

# Глава 0. Краткое введение в сценарии командной оболочки

Bash (как и сценарии на языке командной оболочки в целом) существует уже очень давно, и каждый день новые люди знакомятся с ее возможностями и приемами автоматизации операций с ее применением. И сейчас, когда компания Microsoft выпустила интерактивную оболочку bash и подсистему команд Unix в Windows 10, самое время узнать, насколько простыми и эффективными могут быть сценарии командной оболочки.

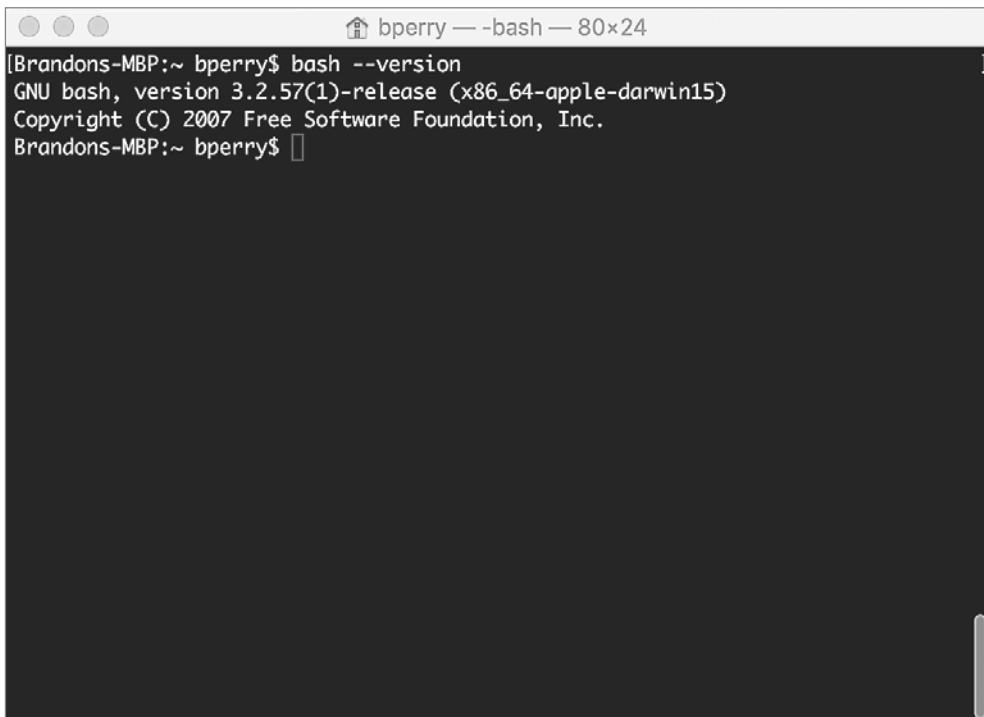
## Что такое командная оболочка?

С первых дней существования компьютеров сценарии командной оболочки помогали системным администраторам и программистам выполнять рутинную работу, на которую иначе пришлось бы потратить массу времени. Так что же такое «сценарии командной оболочки» и почему они должны волновать вас? Сценарии — это текстовые файлы с набором команд, следующих в порядке их выполнения, на языке конкретной командной оболочки (в нашем случае bash). *Командная оболочка (shell)* — это интерфейс командной строки к библиотеке команд в операционной системе.

Сценарии командной оболочки по своей сути являются крохотными программами, написанными с использованием команд операционной системы для автоматизации специальных задач — часто таких, выполнение которых вручную не доставляет никакого удовольствия, например, для сбора информации из сети, слежения за использованием дискового пространства, загрузки данных о погоде, переименования файлов и многих других. В виде сценария нетрудно даже реализовать простенькие игры! Такие сценарии могут включать несложную логику, например, инструкции `if`, которые вы встречали в других языках, но могут быть еще проще, как вы увидите далее.

Многие разновидности командных оболочек, такие как `tcsh`, `zsh` и даже популярная оболочка `bash`, доступны в операционных системах OS X, BSD и Linux. В этой книге основное внимание уделяется главной опоре Unix — командной

оболочке `bash`. Каждая оболочка имеет свои особенности и возможности, но большинство пользователей Unix в первую очередь обычно знакомятся именно с `bash`. В OS X программа `Terminal` открывает окно с оболочкой `bash` (рис. 0.1). В Linux имеется большое разнообразие программ с командной оболочкой, но чаще всего встречаются консоли командной строки: `gnome-terminal` для GNOME и `konsole` для KDE. Эти приложения можно настраивать на использование разных типов командных оболочек, но все они по умолчанию используют `bash`. Фактически в любой Unix-подобной системе, открыв программу-терминал, вы по умолчанию получите доступ к командной оболочке `bash`.

A screenshot of a Terminal window in OS X. The title bar at the top shows three window control buttons (red, yellow, green) on the left, a home icon, the text 'bperry — -bash — 80x24', and a close button on the right. The terminal content shows the command '[Brandons-MBP:~ bperry\$ bash --version]' followed by the output: 'GNU bash, version 3.2.57(1)-release (x86\_64-apple-darwin15)', 'Copyright (C) 2007 Free Software Foundation, Inc.', and 'Brandons-MBP:~ bperry\$'. A cursor is visible at the end of the last line. A vertical scrollbar is on the right side of the terminal area.

**Рис. 0.1.** Вывод версии `bash` в окне приложения `Terminal` в OS X

Использование терминала для взаимодействия с операционной системой может показаться сложнейшей задачей. Однако со временем намного естественней становится просто открыть терминал, чтобы быстро изменить что-то в системе, чем перебирать мышью пункты меню, пытаясь отыскать параметры для изменения.

**ПРИМЕЧАНИЕ**

---

В августе 2016 года компания Microsoft выпустила версию `bash` для Windows 10 Anniversary. То есть теперь ее могут запускать пользователи Windows. В приложении А приводятся инструкции по установке `bash` для Windows 10, но вообще эта книга предполагает, что вы работаете в Unix-подобной системе, такой как OS X или Linux. Вы можете опробовать предлагаемые сценарии в Windows 10, но мы не даем никаких гарантий и сами не тестировали их таким образом! Тем не менее оболочка `bash` славится своей переносимостью и многие сценарии из этой книги должны работать и в Windows.

---

## Запуск команд

Главная особенность `bash` — возможность запускать команды в системе. Давайте опробуем короткий пример «Hello World». Команда `echo` оболочки `bash` выводит текст на экран, например:

```
$ echo "Hello World"
```

Введите данный текст в командной строке `bash`, и вы увидите, как на экране появятся слова `Hello World`. Эта строка кода запускает команду `echo`, хранящуюся в стандартной библиотеке `bash`. Список каталогов, в которых `bash` будет искать стандартные команды, хранится в переменной окружения с именем `PATH`. Вы можете запустить команду `echo` с переменной `PATH`, чтобы увидеть ее содержимое, как показано в листинге 0.1.

**Листинг 0.1.** Вывод текущего содержимого переменной окружения `PATH`

```
$ echo $PATH
/Users/bperry/.rvm/gems/ruby-2.1.5/bin:/Users/bperry/.rvm/gems/ruby-2.1.5@global/
bin:/Users/bperry/.rvm/rubies/ruby-2.1.5/bin:/usr/local/bin:/usr/bin:/bin:/usr/
sbin:/sbin:/opt/X11/bin:/usr/local/MacGPG2/bin:/Users/bperry/.rvm/bin
```

**ПРИМЕЧАНИЕ**

---

В листингах, где присутствуют вводимые команды и результаты их выполнения, вводимые команды выделены жирным и начинаются с символа `$`, чтобы вы могли отличить их от вывода, полученного в ходе выполнения команды.

---

Каталоги в этом выводе отделяются друг от друга двоеточием. Именно их проверит оболочка `bash`, когда от нее потребуют запустить программу или команду. Если искомая команда хранится в каком-то другом каталоге, `bash` не сможет запустить ее. Обратите также внимание, что `bash` проверит перечисленные

каталоги именно *в том порядке, в каком они перечислены в переменной PATH*. Это важно, если у вас имеется две команды с одинаковыми именами, но хранящиеся в разных каталогах, включенных в PATH. Если обнаружится проблема с поиском некоторой команды, попробуйте выполнить команду `which` с ее именем, как показано в листинге 0.2, чтобы увидеть, в каком каталоге из PATH ее найдет оболочка.

**Листинг 0.2.** Поиск команд в PATH с помощью `which`

```
$ which ruby
/Users/bperry/.rvm/rubies/ruby-2.1.5/bin/ruby
$ which echo
/bin/echo
```

Теперь, вооруженные этой информацией, вы сможете переместить или скопировать файл в один из каталогов, перечисленных командой `echo $PATH`, как, например, в листинге 0.1, и затем команда начнет запускаться. Мы будем использовать `which` на протяжении всей книги для определения полного пути к командам. Это удобный инструмент для отладки содержимого переменной PATH.

## Настройка оболочки входа

На всем протяжении книги нам предстоит писать сценарии, которые потом будем использовать в других сценариях, поэтому для нас важна простота вызова новых сценариев. Вы можете настроить переменную PATH так, чтобы ваши собственные сценарии вызывались автоматически, как любые другие команды, в момент запуска новой командной оболочки. Когда открывается новый сеанс командной оболочки, она первым делом читает сценарий входа в домашнем каталоге (`/Users/<username>` в OS X или `/home/<username>` в Linux) и выполняет любые команды, перечисленные в нем. Сценарий входа называется `.login`, `.profile`, `.bashrc` или `.bash_profile`, в зависимости от системы. Чтобы узнать, какой из этих файлов используется как сценарий входа, добавьте в каждый из них следующую строку, заменив последнее слово соответствующим именем файла:

```
echo this is .profile
```

Затем выполните вход. Вверху окна терминала должна появиться строка, сообщающая имя файла сценария, выполненного при входе. Если вы откроете терминал и увидите `this is .profile`, значит, ваша оболочка загружает файл `.profile`; если вы увидите `this is .bashrc`, значит, загружается файл `.bashrc`; и так далее. Однако описанное поведение зависит от типа командной оболочки.

Вы можете добавить в сценарий входа настройку переменной `PATH`, включив в нее другие каталоги. Здесь же можно подкорректировать любые другие настройки `bash`, такие как внешний вид строки приглашения к вводу, содержимое переменной `PATH` и любые другие параметры. Например, воспользуемся командой `cat`, чтобы заглянуть в измененный сценарий входа `.bashrc`. Команда `cat` принимает аргумент с именем файла и выводит его содержимое в окно консоли, как показано в листинге 0.3.

**Листинг 0.3.** Измененный файл `.bashrc`, включающий в переменную `PATH` каталог `RVM`

```
$ cat ~/.bashrc
export PATH="$PATH:$HOME/.rvm/bin" # Добавить в PATH каталог RVM для работы
```

Команда вывела содержимое файла `.bashrc`, в котором переменной `PATH` присваивается новое значение, позволяющее локальной версии `RVM` (Ruby Version Manager — диспетчер версий Ruby) управлять любыми установленными версиями Ruby. Так как сценарий `.bashrc` настраивает `PATH` каждый раз, когда открывается новый сеанс работы с командной оболочкой, диспетчер `RVM` будет доступен по умолчанию.

Аналогично можно открыть доступ к своей библиотеке сценариев командной оболочке. Для этого в своем домашнем каталоге создайте папку, куда будут помещаться разрабатываемые сценарии. Затем добавьте ее в переменную `PATH` в сценарии входа, чтобы упростить вызов сценариев из нее.

Чтобы выяснить путь к домашнему каталогу, дайте команду `echo $HOME`, которая выведет в окне терминала полный путь. Перейдите в указанный каталог и создайте папку для разрабатываемых сценариев (мы рекомендуем назвать ее `scripts`). Затем добавьте эту папку в свой сценарий входа, для чего откройте файл сценария в текстовом редакторе и добавьте в начало файла следующую строку, заменив `/path/to/scripts/` на путь к папке с вашими сценариями:

```
export PATH="/path/to/scripts/:$PATH"
```

Затем вы сможете запустить любой сценарий из этой папки как обычную команду.

