013,461
```

```
this is regarding the HOC6 listing.

he will call you back tomorrow

From pipe!sub11276 Fri May  3 10:38:29 EDT 1991
Message to: BK

From:  Tony Fitzpatrick
       ECL
       Highlands Farm
       Greys Road
       Henley OXON, RG 94 PS
       ENGLAND

       Telephone:  0491 - 575-989 (country code 45)

       FAX:  0491 576 557

1.  H would like permission (which has already been granted by publisher) to
    use HUC 6 program -- commercial software.

2.  Is the listing available on floppy disk?

3.  Thank you for a very interesting and useful book.

4.  He left his fax # and telephone #.  He wasn't sure of the country code.
    He would appreciate hearing from you via fax.

        sub 11276
```

13.7.20 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;
}
```

13.8 Всякая всячина

13.8.1 addup1

```
awk '{ s += $'$1' }
 END { print s }'
```

13.8.2 addup2

```
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
} '
```

13.8.3 backup

```
push -v panther $* /usr/bwk/eff/Code
```

13.8.4 backwards

```
# backwards:  print input in backward line order
awk '   { line[NR] = $0 }
END     { for (i = NR; i > 0; i--) print line[i] } ' $*
```

13.8.5 badpick.c

```
pick(s) /* offer choice of s */
    char *s;
{
    fprintf("%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}
```

13.8.6 bundle

```
# bundle:  group files into distribution package

echo '# To unbundle, sh this file'
for i
do
    echo "echo $i 1>&2"
```

```
    echo "cat >$i <<'End of $i'"
    cat $i
    echo "End of $i"
done
```

13.8.7 cal

```
# cal:  nicer interface to /usr/bin/cal

case $# in
0)      set `date`; m=$2; y=$6 ;;  # no args: use today
1)      m=$1; set `date`; y=$6 ;;  # 1 arg: use this year
*)      m=$1; y=$2 ;;              # 2 args: month and year
esac

case $m in
jan*|Jan*)      m=1 ;;
feb*|Feb*)      m=2 ;;
mar*|Mar*)      m=3 ;;
apr*|Apr*)      m=4 ;;
may*|May*)      m=5 ;;
jun*|Jun*)      m=6 ;;
jul*|Jul*)      m=7 ;;
aug*|Aug*)      m=8 ;;
sep*|Sep*)      m=9 ;;
oct*|Oct*)      m=10 ;;
nov*|Nov*)      m=11 ;;
dec*|Dec*)      m=12 ;;
[1-9]|10|11|12) ;;              # numeric month
*)              y=$m; m="" ;;   # plain year
esac

/usr/bin/cal $m $y              # run the real one
```

13.8.8 calendar1

```
# calendar:  version 1 -- today only
awk <$HOME/calendar '
    BEGIN { split("'"`date`"'", date) }
    $1 == date[2] && $2 == date[3]
' | mail $NAME
```

13.8.9 calendar2

```
# calendar:  version 2 -- today only, no quotes
(date; cat $HOME/calendar) |
awk '
  NR == 1    { mon = $2; day = $3 } # set the date
```

```
  NR > 1 && $1 == mon && $2 == day  # print calendar lines
' | mail $NAME
```

13.8.10 calendar3

```
# calendar:  version 3 -- today and tomorrow
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
```

13.8.11 cat0.c

```
/* cat:  minimal version */
#define SIZE    512     /* arbitrary */

main()
{
    char buf[SIZE];
    int n;

    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    exit(0);
}
```

13.8.12 checkmail.c

```
/* checkmail:  watch user's mailbox */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;
```

```
char *maildir = "/usr/spool/mail";      /* system dependent */

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)     /* no mailbox */
            buf.st_size = 0;
        if (buf.st_size > lastsize)
            fprintf(stderr, "\nYou have mail\007\n");
        lastsize = buf.st_size;
        sleep(60);
    }
}
#include "error.c"
```

13.8.13 checkmail.sh

```
# checkmail:  watch mailbox for growth

PATH=/bin:/usr/bin
MAIL=/usr/spool/mail/`getname`  # system dependent

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 "You have mail" }'
```

13.8.14 cp.c

```
/* cp:  minimal version */
#include <stdio.h>
```

```
#define PERMS 0644 /* RW for owner, R for group, others */
char *progname;

main(argc, argv)        /* cp: copy f1 to 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);
}

#include "error.c"
```

13.8.15 doctype

```
# doctype:  synthesize proper command line for 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"
```

```
} '
```

13.8.16 double

```
awk '
FILENAME != prevfile {   # new file
    NR = 1               # reset line number
    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
    if (NF > 0)
        lastword = $NF
}' $*
```

13.8.17 efopen.c

```
FILE *efopen(file, mode)        /* fopen file, die if can't */
    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);
}
```

13.8.18 error.c

```
error(s1, s2)   /* print error message and die */
    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);
}
```

13.8.19 field1

```
awk '{ print $'$1' }'
```

13.8.20 field2

```
awk "{ print \$$1 }"
```

13.8.21 fold

```
# fold:  fold long lines
sed 's/\(->/    /g' $* |        # convert tabs to spaces
awk '
BEGIN {
    N = 80              # folds at column 80
    for (i = 1; i <= N; i++)    # make a string of blanks
        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)
    }
} '
```

13.8.22 frequent

```
cat $* |
tr -sc A-Za-z '\012' |
sort |
uniq -c |
sort -n |
tail |
5
```

13.8.23 frequent2

```
sed 's/[ \(->][ \(->]*/\
/g' $* | sort | uniq -c | sort -nr | sed 10q
```

13.8.24 get

```
# get:  extract file from history

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]$$; exit 1' 1 2 15
# split into current version and editing commands
sed <$INPUT -n '1,/^@@@/w /tmp/get.a'$$'
                /^@@@/,$w /tmp/get.b'$$
# perform the edits
awk </tmp/get.b$$ '
    /^@@@/      { count++ }
    !/^@@@/ && count > 0 && count <= - '$VERSION'
    END { print "$d"; print "w", "'$OUTPUT'" }
' | ed - /tmp/get.a$$
rm -f /tmp/get.[ab]$$
```

13.8.25 get.c

```
get(fd, pos, buf, n) /* read n bytes from position pos */
    int fd, n;
    long pos;
    char *buf;
{
    if (lseek(fd, pos, 0) == -1)        /* get to pos */
        return -1;
    else
        return read(fd, buf, n);
}
```

13.8.26 getname

```
who am i | sed 's/ .*//'
```

13.8.27 idiff.c

```
/* idiff:  interactive diff */

#include <stdio.h>
#include <ctype.h>
char    *progname;
#define HUGE    10000   /* large number of lines */

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);
}

idiff(f1, f2, fin, fout)        /* process diffs */
    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; /* #lines from 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);      /* can fail on very long files */
    unlink(tempfile);
}

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;
}

nskip(fin, n)   /* skip n lines of file fin */
    FILE *fin;
{
    char buf[BUFSIZ];

    while (n-- > 0)
        fgets(buf, sizeof buf, fin);
}

ncopy(fin, n, fout)     /* copy n lines from fin to fout */
    FILE *fin, *fout;
{
    char buf[BUFSIZ];

    while (n-- > 0) {
        if (fgets(buf, sizeof buf, fin) == NULL)
```

```
            return;
        fputs(buf, fout);
    }
}

#include "efopen.c"
```

13.8.28 makefile

```
files:  files.o files1.o directory.o
    cc files.o files1.o directory.o -o files

p0:     p0.c ttyin0.c
    cc p0.c ttyin0.c

clean:
    rm -f *.o a.out
```

13.8.29 newer

```
# newer f:  list files newer than f
ls -t | sed '/^'$1'$/q'
```

13.8.30 news1

```
# news:  print news files, version 1

HOME=.          # debugging only
cd .            # place holder for /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
```