## Запуск сценариев командной оболочки

К настоящему моменту мы уже воспользовались некоторыми командами, такими как `echo`, `which` и `cat`. Но мы использовали их по отдельности, а не вместе, то есть не в составе сценария. Давайте напишем сценарий, который выполнит

их все последовательно, как показано в листинге 0.4. Этот сценарий выведет *Hello World*, затем путь к сценарию *neqn*, который по умолчанию должен быть доступен в оболочке *bash*. Затем использует этот путь для вывода содержимого сценария *neqn* на экран. (На данный момент содержимое *neqn* для нас не важно; мы просто выбрали первый попавшийся сценарий для примера.) Этот пример наглядно демонстрирует использование сценария для выполнения группы команд по порядку, в данном случае, чтобы увидеть полный путь к сценарию и содержимое сценария.

**Листинг 0.4.** Содержимое нашего первого сценария командной оболочки

```
echo "Hello World"
echo $(which neqn)
cat $(which neqn)
```

Откройте текстовый редактор (в Linux, например, большой популярностью пользуются редакторы Vim и gedit, а в OS X — TextEdit) и введите содержимое листинга 0.4. Затем сохраните сценарий с именем *intro* в своем каталоге для разрабатываемых сценариев. Сценарии командной оболочки не требуют специального расширения файлов, так что сохраните файл с именем без расширения (или, если пожелаете, добавьте расширение *.sh*, но в этом нет необходимости). Первая строка в сценарии вызывает команду *echo*, чтобы просто вывести текст *Hello World*. Вторая строка чуть сложнее; она использует команду *which* для поиска файла сценария *neqn* и затем с помощью *echo* выводит найденный путь на экран. Чтобы выполнить такую связку команд, где одна передается другой в виде аргумента, *bash* использует *подоболочку*, в которой выполняет вторую команду и сохраняет ее вывод для передачи первой. В нашем примере подоболочка выполнит команду *which*, которая вернет полный путь к сценарию *neqn*. Затем этот путь будет передан как аргумент команде *echo*, которая просто выведет его на экран. Наконец, тот же трюк с подоболочкой используется для передачи пути к сценарию *neqn* команде *cat*, которая выведет содержимое сценария *neqn*.

Сохраните файл и запустите сценарий в окне терминала. Вы должны увидеть результат, показанный в листинге 0.5.

**Листинг 0.5.** Результат запуска нашего первого сценария командной оболочки

```
$ sh intro
❶ Hello World
❷ /usr/bin/neqn
❸ #!/bin/sh
# Присутствие этого сценария не должно расцениваться как наличие поддержки
# GNU eqn и groff -Tascii|-Tlatin1|-Tutf8|-Tcyrillic
```

```
GROFF_RUNTIME="${GROFF_BIN_PATH=/usr/bin}:"  
PATH="$GROFF_RUNTIME$PATH"  
export PATH  
exec eqn -Tascii ${1+"$@"}  
  
# eof  
$
```

Запуск сценария производится с помощью команды `sh`, которой имя сценария `intro` передается как аргумент. Команда `sh` обойдет все строки в файле и выполнит их, как если бы это были команды `bash`, введенные в окне терминала. Как показано в листинге 0.5, сначала на экран выводится строка `Hello World` ❶, затем путь к файлу `neqn` ❷. В заключение выводится содержимое файла `neqn` ❸; это исходный код короткого сценария командной оболочки `neqn`, хранящегося на вашем жестком диске (в OS X, по крайней мере, в Linux содержимое этого сценария может немного отличаться).

## Упрощение способа вызова сценариев

Для запуска сценариев не обязательно использовать команду `sh`. Если добавить еще одну строку в сценарий `intro` и изменить его разрешения в файловой системе, его можно будет запускать непосредственно, без команды `sh`, как любые другие команды. Откройте сценарий `intro` в текстовом редакторе и измените его, как показано ниже:

```
❶ #!/bin/bash  
echo "Hello World"  
echo $(which neqn)  
cat $(which neqn)
```

Мы добавили единственную строку в самое начало файла, ссылающуюся на путь в файловой системе `/bin/bash` ❶. Эта строка называется *shebang*<sup>1</sup>. С ее помощью командная оболочка определяет, какую программу запустить для интерпретации сценария. Здесь в качестве интерпретатора мы указали `bash`. Вы можете встретить другие строки *shebang*, например, в сценариях на языке Perl (`#!/usr/bin/perl`) или Ruby (`#!/usr/bin/env ruby`).

После добавления строки нам еще необходимо установить права доступа к файлу, разрешающие выполнять его как обычную программу. Для этого в окне терминала выполните команды, показанные в листинге 0.6.

---

<sup>1</sup> Произносится как «ше-банг». — *Примеч. пер.*

**Листинг 0.6.** Изменение прав доступа к файлу сценария `intro`, разрешающих его выполнение

```
❶ $ chmod +x intro
❷ $ ./intro
Hello World
/usr/bin/neqn
#!/bin/sh
# Присутствие этого сценария не должно расцениваться как наличие поддержки
# GNU eqn и groff -Tascii|-Tlatin1|-Tutf8|-Tcp1047

GROFF_RUNTIME="${GROFF_BIN_PATH=/usr/bin}:"
PATH="$GROFF_RUNTIME$PATH"
export PATH
exec eqn -Tascii ${1+"$@"}

# eof
$
```

Для изменения прав доступа мы использовали команду `chmod` ❶ и передали ей аргумент `+x`, который требует от команды дать указанному файлу право на выполнение, и имя самого файла. После настройки права на выполнение для сценария, чтобы запускать его как обычную программу, мы можем вызвать сценарий непосредственно, как показано в строке ❷, без вызова самой оболочки `bash`. Это общепринятая практика в разработке сценариев командной оболочки, и вы со временем поймете ее полезность. Большинству сценариев, которые мы напишем в этой книге, так же потребуется дать право на выполнение, подобно сценарию `intro`.

Мы привели лишь простой пример, чтобы показать, как запускать сценарии командной оболочки и как использовать сценарии для запуска других сценариев. Во многих сценариях в этой книге мы задействуем именно такой метод, и вы еще не раз увидите строки `shebang` в будущем.

## Почему именно сценарии командной оболочки?

Кого-то из вас может беспокоить вопрос: почему для создания сценариев предпочтительнее использовать язык командной оболочки `bash` вместо более новых и мощных языков, таких как `Ruby` и `Go`. Да, эти языки гарантируют переносимость между разными типами систем, но они не устанавливаются по умолчанию. Причина проста: на любой машине с операционной системой `Unix` имеется командная оболочка, и на подавляющем большинстве из них используется оболочка `bash`. Как отмечалось в начале главы, компания `Microsoft` недавно выпустила для `Windows 10` ту же самую командную оболочку `bash`, которая имеется во всех основных дистрибутивах `Linux` и `OS X`. То есть теперь



сценарии командной оболочки стали еще более переносимыми с минимумом усилий с вашей стороны. Кроме того, сценарии на языке командной оболочки позволяют быстрее и проще решать задачи обслуживания и администрирования системы, чем сценарии на других языках. Оболочка `bash` все еще далека от идеала, но в этой книге вы узнаете, как смягчить некоторые ее недостатки.

В листинге 0.7 приводится пример маленького, удобного и полностью переносимого сценария командной оболочки (фактически, это однострочная команда на `bash`!). Сценарий определяет общее количество страниц во всех документах OpenOffice, находящихся в указанной папке, и может пригодиться писателям.

**Листинг 0.7.** Сценарий для определения общего количества страниц во всех документах OpenOffice в указанной папке

```
#!/bin/bash
echo "${exiftool *.odt | grep Page-count | cut -d ":" -f2 | tr '\n' '+'}"0" | bc
```

Не будем обсуждать тонкости работы этого сценария — в конце концов, мы только в самом начале пути. Но в общих чертах отметим, что он извлекает информацию о количестве страниц из каждого документа, выстраивает строку из полученных чисел, перемежая их операторами сложения, и передает ее калькулятору командной строки для вычисления суммы. На все про все оказалось достаточно одной строки кода. В книге вы найдете еще множество таких же потрясающих сценариев, как этот, и после некоторой практики он покажется вам невероятно простым!

## За дело

Теперь вы должны представлять, как создаются сценарии командной оболочки, если прежде вы этим не занимались. Создание коротких сценариев для решения специализированных задач заложено в основу философии Unix. Умение писать собственные сценарии и расширять возможности системы Unix под свои потребности даст вам огромную власть. Эта глава лишь намекнула, что ждет вас впереди: множество по-настоящему потрясающих сценариев командной оболочки!

# Глава 1. Отсутствующая библиотека

Одна из замечательных особенностей Unix — возможность создавать новые команды, объединяя старые новыми способами. Но даже при том, что Unix включает сотни команд и предоставляет тысячи способов их комбинирования, вы все еще можете столкнуться с ситуацией, когда никакая из комбинаций не позволит решить поставленную задачу правильно. В этой главе мы исследуем основные аспекты, знание которых поможет вам создавать более сложные и интеллектуальные программы на языке командной оболочки.

Но есть еще кое-что, о чем необходимо поговорить в самом начале: среда программирования на языке командной оболочки не так сложна, как другие среды программирования на настоящих языках. Perl, Python, Ruby и даже C имеют структуры и библиотеки, предлагающие дополнительные возможности, тогда как сценарии на языке командной оболочки — это в большей степени ваш собственный мир. Сценарии в данной главе помогут вам найти в нем свой путь. Далее они послужат строительными блоками для создания более мощных сценариев.

Наибольшую сложность при разработке сценариев представляют также тонкие различия между разновидностями Unix и дистрибутивами GNU/Linux. Даже при том, что стандарты IEEE POSIX определяют общую функциональную основу для всех реализаций Unix, иногда все же бывает непросто начать пользоваться системой OS X после нескольких лет работы в окружении Red Hat GNU/Linux. Команды различаются, хранятся в разных каталогах и часто имеют тонкие различия в интерпретации флагов. Эти различия могут сделать создание сценариев командной оболочки непростым занятием, но мы познакомим вас с некоторыми хитростями, помогающими справиться с этими сложностями.

## Что такое POSIX?

В первые дни Unix был сродни Дикому Западу: разные компании создавали новые версии операционной системы и развивали их в разных направлениях, одновременно уверяя клиентов, что все эти новые версии — просто разновидности Unix, совместимые между собой. Но в дело вмешался Институт инженеров

электротехники и электроники (Institute for Electrical and Electronic Engineers, IEEE) и, объединив усилия всех основных производителей, разработал стандартное определение Unix под названием «Интерфейс переносимой операционной системы» (Portable Operating System Interface, или POSIX), которому должны были соответствовать все коммерческие и открытые реализации Unix. Нельзя купить операционную систему POSIX как таковую, но все доступные версии Unix и GNU/Linux в общих чертах соответствуют требованиям POSIX (хотя некоторые ставят под сомнение необходимость стандарта POSIX, когда GNU/Linux сам стал стандартом де-факто).

Однако иногда даже POSIX-совместимые реализации Unix отличаются друг от друга. В качестве примера можно привести команду `echo`, о которой рассказывается далее в этой главе. Отдельные версии этой команды поддерживают флаг `-n`, который запрещает добавлять символ перевода строки по умолчанию. Другие версии `echo` поддерживают экранированную последовательность `\с`, которая интерпретируется как «не включать перевод строки», а третьи вообще не дают возможности запретить добавление этого символа в конце вывода. Более того, отдельные системы Unix имеют командные оболочки, где команда `echo` реализована как встроенная функция, которая игнорирует флаги `-n` и `\с`, а также включают стандартную реализацию команды в виде двоичного файла `/bin/echo`, обрабатывающую эти флаги. В результате возникают сложности со сценариями запросов на ввод данных, потому что сценарии должны работать одинаково в как можно большем количестве версий Unix. Следовательно, для нормальной работы сценариев важно нормализовать поведение команды `echo`, чтобы оно было единообразным в разных системах. Далее в этой главе, в сценарии № 8, вы увидите, как заключить команду `echo` в сценарий командной оболочки, чтобы получить такую нормализованную версию.

---

**ПРИМЕЧАНИЕ**

Некоторые сценарии в этой книге используют дополнительные возможности `bash`, поддерживаемые не всеми POSIX-совместимыми командными оболочками.

---

Но хватит теории — приступим к знакомству со сценариями, которые будут включены в нашу библиотеку!