13.8.31 news2

```
# news:  print news files, version 2

HOME=.          # debugging only
cd .            # place holder for /usr/news
IFS='
'               # just a newline
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
```

13.8.32 news3

```
# news:  print news files, final version

PATH=/bin:/usr/bin
IFS='
'                       # just a newline
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
```

13.8.33 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
```

13.8.34 older

```
# older f:  list files older than f
ls -tr | sed '/^'$1'$/q'
```

13.8.35 overwrite1

```
# overwrite:  copy standard input to output after EOF
# version 1.  BUG here

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               # collect the input
cp $new $1              # overwrite the input file
rm -f $new
```

13.8.36 overwrite2

```
# overwrite:  copy standard input to output after EOF
# version 2.  BUG here too

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               # collect the input
cp $1 $old              # save original file

trap '' 1 2 15          # we are committed; ignore signals
cp $new $1              # overwrite the input file

rm -f $new $old
```

13.8.37 overwrite3

```
# overwrite:  copy standard input to output after EOF
# final version

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   # clean up files

if PATH=$opath "$@" >$new               # collect input
then
    cp $file $old       # save original file
    trap '' 1 2 15      # we are committed; ignore signals
    cp $new $file
else
    echo "overwrite: $1 failed, $file unchanged" 1>&2
    exit 1
fi
rm -f $new $old
```

13.8.38 p1.c

```
/* p:  print input in chunks (version 1) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

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);
}
```

```
print(fp, pagesize)     /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;       /* number of lines so far */
    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;
        }
}

#include "ttyin1.c"
#include "efopen.c"
```

13.8.39 p2.c

```
/* p:  print input in chunks (version 2) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp, *efopen();
    int i, pagesize = PAGESIZE;

    progname = argv[0];
    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++) {
            fp = efopen(argv[i], "r");
```

```
            print(fp, pagesize);
            fclose(fp);
        }
    exit(0);
}

print(fp, pagesize)     /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;       /* number of lines so far */
    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;
        }
}

#include "ttyin2.c"
#include "efopen.c"
```

13.8.40 p3.c

```
/* p:  print input in chunks (version 3) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp, *efopen();
    int i, pagesize = PAGESIZE;
    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++;
    }
    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);
}

print(fp, pagesize)     /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;       /* number of lines so far */
    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;
        }
}

#include "ttyin2.c"
#include "efopen.c"
```

13.8.41 p4.c

```
/* p:  print input in chunks (version 4) */

#include <stdio.h>
#define PAGESIZE        22
char    *progname;      /* program name for error message */

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:    /* no match possible */
                fp = efopen(argv[i], "r");
                break;
            case 1:             /* corrected */
                fprintf(stderr, "\"%s\"? ", buf);
                if (ttyin() == 'n')
                        break;
                argv[i] = buf;
                /* fall through... */
            case 0:     /* exact match */
                fp = efopen(argv[i], "r");
                print(fp, pagesize);
                fclose(fp);
            }
    exit(0);
}

print(fp, pagesize)     /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;       /* number of lines so far */
    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;
        }
}

#include "ttyin2.c"
#include "efopen.c"
#include "spname.c"
```

13.8.42 pick1

```
# pick:  select arguments

PATH=/bin:/usr/bin

for i                           # for each argument
do
    echo -n "$i? " >/dev/tty
    read response
    case $response in
    y*) echo $i ;;
    q*) break
    esac
done </dev/tty
```

13.8.43 pick.c

```
/* pick:  offer choice on each argument */

#include <stdio.h>
char    *progname;      /* program name for error message */

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';  /* drop newline */
            pick(buf);
        }
    else
        for (i = 1; i < argc; i++)
```

```
            pick(argv[i]);
    exit(0);
}

pick(s) /* offer choice of s */
    char *s;
{
    fprintf(stderr, "%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}

#include "ttyin2.c"
#include "efopen.c"
```

13.8.44 prpages

```
# prpages:  compute number of pages that pr will print
wc $* |
awk '!/ total$/ { n += int(($1+55) / 56) }
     END        { print n }'
```

13.8.45 put

```
# put:  install file into history

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$$ $1               # previous version
then                    # merge pieces
    cp $1 /tmp/put.b$$          # current version
    echo "@@@ `getname` `date` $Summary" >>/tmp/put.b$$
    diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$      # latest diffs
    sed -n '/^@@@/,$p' <$HIST >>/tmp/put.b$$ # old diffs
    overwrite $HIST     cat /tmp/put.b$$        # put it back
```

```
else                    # make a new one
    echo "put: creating $HIST"
    cp $1 $HIST
    echo "@@@ `getname` `date` $Summary" >>$HIST
fi
rm -f /tmp/put.[ab]$$
```

13.8.46 readslow.c

```
/* readslow:  keep reading, waiting for more */
#define SIZE    512     /* arbitrary */

main()
{
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

13.8.47 replace

```
# replace:  replace str1 in files with str2, in place

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
```

13.8.48 signaltest.c

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
extern int errno;
```

```
main() {
    int c, n;
    char buf[100];
    int onintr();

    signal(SIGINT, onintr);
    for (;;) {
        n = read(0, buf, 100);
        if (n > 0)
            printf(buf);
        else {
            if (errno == EINTR) {
                errno = 0;
                printf("interrupt side %d\n", n);
            } else {
                printf("true end of file %d\n", n);
            }
        }
    }
}

onintr() {
    signal(SIGINT, onintr);
    printf("interrupt\n");
}
```

13.8.49 spname.c

```
/* spname:  return correctly spelled filename */
/*
 * 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 == '/')     /* skip slashes */
```

```
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0')       /* exact or corrected */
            return strcmp(oldname,newname) != 0;
        p = guess;      /* copy next component into guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZ)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
            return -1;  /* hopeless */
        for (p = best; *new = *p++; ) /* add to end */
            new++;                                          /* of newname */
    }
}

mindist(dir, guess, best)       /* search dir for guess */
    char *dir, *guess, *best;
{
    /* set best, return distance 0..3 */
    int d, nd, fd;
    struct {
        ino_t ino;
        char  name[DIRSIZ+1];   /* 1 more than in dir.h */
    } nbuf;

    nbuf.name[DIRSIZ] = '\0';   /* +1 for terminal '\0' */
    if (dir[0] == '\0')         /* current directory */
        dir = ".";
    d = 3;      /* minimum distance */
    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)             /* exact match */
                        break;
            }
        }
    close(fd);
    return d;
}

/* spdist:  return distance between two names */
/*
```

```
 *      very rough spelling metric:
 *      0 if the strings are identical
 *      1 if two chars are transposed
 *      2 if one char wrong, added or deleted
 *      3 otherwise
 */

#define EQ(s,t) (strcmp(s,t) == 0)

spdist(s, t)
    char *s, *t;
{
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;           /* exact match */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1]
              && *t == s[1] && EQ(s+2, t+2))
                return 1;       /* transposition */
            if (EQ(s+1, t+1))
                return 2;       /* 1 char mismatch */
        }
        if (EQ(s+1, t))
            return 2;           /* extra character */
    }
    if (*t && EQ(s, t+1))
        return 2;                       /* missing character */
    return 3;
}
```

13.8.50 strindex.c

```
strindex(s, t)  /* return index of t in s, -1 if none */
    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;
}
```

13.8.51 sv.c

```
/* sv:  save new files */
```

```
#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);
}

sv(file, dir)   /* save file in 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() in some systems */
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1)       /* target not present */
        sto.st_mtime = 0;       /* so make it look old */
    if (sti.st_mtime < sto.st_mtime)    /* target is newer */
        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);
}

#include "error.c"
```

13.8.52 system1.c

```
#include <signal.h>

system(s)       /* run command line 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;
}
```

13.8.53 system.c

```
/*
 * Safer version of system for interactive programs
 */
#include <signal.h>
#include <stdio.h>

system(s)       /* run command line 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;
}
```

13.8.54 timeout.c