## № 1. Поиск программ в PATH

Сценарии, использующие переменные окружения (такие как `MAILER` или `PAGER`), таят в себе скрытую опасность: некоторые их настройки могут ссылаться на несуществующие программы. Для тех, кто не сталкивался прежде с этими

переменными окружения, отметим, что MAILER должна хранить путь к программе электронной почты (например, /usr/bin/mailx), а PAGER должна ссылаться на программу страничного просмотра длинных документов. Например, если вы решите увеличить гибкость сценария и вместо системной программы страничного просмотра по умолчанию (обычно more или less) использовать для отображения вывода сценария переменную PAGER, необходимо убедиться, что эта переменная содержит действительный путь к существующей программе.

Этот первый сценарий показывает, как проверить доступность указанной программы в списке путей PATH. Он также послужит отличной демонстрацией нескольких приемов программирования на языке командной оболочки, включая определение функций и переменных. Листинг 1.1 показывает, как проверить допустимость путей к файлам.

## Код

**Листинг 1.1.** Сценарий inpath с определениями функций

```
#!/bin/bash
# inpath -- Проверяет допустимость пути к указанной программе
# или ее доступность в каталогах из списка PATH

in_path()
{
    # Получает команду и путь, пытается отыскать команду. Возвращает 0, если
    # команда найдена и является выполняемым файлом; 1 – если нет. Обратите
    # внимание, что эта функция временно изменяет переменную окружения
    # IFS (Internal Field Separator – внутренний разделитель полей), но
    # восстанавливает ее перед завершением.

    cmd=$1          ourpath=$2      result=1
    oldIFS=$IFS     IFS=":"

    for directory in "$ourpath"
    do
        if [ -x $directory/$cmd ] ; then
            result=0          # Если мы здесь, значит, команда найдена.
        fi
    done

    IFS=$oldIFS
    return $result
}

checkForCmdInPath()
{
    var=$1
```

```
if [ "$var" != "" ] ; then
❶   if [ "${var:0:1}" = "/" ] ; then
❷     if [ ! -x $var ] ; then
       return 1
     fi
❸   elif ! in_path $var "$PATH" ; then
       return 2
     fi
  fi
fi
}
```

В главе 0 мы рекомендовали создать в своем домашнем каталоге новую папку *scripts* и добавить полный путь к ней в свою переменную окружения *PATN*. Выполните команду `echo $PATN`, чтобы увидеть текущее значение переменной *PATN*, и добавьте в сценарий входа (*.login*, *.profile*, *.bashrc* или *.bash\_profile*, в зависимости от оболочки) строку, изменяющую значение *PATN*. Подробности ищите в разделе «Настройка оболочки входа» в главе 0.

#### ПРИМЕЧАНИЕ

Если попробовать вывести список файлов в каталоге с помощью команды `ls`, некоторые специальные файлы, такие как *.bashrc* и *.bash\_profile*, могут не отображаться. Это объясняется тем, что файлы, имена которых начинаются с точки, например *.bashrc*, считаются «скрытыми». (Как оказывается, эта «ошибка, превратившаяся в «фишку» была допущена еще в самом начале развития Unix.) Чтобы вывести все файлы, включая скрытые, добавьте в команду `ls` флаг `-a`.

Напомним еще раз: все наши сценарии написаны в предположении, что они будут выполняться командной оболочкой `bash`. Обратите внимание: этот сценарий явно указывает в первой строке (называется *shebang*), что для его интерпретации должен использоваться интерпретатор `/bin/bash`. Многие системы поддерживают также строку `shebang /usr/bin/env bash`, которая определяет местонахождение интерпретатора в момент запуска сценария.

#### ЗАМЕЧАНИЕ О КОММЕНТАРИЯХ

Мы долго думали, включать ли в код подробное описание работы сценария, и решили, что в некоторых случаях будем приводить пояснения к особенно заковыристым фрагментам после самого кода, но в общем случае для пояснения происходящего будем использовать комментарии в коде. Ищите строки, начинающиеся с символа `#`, или текст в строках кода, которому предшествует символ `#`.

Поскольку вам придется читать сценарии других людей (не только наши!), будет полезно попрактиковаться понимать происходящее в сценариях по комментариям в них. Кроме того, писать комментарии — хорошая привычка, которую желательно выработать у себя при работе над собственными сценариями, потому что это поможет вам понять, чего вы стремитесь достичь в разных блоках кода.

## Как это работает

Функция `checkForCmdInPath` отличает значение параметра с одним только именем программы (например, `echo`) от значения, содержащего полный путь, плюс имя файла (например, `/bin/echo`). Для этого она сравнивает первый символ в переданном ей значении с символом `/`; для чего ей требуется изолировать первый символ от остального значения параметра.

Обратите внимание на синтаксис `${var:0:1}` ❶ — это сокращенная форма извлечения подстроки: указывается начальная позиция в исходной строке и длина извлекаемой подстроки (если длина не указана, возвращается остаток строки до конца). Выражение `${var:10}`, например, вернет остаток строки в `$var` начиная с десятого символа, а `${var:10:6}` вернет только символы, заключенные между позициями 10 и 15 включительно. Что это означает, демонстрирует следующий пример:

```
$ var="something wicked this way comes..."
$ echo ${var:10}
wicked this way comes...
$ echo ${var:10:6}
wicked
$
```

В листинге 1.1 данный синтаксис используется, чтобы определить, начинается ли указанный путь с символа слеша. Если это так, то далее функция проверяет наличие указанного файла в файловой системе по указанному пути. Пути, начинающиеся с символа `/`, являются абсолютными, и для их проверки можно использовать оператор `-x` ❷. В противном случае значение параметра передается в функцию `inpath` ❸, чтобы проверить наличие указанного файла в одном из каталогов, перечисленных в `PATH`.

## Запуск сценария

Чтобы запустить сценарий как самостоятельную программу, нужно добавить в самый конец файла короткий блок команд. Эти команды просто принимают ввод пользователя и передают его в функцию, как показано ниже.

```
if [ $# -ne 1 ] ; then
    echo "Usage: $0 command" >&2
    exit 1
fi

checkForCmdInPath "$1"
case $? in
    0 ) echo "$1 found in PATH" ;;
    1 ) echo "$1 not found or not executable" ;;
    2 ) echo "$1 not found in PATH" ;;
esac

exit 0
```

После добавления кода сценарий можно запустить непосредственно, как показано далее, в разделе «Результаты». Закончив эксперименты со сценарием, не забудьте удалить или закомментировать дополнительный код, чтобы потом его можно было подключать как библиотеку функций.

## Результаты

Для проверки вызовем сценарий `inpath` с именами трех программ: существующей программы, также существующей программы, но находящейся в каталоге, не включенном в список `PATН`, и несуществующей программы, но с полным путем к ней. Пример тестирования сценария приводится в листинге 1.2.

### Листинг 1.2. Тестирование сценария `inpath`

```
$ inpath echo
echo found in PATH
$ inpath MrEcho
MrEcho not found in PATH
$ inpath /usr/bin/MrEcho
/usr/bin/MrEcho not found or not executable
```

Последний блок кода, добавленный позднее, преобразует результат вызова функции `in_path` в нечто более читаемое, поэтому теперь мы легко можем видеть, что все три случая обрабатываются, как ожидалось.

## Усовершенствование сценария

Для желающих начать овладевать мастерством программирования с первого сценария, покажем, как заменить выражение `${var:0:1}` его более сложной формой: `${var%${var#?}}`. Такой метод извлечения подстрок определяет стандарт POSIX. Эта галиматья в действительности включает два выражения

извлечения подстроки. Внутреннее выражение `${var#?}` извлекает из `var` все, кроме первого символа, где `#` удаляет первое совпадение с заданным шаблоном, а `?` — это регулярное выражение, которому соответствует точно один символ.

Внешнее выражение `${var%pattern}` возвращает подстроку из строки слева, оставшуюся после удаления указанного шаблона `pattern` из `var`. В данном случае удаляемый шаблон `pattern` — это результат внутреннего выражения, то есть внешнее выражение вернет первый символ в строке.

Для тех, кому POSIX-совместимый синтаксис кажется пугающим, отметим, что большинство командных оболочек (включая `bash`, `ksh` и `zsh`) поддерживает другой метод извлечения подстрок, `${varname:start:size}`, который был использован в сценарии.

Те, кому не нравится ни один из представленных способов извлечения первого символа, могут использовать системные команды: `$(echo $var | cut -c1)`. В программировании на `bash` практически любую задачу, будь то извлечение, преобразование или загрузка данных из системы, можно решить несколькими способами. При этом важно понимать, что наличие нескольких способов не означает, что один способ лучше другого.

Кроме того, чтобы сценарий различал, запускается он как самостоятельная программа или подключается другим сценарием, можно добавить в начало условный оператор:

```
if [ "$BASH_SOURCE" = "$0" ]
```

Это сработает и с любым другим сценарием. Однако мы предлагаем вам, дорогой читатель, дописать остальной код после экспериментов!

---

#### ПРИМЕЧАНИЕ

Сценарий № 47 в главе 6 тесно связан с этим сценарием. Он проверяет каталоги в `PATH` и переменные в окружении пользователя.

---

## № 2. Проверка ввода: только алфавитно-цифровые символы

Пользователи постоянно игнорируют указания и вводят недопустимые данные, в неправильном формате или неправильным синтаксисом. Как разработчик сценариев командной оболочки вы должны обнаружить и отметить такие ошибки еще до того, как они превратятся в проблемы.



Часто подобные ситуации связаны с вводом имен файлов или ключей в базе данных. Программа просит пользователя ввести строку, которая должна содержать только *алфавитно-цифровые символы*, то есть только буквы верхнего или нижнего регистра и цифры — никаких знаков пунктуации, специальных символов и пробелов. Правильную ли строку ввел пользователь? Ответ на этот вопрос дает сценарий в листинге 1.3.

## Код

### Листинг 1.3. Сценарий validalnum

```
#!/bin/bash
# validAlphaNum - проверяет, содержит ли строка только
# алфавитные и цифровые символы
validAlphaNum()
{
    # Проверка аргумента: возвращает 0, если все символы в строке являются
    # буквами верхнего/нижнего регистра или цифрами; иначе возвращает 1

    # Удалить все недопустимые символы.
❶ validchars="$(echo $1 | sed -e 's/^[[:alnum:]]//g')"
❷ if [ "$validchars" = "$1" ] ; then
    return 0
else
    return 1
fi
}

# НАЧАЛО ОСНОВНОГО СЦЕНАРИЯ -- УДАЛИТЕ ИЛИ ЗАКОММЕНТИРУЙТЕ ВСЕ, ЧТО НИЖЕ,
# ЧТОБЫ ЭТОТ СЦЕНАРИЙ МОЖНО БЫЛО ПОДКЛЮЧАТЬ К ДРУГИМ СЦЕНАРИЯМ.
# =====
/bin/echo -n "Enter input: "
read input

# Проверка ввода
if ! validAlphaNum "$input" ; then
    echo "Please enter only letters and numbers." >&2
    exit 1
else
    echo "Input is valid."
fi

exit 0
```

## Как это работает

Логика работы сценария проста: сначала с помощью редактора `sed` создается новая версия введенных данных, из которой удалены все недопустимые символы ❶. Затем новая версия сравнивается с оригиналом ❷. Если две версии оказались одинаковыми, все в порядке. В противном случае, если в результате

обработки редактором `sed` потерялись данные, значит, исходная версия содержит недопустимые символы.

В основе работы сценария лежит операция подстановки редактора `sed`, которая удаляет любые символы, не входящие в множество `[:alnum:]`, где `[:alnum:]` — это сокращение POSIX для регулярного выражения, соответствующего всем алфавитно-цифровым символам. Если результат операции подстановки не совпадает с исходным вводом, значит, в исходной строке присутствуют другие символы, кроме алфавитно-цифровых, недопустимые в данном случае. Функция возвращает ненулевое значение, чтобы сообщить о проблеме. Имейте в виду: в этом примере предполагается, что введенные данные являются текстом ASCII.

## Запуск сценария

Сценарий содержит все необходимое для его запуска как самостоятельной программы. Он предлагает ввести строку и затем сообщает о ее допустимости. Однако чаще эта функция используется для копирования в начало другого сценария в виде ссылки, как показано в сценарии № 12.

Сценарий `validalnum` также представляет собой хороший пример программирования на языке командной оболочки вообще: сначала пишутся функции, а затем они тестируются перед включением в другие, более сложные сценарии. Такой подход позволяет избавиться от многих неприятностей.

## Результаты

Сценарий `validalnum` прост в применении, он предлагает пользователю ввести строку для проверки. В листинге 1.4 показано, как сценарий реагирует на допустимый и недопустимый ввод.

### Листинг 1.4. Тестирование сценария `validalnum`

```
$ validalnum
Enter input: valid123SAMPLE
Input is valid.
$ validalnum
Enter input: this is most assuredly NOT valid, 12345
Please enter only letters and numbers.
```

## Усовершенствование сценария

Метод «удалить недопустимые символы и посмотреть, что осталось» хорошо подходит для проверки благодаря своей гибкости. При этом важно помнить, что обе переменные — исходная строка и шаблон — должны заключаться

в двойные кавычки, чтобы избежать ошибок в случае ввода пустой строки (или пустого шаблона). Пустые значения переменных — извечная проблема в программировании сценариев, потому что при проверке в условном операторе они вызывают сообщение об ошибке. Всегда помните, что пустая строка в кавычках отличается от пустого значения переменной.

Хотите потребовать, чтобы ввод содержал только буквы верхнего регистра, пробелы, запятые и точки? Просто измените шаблон подстановки в строке ❶, как показано ниже:

```
sed 's/[^[[:upper:]] ,.]/g'
```

Эту же функцию можно использовать для простейшей проверки телефонных номеров (допускается присутствие цифр, пробелов, круглых скобок и дефисов, но не допускается наличие пробелов в начале или нескольких пробелов, идущих подряд), если использовать шаблон:

```
sed 's/[^- [:digit:]]\\(\\)/g'
```

Но, если нужно ограничить ввод целыми числами, опасайтесь ловушки. Например, на первый взгляд кажется, что следующий шаблон справится с этой задачей:

```
sed 's/[^[[:digit:]]]/g'
```

Однако он будет пропускать только положительные целые числа. А что, если вам необходимо разрешить ввод отрицательных чисел? Если вы просто добавите знак «минус» в множество допустимых символов, функция признает допустимой строку -3-4, хотя совершенно очевидно, что она не является допустимым целым числом. Обработка отрицательных чисел демонстрируется в сценарии № 5.

### № 3. Нормализация форматов дат

Разработчикам сценариев часто приходится иметь дело с большим количеством разнообразных форматов представления дат, нормализация которых может быть сопряжена с разными сложностями. Самые серьезные проблемы связаны с датами, потому что они записываются самыми разными способами. Даже если потребовать ввести дату в определенном формате, например месяц-день-год, вы почти наверняка получите несовместимый ввод: номер месяца вместо названия, сокращенное название вместо полного или даже полное название со всеми буквами в верхнем регистре. По этой причине функция нормализации дат, даже самая простенькая, послужит очень хорошим строительным блоком для многих сценариев, особенно таких, как сценарий № 7.

## Код

Сценарий в листинге 1.5 нормализует строки с датами, используя относительно простой набор критериев: месяц должен задаваться именем или числом в диапазоне от 1 до 12, а год — четырехзначным числом. Нормализованная строка с датой включает название месяца (в виде трехсимвольного сокращения), за которым следуют день месяца и четырехзначный год.

### Листинг 1.5. Сценарий normdate

```
#!/bin/bash
# normdate -- Нормализует поле месяца в строке с датой в трехсимвольное
# представление, с первой буквой в верхнем регистре.
# Вспомогательная функция для сценария № 7, valid-date.
# В случае успеха возвращает 0.

monthNumToName()
{
    # Присвоить переменной 'month' соответствующее значение.
    case $1 in
        1 ) month="Jan" ;; 2 ) month="Feb" ;;
        3 ) month="Mar" ;; 4 ) month="Apr" ;;
        5 ) month="May" ;; 6 ) month="Jun" ;;
        7 ) month="Jul" ;; 8 ) month="Aug" ;;
        9 ) month="Sep" ;; 10) month="Oct" ;;
        11) month="Nov" ;; 12) month="Dec" ;;
        * ) echo "$0: Unknown month value $1" >&2
            exit 1
    esac
    return 0
}

# НАЧАЛО ОСНОВНОГО СЦЕНАРИЯ -- УДАЛИТЕ ИЛИ ЗАКОММЕНТИРУЙТЕ ВСЕ, ЧТО НИЖЕ,
# ЧТОБЫ ЭТОТ СЦЕНАРИЙ МОЖНО БЫЛО ПОДКЛЮЧАТЬ К ДРУГИМ СЦЕНАРИЯМ.
# =====
# Проверка ввода
if [ $# -ne 3 ] ; then
    echo "Usage: $0 month day year" >&2
    echo "Formats are August 3 1962 and 8 3 1962" >&2
    exit 1
fi
if [ $3 -le 99 ] ; then
    echo "$0: expected 4-digit year value." >&2
    exit 1
fi

# Месяц введен как число?
❶ if [ -z $(echo $1|sed 's/[[:digit:]]//g') ] ; then
    monthNumToName $1
else
```

```
# Нормализовать до 3 первых букв, первая в верхнем регистре, остальные в нижнем.
❷ month="$(echo $1|cut -c1|tr '[:lower:]' '[:upper:]')"
❸ month="$month$(echo $1|cut -c2-3 | tr '[:upper:]' '[:lower:]')"
fi

echo $month $2 $3

exit 0
```

## Как это работает

Обратите внимание на третий условный оператор в этом сценарии **❸**. Он выбрасывает из поля с месяцем все цифры и затем с помощью оператора `-z` проверяет, получилась ли в результате пустая строка. Если получилась, это означает, что в поле содержатся только цифры, соответственно, его можно напрямую преобразовать в название месяца вызовом функции `monthNumToName`, которая дополнительно проверяет номер месяца на попадание в диапазон от 1 до 12. Иначе предполагается, что первое поле во введенной строке содержит название месяца, которое нормализуется сложной последовательностью команд `cut` и `tr` с использованием двух подболочек (то есть последовательности команд заключены в скобки `$( и )`), которые вызывают заключенные в них команды и возвращают их вывод).

Первая последовательность команд в подболочке, в строке **❷**, извлекает первый символ из поля с названием месяца и с помощью `tr` преобразует его в верхний регистр (последовательность `echo $1|cut -c1` можно также записать в стиле POSIX: `${1%${1#?}}`, как было показано выше). Вторая последовательность, в строке **❸**, извлекает второй и третий символы и преобразует их в нижний регистр. В результате получается трехсимвольное сокращенное название месяца с первым символом в верхнем регистре. Обратите внимание, что в данном случае не проверяется — содержит ли исходное поле допустимое название месяца, в отличие от случая, когда месяц задается числом.

## Запуск сценария

Для максимальной гибкости будущих сценариев, использующих `normdate`, этот сценарий спроектирован так, что принимает исходные данные в виде трех аргументов командной строки, как показано в листинге 1.6. Если вы предполагаете использовать сценарий только интерактивно, предложите пользователю ввести дату в виде трех значений, однако это усложнит вызов `normdate` из других сценариев.

## Результаты

### Листинг 1.6. Тестирование сценария normdate

```
$ normdate 8 3 62
normdate: expected 4-digit year value.
$ normdate 8 3 1962
Aug 3 1962
$ normdate AUGUST 03 1962
Aug 03 1962
```

Обратите внимание, что этот сценарий нормализует только представление месяца; представление дня (в том числе с ведущими нулями) и года не изменяется.

## Усовершенствование сценария

Прежде чем знакомиться с разными усовершенствованиями, которые можно добавить в этот сценарий, загляните в раздел с описанием сценария № 7, где используется `normdate` для проверки вводимых дат.

Одно из изменений, которые можно внедрить уже сейчас, касается включения поддержки дат в форматах `MM/DD/YYYY` и `MM-DD-YYYY`, для чего достаточно добавить следующий код непосредственно перед первым условным оператором:

```
if [ $# -eq 1 ] ; then # Чтобы компенсировать форматы с / и -
    set -- $(echo $1 | sed 's/[\\\/-]/ /g')
fi
```

С этим изменением сценарий позволяет вводить и нормализовать даты в следующих распространенных форматах:

```
$ normdate 6-10-2000
Jun 10 2000
$ normdate March-11-1911
Mar 11 1911
$ normdate 8/3/1962
Aug 3 1962
```

Если вы прочитаете код очень внимательно, то заметите, что в нем можно также усовершенствовать проверку поля с номером года, не говоря уже о поддержке разных международных форматов представления дат. Мы оставляем это вам как упражнение для самостоятельных исследований!

## № 4. Удобочитаемое представление больших чисел

Программисты часто допускают типичную ошибку, отображая результаты вычислений без предварительного форматирования. Пользователям сложно определить, например, сколько миллионов содержится в числе 43245435, не подсчитав количество цифр справа налево и не добавив мысленно запятые после каждого третьего знака. Сценарий в листинге 1.7 выводит большие числа в удобочитаемом формате.

### Код

**Листинг 1.7.** Сценарий `nicenumber` форматирует большие числа, делая их удобочитаемыми

```
#!/bin/bash
# nicenumber -- Отображает переданное число в формате представления с запятыми.
# Предполагает наличие переменных DD (decimal point delimiter -- разделитель
# дробной части) и TD (thousands delimiter -- разделитель групп разрядов).
# Создает переменную nicenum с результатом, а при наличии второго аргумента
# дополнительно выводит результат в стандартный вывод.
nicenumber()
{
    # Обратите внимание: предполагается, что для разделения дробной и целой
    # части во входном значении используется точка.
    # В выходной строке в качестве такого разделителя используется точка, если
    # пользователь не определил другой символ с помощью флага -d.
    ❶ integer=$(echo $1 | cut -d. -f1) # Слева от точки
    ❷ decimal=$(echo $1 | cut -d. -f2) # Справа от точки

    # Проверить присутствие дробной части в числе.
    if [ "$decimal" != "$1" ]; then
        # Дробная часть есть, включить ее в результат.
        result="{DD:= '.$'}$decimal"
    fi

    thousands=$integer

    ❸ while [ $thousands -gt 999 ]; do
        ❹ remainder=$((thousands % 1000)) # Три последние значимые цифры

        # В 'remainder' должно быть три цифры. Требуется добавить ведущие нули?
        while [ ${#remainder} -lt 3 ]; do # Добавить ведущие нули
            remainder="0$remainder"
        done

        ❺ result="{TD:=','}${remainder}${result}" # Конструировать справа налево
```

```

❸ thousands=$((thousands / 1000)) # Оставить остаток, если есть
done

nicenum="${thousands}${result}"
if [ ! -z $2 ] ; then
    echo $nicenum
fi
}

DD="." # Десятичная точка для разделения целой и дробной части
TD="," # Разделитель групп разрядов

# Начало основного сценария
# =====

❷ while getopts "d:t:" opt; do
    case $opt in
        d ) DD="$OPTARG" ;;
        t ) TD="$OPTARG" ;;
        esac
    done
    shift $((OPTIND - 1))

    # Проверка ввода
    if [ $# -eq 0 ] ; then
        echo "Usage: $(basename $0) [-d c] [-t c] number"
        echo " -d specifies the decimal point delimiter"
        echo " -t specifies the thousands delimiter"
        exit 0
    fi
done

❹ nicenumber $1 1 # Второй аргумент заставляет nicenumber вывести результат.

exit 0

```

## Как это работает

Основная работа в этом сценарии выполняется циклом `while` внутри функции `nicenumber()` ❸, который последовательно удаляет три младших значащих разряда из числового значения в переменной `thousands` ❹ и присоединяет их к создаваемой форматированной версии числа ❺. Затем цикл уменьшает числовое значение в `thousands` ❻ и повторяет итерацию, если необходимо. Вслед за функцией `nicenumber()` начинается основная логика сценария. Сначала с помощью `getopts` ❷, анализируются параметры, переданные в сценарий, и затем вызывается функция `nicenumber()` ❸ с последним аргументом, указанным пользователем.



## Запуск сценария

Чтобы опробовать этот сценарий, просто вызовите его с очень большим числом. Сценарий добавит десятичную точку и разделители групп разрядов, используя значения либо по умолчанию, либо указанные с помощью флагов.

Результат можно внедрить в сообщение, как показано ниже:

```
echo "Do you really want to pay \${$(nicenumber $price)}?"
```

## Результаты

Сценарий `nicenumber` может также принимать дополнительные параметры. Листинг 1.8 демонстрирует форматирование нескольких чисел с использованием сценария.