```
/* timeout:  set time limit on a process */
#include <stdio.h>
#include <signal.h>
int     pid;            /* child process id */
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()       /* kill child when alarm arrives */
{
    kill(pid, SIGKILL);
}

#include "error.c"
```

13.8.55 toolong

```
length($0) > 72 { print "Line", NR, "too long:", substr($0,1,60) }
```

13.8.56 ttyin1.c

```
ttyin() /* process response from /dev/tty (version 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];
}
```

13.8.57 ttyin2.c

```
ttyin() /* process response from /dev/tty (version 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);      /* BUG here */
            printf("!\n");
        }
        else    /* ordinary line */
            return buf[0];
    }
}

#include "system.c"
```

13.8.58 vis1.c

```
/* vis:  make funny characters visible (version 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);
}
```

13.8.59 vis2.c

```
/* vis:  make funny characters visible (version 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);
}
```

13.8.60 vis3.c

```
/* vis:  make funny characters visible (version 3) */

#include <stdio.h>
#include <ctype.h>
int strip = 0;          /* 1 => discard special characters */

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    FILE *fp;

    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
        case 's':       /* -s: strip funny chars */
            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",
```

```
                        argv[0], argv[i]);
                exit(1);
            } else {
                vis(fp);
                fclose(fp);
            }
    exit(0);
}

vis(fp) /* make chars visible in 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);
}
```

13.8.61 waitfile.c

```
/* waitfile:  wait until file stops changing */
#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) {    /* copy file */
        execlp("cat", "cat", argv[1], (char *) 0);
        error("can't execute cat %s", argv[1]);
    } else {                    /* run process */
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    exit(0);
}

#include "error.c"
```

13.8.62 watchfor

```
# watchfor:  watch for someone to log in

PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: watchfor person' 1>&2; exit 1
esac

until who | egrep "$1"
do
    sleep 60
done
```

13.8.63 watchwho

```
# watchwho:  watch who logs in and out

PATH=/bin:/usr/bin
new=/tmp/wwho1.$$
old=/tmp/wwho2.$$
>$old           # create an empty file

while :         # loop forever
do
    who >$new
    diff $old $new
    mv $new $old
    sleep 60
done | awk '/>/ { $1 = "in:     "; print }
        /</ { $1 = "out:        "; print }'
```

13.8.64 which1

```
# which cmd:  which cmd in PATH is executed, version 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    # use test -x if you can
    then
        echo $i/$1
        exit 0          # found it
    fi
done
exit 1          # not found
```

13.8.65 which1.H

```
# which cmd:  which cmd in PATH is executed, version 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    # use test -x if you can
    then
        echo $i/$1
        exit 0          # found it
    fi
done
exit 1          # not found
@@@  Fri Oct 14 14:21:11 EDT 1983 original version
```

13.8.66 which2

```
# which cmd:  which cmd in PATH is executed, final version

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    # this is /bin/test
    then                        # or /usr/bin/test only
        echo $i/$1
        exit 0          # found it
    fi
done
exit 1          # not found
```

13.8.67 wordfreq

```
awk '   { for (i = 1; i <= NF; i++) num[$i]++ }
END     { for (word in num) print word, num[word] }
' $*
```

13.8.68 zap1

```
# zap pattern:  kill all processes matching pattern
# BUG in this version

PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: zap pattern' 1>&2; exit 1
esac

kill `pick \`ps -ag | grep "$*"\` | awk '{print $1}'`
```

13.8.69 zap2

```
# zap pat:  kill all processes matching pat
# final version

PATH=/bin:/usr/bin
IFS='
'                       # just a newline
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}'`
```

13.8.70 zap.c

```
/* zap:  interactive process killer */

#include <stdio.h>
#include <signal.h>
char    *progname;      /* program name for error message */
char    *ps = "ps -ag"; /* system dependent */

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);        /* get header line */
    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);
}

#include "ttyin2.c"
#include "strindex.c"
#include "efopen.c"
```

Список иллюстраций

Список таблиц

Оглавление