### Листинг 1.8: Тестирование сценария `nicenumber`

```
$ nicenumber 5894625
5,894,625
$ nicenumber 589462532.433
589,462,532.433
$ nicenumber -d, -t. 589462532.433
589.462.532,433
```

## Усовершенствование сценария

В разных странах используют разные символы в качестве десятичной точки и для разделения групп разрядов, поэтому в сценарии предусмотрена возможность передачи дополнительных флагов. Например, в Германии и Италии сценарию следует передать `-d "."` и `-t ","`, во Франции `-d ","` и `-t " "`, а в Швейцарии, где четыре государственных языка, следует использовать `-d "."` и `-t "'"`. Это отличный пример ситуации, когда гибкость оказывается ценнее жестко определенных значений, потому что инструмент становится полезным для более широкого круга пользователей.

С другой стороны, мы жестко установили, что во входных значениях роль десятичной точки будет играть символ `"."`, то есть, если вы предполагаете использование другого разделителя дробной и целой части во входных значениях, измените символ в двух вызовах команды `cut` в строках ❶ и ❷, где сейчас используется `"."`.

Ниже показано одно из решений:

```
integer=$(echo $1 | cut -d$DD -f1) # Слева от точки
decimal=$(echo $1 | cut -d$DD -f2) # Справа от точки
```

Это решение работоспособно, только если разделитель дробной и целой части во входном значении не отличается от разделителя, выбранного для результата, в противном случае сценарий просто не будет работать. Более сложное решение состоит в том, чтобы непосредственно перед этими двумя строками включить проверку, позволяющую убедиться, что разделитель дробной и целой части во входном значении совпадает с разделителем, указанным пользователем. Для реализации проверки можно использовать тот же трюк, что был показан в сценарии № 2: отбросить все цифры и посмотреть, что осталось, например:

```
separator="$(echo $1 | sed 's/[[[:digit:]]//g')"  
if [ ! -z "$separator" -a "$separator" != "$DD" ] ; then  
    echo "$0: Unknown decimal separator $separator encountered." >&2  
    exit 1  
fi
```

## № 5. Проверка ввода: целые числа

Как было показано в сценарии № 2, проверка целых чисел осуществляется очень просто, пока дело не доходит до отрицательных значений. Проблема в том, что всякое отрицательное число может содержать только один знак «минус», который обязан быть первым. Процедура проверки в листинге 1.9 оценивает правильность форматирования отрицательных чисел и, что особенно ценно, может проверить вхождение значений в установленный пользователем диапазон.

### Код

#### Листинг 1.9. Сценарий validint

```
#!/bin/bash  
# validint -- Проверяет целые числа, поддерживает отрицательные значения  
  
validint()  
{  
    # Проверяет первое значение и сравнивает с минимальным значением $2 и/или  
    # с максимальным значением $3, если они заданы. Если проверяемое значение  
    # вне заданного диапазона или не является допустимым целым числом,  
    # возвращается признак ошибки.  
  
    number="$1"; min="$2"; max="$3"  
  
    ❶ if [ -z $number ] ; then  
        echo "You didn't enter anything. Please enter a number." >&2  
        return 1  
    fi
```

```

# Первый символ -- знак "минус"?
❷ if [ "${number%${number#?}}" = "-" ] ; then
    testvalue="${number#?}" # Оставить для проверки все, кроме первого символа
else
    testvalue="$number"
fi

# Удалить все цифры из числа для проверки.
❸ nodigits="$(echo $testvalue | sed 's/[[[:digit:]]//g')"

# Проверить наличие нецифровых символов.
if [ ! -z $nodigits ] ; then
    echo "Invalid number format! Only digits, no commas, spaces, etc." >&2
    return 1
fi

❹ if [ ! -z $min ] ; then
    # Входное значение меньше минимального?
    if [ "$number" -lt "$min" ] ; then
        echo "Your value is too small: smallest acceptable value is $min." >&2
        return 1
    fi
fi

if [ ! -z $max ] ; then
    # Входное значение больше максимального?
    if [ "$number" -gt "$max" ] ; then
        echo "Your value is too big: largest acceptable value is $max." >&2
        return 1
    fi
fi

return 0
}

```

## Как это работает

Проверка целочисленных значений реализуется очень просто благодаря тому что такие значения состоят исключительно из последовательности цифр (от 0 до 9), перед которой может находиться единственный знак «минус». Если в вызов функции `validint()` передать минимальное и (или) максимальное значение, она также проверит вхождение заданного значения в указанный диапазон.

Сначала функция проверяет ввод непустого значения ❶ (еще один пример, когда важно использовать двойные кавычки, чтобы предотвратить появление сообщения об ошибке в случае ввода пустой строки). Затем, в строке ❷, она проверяет наличие знака «минус» и в строке ❸ удаляет из введенного

значения все цифры. Если в результате получилась непустая строка, значит, введено значение, не являющееся целым числом, и функция возвращает признак ошибки.

Если введенное значение допустимо, оно сравнивается с минимальным и максимальным значениями ④. Наконец, в случае ошибки функция возвращает 1 и 0 — в случае успеха.

## Запуск сценария

Весь сценарий целиком является функцией. Его можно скопировать в другой сценарий или подключить как библиотечный файл. Чтобы преобразовать его в команду, просто добавьте в конец файла код из листинга 1.10.

**Листинг 1.10.** Дополнительная поддержка, превращающая сценарий в самостоятельную команду

```
# Проверка ввода
if validint "$1" "$2" "$3" ; then
    echo "Input is a valid integer within your constraints."
fi
```

## Результаты

После добавления кода из листинга 1.10, сценарий можно использовать, как показано в листинге 1.11:

**Листинг 1.11.** Тестирование сценария validint

```
$ validint 1234.3
Invalid number format! Only digits, no commas, spaces, etc.
$ validint 103 1 100
Your value is too big: largest acceptable value is 100.
$ validint -17 0 25
Your value is too small: smallest acceptable value is 0.
$ validint -17 -20 25
Input is a valid integer within your constraints.
```

## Усовершенствование сценария

Обратите внимание на строку ②, которая проверяет, не является ли первый символ знаком «минус»:

```
if [ "${number%${number#?}}" = "-" ] ; then
```

Если первый символ действительно является знаком «минус», переменной `testvalue` присваивается числовая часть значения. Затем из этого неотрицательного значения удаляются все цифры и выполняется следующая проверка.

В данном случае велик соблазн использовать логический оператор И (`-a`), чтобы объединить выражения и избавиться от вложенных инструкций `if`. Например, на первый взгляд кажется, что следующий код должен работать:

```
if [ ! -z $min -a "$number" -lt "$min" ] ; then
    echo "Your value is too small: smallest acceptable value is $min." >&2
    exit 1
fi
```

Но он не работает, потому что, даже если первое выражение, слева от оператора И, вернет ложное значение, нет никаких гарантий, что вторая проверка не будет выполнена (хотя в большинстве других языков программирования получилось бы именно так). То есть вы рискуете столкнуться со множеством ошибок из-за сравнения недействительных или неожиданных значений. Так быть не должно, но таковы реалии программирования на языке командной оболочки.

## № 6. Проверка ввода: вещественные числа

Проверка вещественных значений (с плавающей точкой) при ограниченных возможностях командной оболочки на первый взгляд кажется сложнейшей задачей, но представьте, что вещественное число состоит из двух целых чисел, разделенных десятичной точкой. Добавьте сюда возможность сослаться на другой сценарий (`validint`), и вы удивитесь, насколько короткой бывает проверка вещественных значений. Сценарий в листинге 1.12 предполагает, что находится в одном каталоге со сценарием `validint`.

### Код

#### Листинг 1.12. Сценарий `validfloat`

```
#!/bin/bash
# validfloat - Проверяет допустимость вещественного значения.
# Имейте в виду, что сценарий не распознает научную форму записи (1.304e5).

# Чтобы проверить вещественное значение, его нужно разбить на две части:
# целую и дробную. Первая часть проверяется как обычное целое число,
# а дробная – как положительное целое число. То есть число -30.5 оценивается
# как допустимое, а -30.-8 нет.

# Подключение других сценариев к текущему осуществляется с помощью оператора "."
# Довольно просто.

. validint
```

```

validfloat()
{
    fvalue="$1"

    # Проверить наличие десятичной точки.
    ❶ if [ ! -z $(echo $fvalue | sed 's/[^\./]/g') ] ; then

        # Извлечь целую часть числа, слева от десятичной точки.
        ❷ decimalPart=$(echo $fvalue | cut -d. -f1)

        # Извлечь дробную часть числа, справа от десятичной точки.
        ❸ fractionalPart="${fvalue#*\.}"

        # Проверить целую часть числа, слева от десятичной точки
        ❹ if [ ! -z $decimalPart ] ; then
            # "!" инвертирует логику проверки, то есть ниже проверяется
            # "если НЕ допустимое целое число"
            if ! validint "$decimalPart" "" "" ; then
                return 1
            fi
        fi

        # Теперь проверим дробную часть.

        # Прежде всего, она не может содержать знак "минус" после десятичной точки,
        # например: 33.-11, поэтому проверим знак '-' в дробной части.
        ❺ if [ "${fractionalPart}${fractionalPart#?}" = "-" ] ; then
            echo "Invalid floating-point number: '-' not allowed \
                after decimal point." >&2
            return 1
        fi
        if [ "$fractionalPart" != "" ] ; then
            # Если дробная часть НЕ является допустимым целым числом...
            if ! validint "$fractionalPart" "0" "" ; then
                return 1
            fi
        fi
    else
        # Если все значение состоит из единственного знака "-",
        # это недопустимое значение.
        ❻ if [ "$fvalue" = "-" ] ; then
            echo "Invalid floating-point format." >&2
            return 1
        fi

        # В заключение проверить, что оставшиеся цифры представляют
        # допустимое целое число.
        if ! validint "$fvalue" "" "" ; then
            return 1
        fi
    fi
    return 0
}

```

## Как это работает

Сценарий сначала проверяет наличие десятичной точки во входном значении ❶. Если точки в числе нет, это не вещественное число. Далее для анализа извлекаются целая ❷ и дробная ❸ части числа. Затем, в строке ❹, сценарий проверяет, является ли целая часть (*слева* от десятичной точки) допустимым целым числом. Следующая последовательность проверок сложнее, потому что требуется проверить ❺ отсутствие дополнительного знака «минус» (чтобы исключить такие странные числа, как 17. -30) и убедиться, что дробная часть (*справа* от десятичной точки) является допустимым целым числом.

Последняя проверка в строке ❻ выясняет, не является ли проверяемое значение единственным знаком «минус» (такое число выглядело бы слишком странно, чтобы пропустить его).

Все проверки выполнены успешно? Тогда сценарий возвращает 0, указывающий, что ввод пользователя содержит допустимое вещественное число.

## Запуск сценария

Если во время выполнения функции не будет выведено сообщения об ошибке, она вернет 0 для числа, являющегося допустимым вещественным значением. Чтобы протестировать сценарий, добавьте в конец следующие строки кода:

```
if validfloat $1 ; then
    echo "$1 is a valid floating-point value."
fi

exit 0
```

Если попытка подключить сценарий `validint` сгенерирует ошибку, убедитесь, что он находится в одном из каталогов, перечисленных в `PATH`, или просто скопируйте функцию `validint` непосредственно в начало сценария `validfloat`.

## Результаты

Сценарий `validfloat` принимает единственный аргумент для проверки. Листинг 1.13 демонстрирует проверку нескольких значений с помощью `validfloat`.

### Листинг 1.13. Тестирование сценария `validfloat`

```
$ validfloat 1234.56
1234.56 is a valid floating-point value.
$ validfloat -1234.56
```

```
-1234.56 is a valid floating-point value.  
$ validfloat -.75  
-.75 is a valid floating-point value.  
$ validfloat -11.-12  
Invalid floating-point number: '-' not allowed after decimal point.  
$ validfloat 1.0344e22  
Invalid number format! Only digits, no commas, spaces, etc.
```

Если вы увидите лишний вывод, это может объясняться присутствием строк, добавленных ранее в `validint` для тестирования, которые вы забыли удалить перед переходом к этому сценарию. Просто вернитесь назад, к описанию сценария № 5 и прокомментируйте или удалите строки, добавленные для тестирования функции.

## Усовершенствование сценария

Было бы круто добавить в функцию поддержку научной формы записи, продемонстрированной в последнем примере. Это не так уж трудно. Вам нужно проверить присутствие в числе символа 'e' или 'E' и затем разбить его на три сегмента: целую часть (всегда представлена единственной цифрой), дробную часть и степень числа 10. После этого каждую часть можно проверить с помощью `validint`.

## № 7. Проверка форматов дат

Одна из наиболее сложных, но очень важная команда проверки — это проверка допустимости дат. Если не принимать в расчет високосные годы, задача не кажется особенно трудной, потому что каждый год календарь остается неизменным. В данном случае достаточно иметь таблицу с числом дней в месяцах и использовать ее для проверки каждой конкретной даты. Чтобы учесть високосные годы, нужно добавить в сценарий дополнительную логику, и именно этот аспект вызывает наибольшие сложности.

Ниже приводится набор критериев, проверка которых позволяет сказать, является ли проверяемый год високосным:

- Если год не кратен 4, он *не високосный*.
- Если год делится на 4 и на 400 — это *високосный* год.
- Если год делится на 4 и не делится на 400, но делится на 100 — это *не високосный* год.
- Все остальные годы, кратные 4, являются *високосными*.



Просматривая исходный код в листинге 1.14, обратите внимание, что для нормализации исходной даты перед проверкой этот сценарий использует `normdate`.

## Код

### Листинг 1.14. Сценарий `valid-date`

```
#!/bin/bash
# valid-date – Проверяет дату с учетом правил определения високосных лет

normdate="укажите здесь имя файла, в котором вы сохранили сценарий normdate.sh"

exceedsDaysInMonth()
{
    # С учетом названия месяца и числа дней в этом месяце, данная функция
    # вернет: 0, если указанное число меньше или равно числу дней в месяце;
    # 1 -- в противном случае.

❶ case $(echo $1|tr '[:upper:]' '[:lower:]') in
    jan* ) days=31 ;; feb* ) days=28 ;;
    mar* ) days=31 ;; apr* ) days=30 ;;
    may* ) days=31 ;; jun* ) days=30 ;;
    jul* ) days=31 ;; aug* ) days=31 ;;
    sep* ) days=30 ;; oct* ) days=31 ;;
    nov* ) days=30 ;; dec* ) days=31 ;;
    * ) echo "$0: Unknown month name $1" >&2
        exit 1
    esac
    if [ $2 -lt 1 -o $2 -gt $days ] ; then
        return 1
    else
        return 0 # Число месяца допустимо.
    fi
}

isLeapYear()
{
    # Эта функция возвращает 0, если указанный год является високосным;
    # иначе возвращается 1.
    # Правила проверки високосного года:
    # 1. Если год не делится на 4, значит, он не високосный.
    # 2. Если год делится на 4 и на 400, значит, он високосный.
    # 3. Если год делится на 4, не делится на 400 и делится
    #    на 100, значит, он не високосный.
    # 4. Любой другой год, который делится на 4, является високосным.

    year=$1
❷ if [ "$((year % 4))" -ne 0 ] ; then
        return 1 # Nope, not a leap year.
    elif [ "$((year % 400))" -eq 0 ] ; then
        return 0 # Yes, it's a leap year.
    elif [ "$((year % 100))" -eq 0 ] ; then
```

```

        return 1
    else
        return 0
    fi
}

# Начало основного сценария
# =====

if [ $# -ne 3 ] ; then
    echo "Usage: $0 month day year" >&2
    echo "Typical input formats are August 3 1962 and 8 3 1962" >&2
    exit 1
fi

# Нормализовать дату и сохранить для проверки на ошибки.

❸ newdate="$(($normdate "$@")"
if [ $? -eq 1 ] ; then
    exit 1 # Error condition already reported by normdate
fi

# Разбить нормализованную дату, в которой
# первое слово = месяц, второе слово = число месяца
# третье слово = год.
month="$(echo $newdate | cut -d\ -f1)"
day="$(echo $newdate | cut -d\ -f2)"
year="$(echo $newdate | cut -d\ -f3)"

# После нормализации данных проверить допустимость
# числа месяца (например, Jan 36 является недопустимой датой).
if ! exceedsDaysInMonth $month "$2" ; then
    if [ "$month" = "Feb" -a "$2" -eq "29" ] ; then
        if ! isLeapYear $3 ; then
            ❹ echo "$0: $3 is not a leap year, so Feb doesn't have 29 days." >&2
                exit 1
            fi
        else
            echo "$0: bad day value: $month doesn't have $2 days." >&2
            exit 1
        fi
    fi

    echo "Valid date: $newdate"

    exit 0

```

## Как это работает

Этот сценарий было очень интересно писать, потому что он требует проверки большого количества непростых условий: числа месяца, високосного года

и так далее. Логика сценария не просто проверяет месяц как число от 1 до 12 или день — от 1 до 31. Чтобы сценарий проще было писать и читать, в нем используются специализированные функции.

Первая функция, `exceedsDaysInMonth()`, анализирует месяц, указанный пользователем, разрешая вероятные допущения (например, пользователь может передать название `JANUAR`, и оно будет правильно опознано). Анализ выполняется инструкцией `case` в строке ❶, которая преобразует свой аргумент в нижний регистр и затем сравнивает полученное значение с константами, чтобы получить число дней в месяце. Единственный недостаток — для февраля функция всегда возвращает 28 дней.

Вторая функция, `isLeapYear()`, с помощью простых арифметических проверок выясняет, содержит ли февраль в указанном году 29-е число ❷.

В основном сценарии исходные данные передаются сценарию `normdate`, представленному выше, для нормализации ❸ и затем разбиваются на три поля: `$month`, `$day` и `$year`. Затем вызывается функция `exceedsDaysInMonth` для проверки допустимости указанного числа для данного месяца, при этом 29 февраля обрабатывается отдельно — в этом случае вызовом функции `isLeapYear` проверяется год ❹ и при необходимости выводится сообщение об ошибке. Если пользовательская дата успешно преодолела все проверки, значит, она допустима!

## Запуск сценария

Запуская сценарий (как показано в листинге 1.15), введите в командной строке дату в формате месяц-день-год. Месяц можно указать в виде трехсимвольного сокращения, полного названия или числа; год должен состоять из четырех цифр.

## Результаты

**Листинг 1.15.** Тестирование сценария `valid-date`

```
$ valid-date august 3 1960
Valid date: Aug 3 1960
$ valid-date 9 31 2001
valid-date: bad day value: Sep doesn't have 31 days.
$ valid-date feb 29 2004
Valid date: Feb 29 2004
$ valid-date feb 29 2014
valid-date: 2014 is not a leap year, so Feb doesn't have 29 days.
```

## Усовершенствование сценария

Подход, аналогичный используемому в этом сценарии, можно применить для проверки значения времени в 24-часовом формате или в 12-часовом формате с суффиксом AM/PM (Ante Meridiem/Post Meridiem — пополудни/пополудни). Разбив значение времени по двоеточиям, нужно убедиться, что число минут и секунд (если указано) находится в диапазоне от 0 до 59, и затем проверить первое поле на вхождение в диапазон от 0 до 12, если присутствует суффикс AM/PM, или от 0 до 24, если предполагается 24-часовой формат. К счастью, несмотря на существование секунд координации (високосных секунд) и других небольших корректировок, помогающих сохранить сбалансированность календарного времени, их можно игнорировать в повседневной работе, то есть нет необходимости использовать замысловатые вычисления.

При наличии доступа к GNU-команде `date` в Unix или GNU/Linux можно использовать совершенно иной способ проверки високосных лет. Попробуйте выполнить следующую команду и посмотрите, что получится:

```
$ date -d 12/31/1996 +%j
```

Если у вас в системе используется новейшая, улучшенная версия `date`, вы получите результат **366**. Более старая версия просто пожалуется на ошибочный формат входных данных. Теперь подумайте о результате, возвращаемом новейшей командой `date`. Сможете ли вы написать двухстрочную функцию, проверяющую високосный год?

Наконец, данный сценарий слишком терпимо относится к названиям месяцев, например, название `febmar` будет опознано как допустимое, потому что инструкция `case` в строке ❶ проверяет только первые три буквы. Эту проблему можно устранить, организовав точную проверку общепринятых сокращений (таких как `feb`) и полных названий месяцев (`february`), и даже некоторых типичных опечаток (`febuary`). Все это легко реализуется, было бы желание!

## № 8. Улучшение некачественных реализаций echo

Как упоминалось в разделе «Что такое POSIX?» в начале этой главы, большинство современных реализаций Unix и GNU/Linux включают команду `echo`, поддерживающую флаг `-n`, который подавляет вывод символа перевода строки в конце, но такая поддержка имеется не во всех реализациях. Некоторые для подавления поведения по умолчанию используют специальный символ `\c`, другие просто добавляют символ перевода строки, не давая никакой возможности изменить это поведение.

Выяснить, какая реализация `echo` используется в текущей системе, довольно просто: введите следующие команды и посмотрите, что из этого получится:

```
$ echo -n "The rain in Spain"; echo " falls mainly on the Plain"
```

Если команда `echo` поддерживает флаг `-n`, вы увидите следующий вывод:

```
The rain in Spain falls mainly on the Plain
```

Если нет, вывод будет иметь следующий вид:

```
-n The rain in Spain
falls mainly on the Plain
```

Гарантировать определенный формат вывода очень важно, и эта важность будет расти с увеличением интерактивности сценариев. Так что мы напишем альтернативную версию `echo`, с именем `echon`, которая всегда будет подавлять вывод завершающего символа перевода строки. Благодаря этому мы получим достаточно надежный инструмент, который сможем использовать, когда понадобится функциональность `echo -n`.

## Код

Способов исправить проблему с командой `echo` так же много, как страниц в этой книге. Но больше всего нам нравится очень компактная реализация, которая просто фильтрует ввод с помощью команды `awk printf`, как показано в листинге 1.16.

**Листинг 1.16.** Простая альтернатива `echo`, использующая команду `awk printf`

```
echon()
{
    echo "$*" | awk '{ printf "%s", $0 }'
}
```

Однако есть возможность избежать накладных расходов на вызов команды `awk`. Если у вас в системе имеется команда `printf`, используйте ее в сценарии `echon`, как показано в листинге 1.17.

**Листинг 1.17.** Альтернатива `echo`, использующая команду `printf`

```
echon()
{
    printf "%s" "$*"
}
```

А как быть, если команды `printf` нет и вы не желаете использовать `awk`? Тогда отсекайте любые завершающие символы перевода строки с помощью команды `tr`, как показано в листинге 1.18.

**Листинг 1.18.** Простая альтернатива `echo`, использующая команду `tr`

```
echon()  
{  
    echo "$*" | tr -d '\n'  
}
```

Это простой и эффективный способ с хорошей переносимостью.

## Запуск сценария

Просто добавьте этот сценарий в каталог из списка `PATH`, и вы сможете заменить все вызовы `echo -n` командой `echon`, надежно помещающей текстовый курсор в конец строки после вывода.

## Результаты

Для демонстрации функции `echon` сценарий принимает аргумент и выводит его, затем читает ввод пользователя. В листинге 1.19 показан сеанс тестирования сценария.

**Листинг 1.19.** Тестирование команды `echon`

```
$ echon "Enter coordinates for satellite acquisition: "  
Enter coordinates for satellite acquisition: 12,34
```

## Усовершенствование сценария

Скажем честно: тот факт, что одни командные оболочки имеют команду `echo`, поддерживающую флаг `-n`, другие предполагают использование специального символа `\c` в конце вывода, а третьи вообще не дают возможности подавить отображение символа перевода строки, доставляет массу проблем создателям сценариев. Чтобы устранить это несоответствие, можно написать свою функцию, которая автоматически проверит поведение `echo`, определит, какая версия используется в системе и затем изменит вызов соответственно. Например, можно выполнить команду `echo -n hi | wc -c` и проверить количество символов в результате: два (`hi`), три (`hi` плюс символ перевода строки), четыре (`-n hi`) или пять (`-n hi` плюс символ перевода строки).

## № 9. Вычисления произвольной точности с вещественными числами

В сценариях часто используется синтаксическая конструкция `$(( ))`, позволяющая выполнять вычисления с использованием простейших математических функций. Эта конструкция может очень пригодиться для упрощения таких распространенных операций, как увеличение на единицу переменных-счетчиков. Она поддерживает операции сложения, вычитания, деления, деления по модулю (остаток от деления нацело) и умножения, но только с целыми числами. Другими словами, следующая команда вернет 0, а не 0,5:

```
echo $(( 1 / 2 ))
```

То есть вычисления с большей точностью превращаются в проблему. Существует не так много хороших программ-калькуляторов, работающих в командной строке. Одна из них — замечательная программа `bc`, которой владеют очень немногие пользователи Unix. Позиционирующая себя как калькулятор для вычислений с произвольной точностью, `bc` появилась на заре развития Unix, славится малопонятными сообщениями об ошибках и отсутствием подсказок. Предполагается, что пользователь и так знает, что делает. Но в этом есть свои плюсы. Мы можем написать сценарий-обертку, делающий программу `bc` более дружелюбной, как показано в листинге 1.20.

### Код

#### Листинг 1.20. Сценарий `scriptbc`

```
#!/bin/bash

# scriptbc -- обертка для 'bc', возвращающая результат вычислений
❶ if ["$1" = "-p" ] ; then
    precision=$2
    shift 2
    else
❷    precision=2 # По умолчанию
fi

❸ bc -q -l << EOF
    scale=$precision
    $*
    quit
EOF

exit 0
```

## Как это работает

Синтаксис `<<` в строке ③ позволяет включить в сценарий произвольное содержимое и интерпретировать его как текст, введенный непосредственно в поток ввода, что в данном случае дает простой способ передачи команд программе `bc`. Такие вставки называют *встроенными документами* (*here document*). Вслед за парой символов `<<` помещается текстовая метка, которая будет интерпретироваться как признак конца такого потока ввода (при условии, что она находится в отдельной строке). В листинге 1.20 используется метка `EOF`.

Этот сценарий демонстрирует также, как использовать аргументы для увеличения гибкости команд. В данном случае сценарий можно вызвать с флагом `-p` ① и указать желаемую точность чисел для вывода. Если точность не указана, по умолчанию используется точность `scale=2` ②.

Работая с программой `bc`, важно понимать разницу между ее параметрами `length` (длина) и `scale` (точность). В терминологии `bc` под длиной (`length`) понимается общее количество цифр в числе, а под точностью (`scale`) — количество цифр после десятичной точки. То есть число 10,25 имеет длину 4 и точность 2, а число 3,14159 имеет длину 6 и точность 5.

По умолчанию `bc` имеет переменное значение для `length`, но, так как параметр `scale` по умолчанию получает нулевое значение, без параметров программа `bc` действует подобно синтаксической конструкции `$(( ))`. К счастью, если в вызов `bc` добавить параметр `scale`, она продемонстрирует огромную скрытую мощь, как показано в следующем примере, где вычисляется количество недель между 1962 и 2002 годами (исключая високосные дни):

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
scale=10
(2002-1962)*365
14600
14600/7
2085.7142857142
quit
```

Чтобы получить доступ к возможностям `bc` из командной строки, сценарий-обертка должен удалить начальную информацию об авторских правах, если она имеется, однако большинство реализаций `bc` автоматически подавляют



вывод начального баннера, если вводом является не терминал (`stdin`). Кроме того, сценарий-обертка определяет довольно разумное значение для масштаба (`scale`), передает программе `bc` фактическое выражение и затем завершает ее командой `quit`.

## Запуск сценария

Чтобы запустить сценарий, передайте математическое выражение программе в виде аргумента, как показано в листинге 1.21.

## Результаты

**Листинг 1.21.** Тестирование сценария `scriptbc`

```
$ scriptbc 14600/7
2085.71
$ scriptbc -p 10 14600/7
2085.7142857142
```

## № 10. Блокировка файлов

Любому сценарию, читающему или записывающему данные в общий файл, например в файл журнала, необходим надежный способ блокировки файлов, чтобы другие экземпляры сценария не могли по ошибке затереть данные в файле до того, как он перестанет использоваться. Для этого часто создается отдельный *файл-блокировка* для каждого используемого файла. Наличие файла-блокировки играет роль *семафора*, или индикатора, сообщающего, что файл задействован другим сценарием и не должен использоваться. Запрашивающий сценарий в этом случае многократно проверяет наличие файла-блокировки, ожидая его удаления, после которого файл можно свободно использовать.

Однако применение файлов-блокировок сопряжено с большими трудностями, потому что многие решения, кажущиеся надежными, в действительности очень ненадежны. Например, для организации блокировки доступа к файлам часто используется следующее решение:

```
while [ -f $lockfile ] ; do
    sleep 1
done
touch $lockfile
```

Кажется, что такое решение должно работать. Или нет? Сценарий в цикле проверяет присутствие файла-блокировки и, как только он исчезает, тут же

создает собственный, чтобы в безопасности изменить рабочий файл. Если в это время другой сценарий увидит файл-блокировку, то продолжит выполнять цикл ожидания, пока тот не исчезнет. Однако на практике такой способ не работает. Представьте, что сразу после выхода из цикла `while`, но перед вызовом команды `touch` диспетчер задач приостановит сценарий, дав возможность поработать другому сценарию.

Если вам непонятно о чем речь, вспомните, что хотя кажется, что компьютер делает что-то одно, в действительности он выполняет сразу несколько программ, переключаясь между ними через короткие интервалы времени. Проблема в том, что между завершением цикла, проверяющего существование файла-блокировки, и созданием нового проходит время, в течение которого система может переключиться с одного сценария на другой, а тот в свою очередь благополучно убедится в отсутствии файла-блокировки и создаст свою версию. Затем система переключится на первый сценарий, который тут же выполнит команду `touch`. В результате оба сценария будут считать, что имеют исключительный доступ к файлу-блокировке, то есть сложится ситуация, которой мы пытаемся избежать.

К счастью, Стефан ван ден Берг (Stephen van den Berg) и Филип Гюнтер (Philip Guenther), авторы программы `procmail` для фильтрации электронной почты, также создали утилиту командной строки `lockfile`, которая дает возможность безопасной и надежной работы с файлами-блокировками в сценариях командной оболочки.

Многие реализации Unix, включая GNU/Linux и OS X, устанавливают утилиту `lockfile` по умолчанию. Ее присутствие в системе можно проверить простой командой `man 1 lockfile`. Если в результате откроется страница справочного руководства, значит, удача сопутствует вам! Сценарий в листинге 1.22 предполагает наличие команды `lockfile`, и все последующие сценарии требуют работоспособности механизма надежной блокировки, реализованного в сценарии № 10, поэтому перед их использованием также проверьте наличие команды `lockfile` в вашей системе.

## Код

### Листинг 1.22. Сценарий `filelock`

```
#!/bin/bash
# filelock -- Гибкий механизм блокировки файлов

retries="10"          # Число попыток по умолчанию
action="lock"         # Действие по умолчанию
nullcmd="'which true'" # Пустая команда для lockfile
```

```

❶ while getopts "lur:" opt; do
    case $opt in
        l ) action="lock" ;;
        u ) action="unlock" ;;
        r ) retries="$OPTARG" ;;
    esac
done
❷ shift $(( $OPTIND - 1 ))

if [ $# -eq 0 ] ; then # Вывести в stdout многострочное сообщение об ошибке.
    cat << EOF >&2
        Usage: $0 [-l|-u] [-r retries] LOCKFILE
        Where -l requests a lock (the default), -u requests an unlock, -r X
        specifies a max number of retries before it fails (default = $retries).
    EOF1
    exit 1
fi

# Проверка наличия команды lockfile.

❸ if [ -z "$(which lockfile | grep -v '^no ')" ] ; then
    echo "$0 failed: 'lockfile' utility not found in PATH." >&2
    exit 1
fi
❹ if [ "$action" = "lock" ] ; then
    if ! lockfile -l -r $retries "$1" 2> /dev/null; then
        echo "$0: Failed: Couldn't create lockfile in time." >&2
        exit 1
    fi
else # Действие = разблокировка
    if [ ! -f "$1" ] ; then
        echo "$0: Warning: lockfile $1 doesn't exist to unlock." >&2
        exit 1
    fi
    rm -f "$1"
fi

exit 0

```

## Как это работает

Как это часто бывает с хорошо написанными сценариями командной оболочки, половину листинга 1.22 занимает анализ входных данных и проверка на наличие ошибок. Затем выполняется инструкция `if` и осуществляется фактическая

---

<sup>1</sup> Символы «EOF» должны находиться в начале строки, т. е. перед ними не должно быть пробелов. Это требование синтаксиса встроенных документов. В оригинале это правило нарушено. В данном листинге перед всеми строками добавлены 2 пробела, чтобы не нарушить отступы. Они к делу не относятся, и в данном случае считается, что метка EOF находится в начале строки, без отступа. — *Примеч. пер.*

попытка использовать системную команду `lockfile`. Она вызывается с заданным числом попыток и генерирует собственное сообщение об ошибке, если ей так и не удалось заблокировать файл. А что произойдет, если предложить сценарию снять блокировку (например, удалить файл-блокировку), которой в действительности нет? В результате будет сгенерировано другое сообщение об ошибке. В противном случае `lockfile` просто удалит блокировку.

Если говорить более конкретно, первый блок ❶ использует мощную функцию `getopts` для анализа всех поддерживаемых флагов (`-l`, `-u`, `-r`) в цикле `while`. Это наиболее типичный способ использования `getopts`, который снова и снова будет встречаться в книге. Обратите внимание на команду `shift $((OPTIND - 1))` в строке ❷: переменная `OPTIND` устанавливается функцией `getopts`, благодаря чему сценарий получает возможность сдвинуть входные параметры вниз (то есть значение параметра `$2` сместится в параметр `$1`, например), вытолкнув тем самым обработанные параметры, начинающиеся с дефиса.

Поскольку этот сценарий использует системную утилиту `lockfile`, он сначала проверяет ее доступность в списке путей пользователя ❸ и завершается с сообщением об ошибке, если утилита недоступна. Далее следует простая условная инструкция ❹, выясняющая, какая операция запрошена — блокировка или разблокировка, — и производится соответствующий вызов утилиты `lockfile`.

## Запуск сценария

Сценарий `filelock` относится к категории сценариев, которые редко используются сами по себе, и для его проверки потребуется открыть два окна терминала. Чтобы установить блокировку, просто укажите имя файла, который будет играть роль блокировки, в аргументе сценария `filelock`. Чтобы снять блокировку, запустите сценарий еще раз с флагом `-u`.

## Результаты

Сначала создадим заблокированный файл, как показано в листинге 1.23.

**Листинг 1.23.** Создание файла-блокировки командой `filelock`

```
$ filelock /tmp/exclusive.lck
$ ls -l /tmp/exclusive.lck
-r--r--r-- 1 taylor wheel 1 Mar 21 15:35 /tmp/exclusive.lck
```

Когда в следующий раз вы попытаетесь установить ту же блокировку, `filelock` выполнит указанное количество попыток (10 по умолчанию) и завершится с ошибкой (как показано в листинге 1.24):

**Листинг 1.24.** Ошибка при попытке создать файл-блокировку обращением к сценарию `filelock`

```
$ filelock /tmp/exclusive.lck
filelock : Failed: Couldn't create lockfile in time.
```

Завершив работу с файлом, можно освободить блокировку, как показано в листинге 1.25.

**Листинг 1.25.** Освобождение блокировки с помощью сценария `filelock`

```
$ filelock -u /tmp/exclusive.lck
```

Чтобы увидеть, как сценарий действует в двух терминалах, выполните команду разблокировки в одном из них, пока в другом сценарий крутится в цикле, пытаясь приобрести блокировку.

## Усовершенствование сценария

Поскольку наличие блокировки определяется сценарием, было бы полезно добавить еще один параметр, ограничивающий время ее действия. Если команда `lockfile` завершится неудачей, можно проверить последнее время доступа к файлу-блокировке и, если он старше значения этого параметра, безопасно удалить его, добавив, при желании, вывод предупреждающего сообщения.

Скорее всего, это не затронет вас, но `lockfile` не поддерживает работу с сетевой файловой системой (NFS) на смонтированных сетевых устройствах. Действительно надежный механизм блокировки файлов в NFS чрезвычайно сложен в реализации. Лучшее решение этой проблемы — всегда создавать файлы-блокировки только на локальных дисках или задействовать специализированный сценарий, способный управлять блокировками, используемыми несколькими системами.

## № 11. ANSI-последовательности управления цветом

Вероятно, вы замечали, что разные приложения командной строки поддерживают разные стили отображения текста. Существует большое количество вариантов оформления. Например, сценарий может выводить определенные слова жирным шрифтом или красным цветом на желтом фоне. Однако работать с ANSI-последовательностями (American National Standards Institute — американский национальный институт стандартов) очень неудобно из-за их сложности. Чтобы упростить их применение, в листинге 1.26 создается набор

переменных, значениями которых являются ANSI-последовательности, управляющие цветом и форматированием.

## Код

### Листинг 1.26. Функция initializeANSI

```
#!/bin/bash
# ANSI-последовательности управления цветом -- используйте эти переменные
# для управления цветом и форматом выводимого текста.
# Имена переменных, оканчивающиеся символом 'f', соответствуют цветам шрифта
# (foreground), а имена переменных, оканчивающиеся символом 'b', соответствуют
# цветам фона (background).

initializeANSI()
{
    esc="\033" # Если эта последовательность не будет работать,
               # введите символ ESC непосредственно.

    # Цвета шрифта
    blackf="${esc}[30m";   redf="${esc}[31m";   greenf="${esc}[32m"
    yellowf="${esc}[33m";  bluef="${esc}[34m";   purplef="${esc}[35m"
    cyanf="${esc}[36m";    whitef="${esc}[37m"

    # Цвета фона
    blackb="${esc}[40m";   redb="${esc}[41m";   greenb="${esc}[42m"
    yellowb="${esc}[43m";  blueb="${esc}[44m";   purpleb="${esc}[45m"
    cyanb="${esc}[46m";    whiteb="${esc}[47m"

    # Жирный, наклонный, с подчеркиванием и инверсное отображение
    boldon="${esc}[1m";    boldoff="${esc}[22m"
    italicson="${esc}[3m";  italicsoff="${esc}[23m"
    ulon="${esc}[4m";      uloff="${esc}[24m"
    invon="${esc}[7m";     invoff="${esc}[27m"

    reset="${esc}[0m"
}
```

## Как это работает

Если вы привыкли использовать язык разметки HTML, работа с этими последовательностями может показаться вам слишком сложной. В HTML вы просто вставляете открывающие теги и закрываете их в обратном порядке, следя за тем, чтобы закрыть все открытые теги. Чтобы выделить наклонным шрифтом фрагмент приложения, отображаемого жирным шрифтом, можно написать такой код HTML:

```
<b>this is in bold and <i>this is italics</i> within the bold</b>
```

Попытка закрыть тег, управляющий жирностью шрифта, раньше, чем тег, управляющий наклонным отображением, может вызвать беспорядок в отдельных веб-браузерах. Но в случае с ANSI-последовательностями дело обстоит иначе: некоторые из них фактически отменяют действие предыдущих, а также существует общая последовательность сброса, отменяющая действие всех других. Ее обязательно нужно добавить в конце вывода, а за последовательностью, включающей тот или иной режим форматирования, должна идти соответствующая ей последовательность, выключающая этот режим. Используя переменные из сценария, предыдущее предложение можно вывести, как показано ниже:

```
`${boldon}`this is in bold and `${italicson}`this is  
italics`${italicsoff}`within the bold`${reset}`
```

## Запуск сценария

Чтобы опробовать этот сценарий, нужно сначала вызвать функцию инициализации, а затем выполнить несколько команд `echo` с разными комбинациями цвета и эффектами форматирования:

```
initializeANSI  
  
echo -e "${yellowf}This is a phrase in yellow${redb} and red${reset}"  
echo -e "${boldon}This is bold${ulon} this is ul${reset} bye-bye"  
echo -e "${italicson}This is italics${italicsoff} and this is not"  
echo -e "${ulon}This is ul${uloff} and this is not"  
echo -e "${invon}This is inv${invoff} and this is not"  
echo -e "${yellowf}${redb}Warning I ${yellowb}${redf}Warning II${reset}"
```

## Результаты

Результаты работы сценария в листинге 1.27, воспроизведенные в книге, не впечатляют, но на экране, где поддерживаются все управляющие последовательности, они определенно привлекут ваше внимание.

**Листинг 1.27.** Как можно оформить текст с применением переменных из листинга 1.26

```
This is a phrase in yellow and red  
This is bold this is ul bye-bye  
This is italics and this is not  
This is ul and this is not  
This is inv and this is not  
Warning I Warning II
```

## Усовершенствование сценария

Запустив этот сценарий, можно увидеть такой вывод:

```
\033[33m\033[41mWarning!\033[43m\033[31mWarning!\033[0m
```

Эта проблема может заключаться в отсутствии поддержки управляющих ANSI-последовательностей в программе терминала или неправильной интерпретации формы записи `\033` в определении переменной `esc`. Чтобы устранить последнюю проблему, откройте сценарий в редакторе `vi` или в другом терминальном редакторе, удалите последовательность `\033` и нажмите клавиши `^V` (`ctrl-V`) и `esc`, в результате должна отобразиться последовательность `^[`. Если результат на экране выглядит как `esc="^[`, все должно заработать, как ожидается.

С другой стороны, если программа-терминал вообще не поддерживает ANSI-последовательности, стоит обновить ее, чтобы получить возможность расцвечивать и форматировать вывод других своих сценариев. Но прежде чем распрощаться со своим нынешним терминалом, проверьте его настройки — вполне вероятно, что там предусмотрены параметры для включения полноценной поддержки ANSI.

## № 12. Создание библиотечных сценариев

Многие сценарии в этой главе написаны как функции, а не самостоятельные сценарии, то есть их легко можно включить в другие сценарии без увеличения накладных расходов на выполнение дополнительных команд. Даже при том, что в командной оболочке отсутствует директива `#include`, как в языке C, в ней имеется операция *подключения файла-источника* (*sourcing*), которая служит тем же целям, позволяя подключать другие сценарии как библиотечные функции.

Чтобы понять важность этой операции, рассмотрим альтернативное решение. Если вызвать один сценарий командной оболочки из другого, по умолчанию он будет выполнен в собственной подоболочке. Проверить это можно экспериментально, как показано ниже:

```
$ echo "test=2" >> tinyscript.sh
$ chmod +x tinyscript.sh
$ test=1
$ ./tinyscript.sh
$ echo $test
1
```



Сценарий *tinyscript.sh* изменяет значение переменной `test`, но только внутри подоболочки, в которой он выполняется, то есть не затрагивая значение переменной `test` в текущей оболочке. Если выполнить сценарий с помощью точки (`.`), подключающей файл-источник, этот сценарий выполнится в текущей оболочке:

```
$ . tinyscript.sh
$ echo $test
2
```

Как нетрудно догадаться, если подключаемый таким способом сценарий выполнит команду `exit 0`, произойдет выход из текущей оболочки и окно программы терминала закроется, потому что операция подключения выполняет подключаемый сценарий в текущем процессе. В подоболочке команда `exit` произведет выход из нее, не вызвав остановки основного сценария. Это главное отличие и одна из причин, влияющих на выбор между командами `.` или `source` и `exec` (как будет показано ниже). Команда `.` фактически идентична команде `source` в `bash`; мы использовали точку просто потому, что такая форма подключения файлов более переносима между разными POSIX-совместимыми командными оболочками.

## Код

Чтобы превратить функции, представленные в этой главе, в библиотеку для использования в других сценариях, извлеките все функции и необходимые глобальные переменные или массивы (то есть значения, общие для нескольких функций) и поместите их в один большой файл. Если назвать этот файл *library.sh*, его можно использовать, как показано в тестовом сценарии из листинга 1.28, для доступа ко всем функциям, написанным в этой главе, и их проверки.

**Листинг 1.28.** Подключение единой библиотеки с прежде реализованными функциями и их вызов

```
#!/bin/bash

# Сценарий тестирования библиотеки

# Сначала подключить (прочитать) файл library.sh.
❶ . library.sh

initializeANSI # Настроить управляющие ANSI-последовательности.

# Проверить функцию validint.
echon "First off, do you have echo in your path? (1=yes, 2=no) "
read answer
```

```
while ! validint $answer 1 2 ; do
    echon "${boldon}Try again${boldoff}. Do you have echo "
    echon "in your path? (1=yes, 2=no) "
    read answer
done

# Проверить работу функции поиска команды в списке путей.
if ! checkForCmdInPath "echo" ; then
    echo "Nope, can't find the echo command."
else
    echo "The echo command is in the PATH."
fi

echo ""
echon "Enter a year you think might be a leap year: "
read year

# Убедиться, что значение года находится в диапазоне между 1 и 9999,
# с помощью validint, передав ей минимальное и максимальное значения.
while ! validint $year 1 9999 ; do
    echon "Please enter a year in the ${boldon}correct${boldoff} format: "
    read year
done

# Проверить, является ли год високосным.
if isLeapYear $year ; then
    echo "${greenf}You're right! $year is a leap year.${reset}"
else
    echo "${redf}Nope, that's not a leap year.${reset}"
fi

exit 0
```

## Как это работает

Обратите внимание, что библиотека и все содержащиеся в ней функции включаются в окружение сценария выполнением единственной строки ❶.

Этот очень удобный прием можно снова и снова использовать со многими сценариями, представленными в книге. Просто поместите подключаемый библиотечный файл в один из каталогов, перечисленных в переменной окружения `PATH`, чтобы команда `.` могла найти его.

## Запуск сценария

Чтобы запустить тестовый сценарий, вызовите его из командной строки, подобно любому другому сценарию, как показано в листинге 1.29.

## Результаты

### Листинг 1.29. Запуск сценария library-test

```
$ library-test
First off, do you have echo in your PATH? (1=yes, 2=no) 1
The echo command is in the PATH.

Enter a year you think might be a leap year: 432423
Your value is too big: largest acceptable value is 9999.
Please enter a year in the correct format: 432
You're right! 432 is a leap year.
```

В случае ввода слишком большого значения, сообщение об ошибке будет показано жирным шрифтом. Кроме того, сообщение, подтверждающее правильность выбранного високосного года, отображается зеленым цветом.

Исторически 432 год не считается високосным, потому что учет високосных лет не производился до 1752 года. Но мы говорим о сценариях командной оболочки, а не о хитрости летоисчисления, так что оставим эту неточность без внимания.

## № 13. Отладка сценариев

Этот раздел не содержит настоящего сценария, но мы хотели бы потратить несколько страниц в книге, чтобы поговорить об основах отладки сценариев, потому что рано или поздно вы все равно столкнетесь с ошибками!

По нашему опыту, лучшая стратегия отладки — наращивать возможности сценариев постепенно. Некоторые программисты оптимистично надеются, что все заработает правильно с первого раза, но вы будете по-настоящему уверены двигаться вперед, если начнете с малого. Кроме того, для трассировки переменных можно свободно использовать команды `echo`, а также запускать сценарии командой `bash -x`, чтобы обеспечить вывод отладочной информации, например:

```
$ bash -x myscript.sh
```

Как вариант, можно добавить команду `set -x` перед началом отлаживаемого фрагмента и `set +x` — после него, как показано ниже:

```
$ set -x
$ ./myscript.sh
$ set +x
```

Чтобы увидеть, как действуют флаги `-x` и `+x`, попробуем отладить простую игру «угадай число», представленную в листинге 1.30.

## Код

**Листинг 1.30.** Сценарий `hi_low`, возможно содержащий несколько ошибок, который нужно отладить...

```
#!/bin/bash
# hi_low -- Простая игра "угадай число"

biggest=100          # Максимальное возможное число
guess=0              # Число, предложенное игроком
guesses=0             # Количество попыток
❶ number=$(( $RANDOM % $biggest )) # Случайное число от 1 до $biggest
echo "Guess a number between 1 and $biggest"

while [ "$guess" -ne $number ] ; do
❷  /bin/echo -n "Guess? " ; read answer
  if [ "$guess" -lt $number ] ; then
❸    echo "... bigger!"
  elif [ "$guess" -gt $number ] ; then
❹    echo "... smaller!"
  fi
  guesses=$(( $guesses + 1 ))
done

echo "Right!! Guessed $number in $guesses guesses."

exit 0
```

## Как это работает

Чтобы было понятнее, как происходит получение случайного числа в ❶, напомним, что специальная переменная `$RANDOM` хранит числовой идентификатор процесса (Process ID, PID) командной оболочки, в которой выполняется сценарий. Обычно это 5- или 6-значное число. При каждом запуске сценарий получает новый PID. Последовательность `% $biggest` делит значение PID на заданное наибольшее значение и возвращает остаток. Иными словами, `5 % 4 = 1`, так же как `41 % 4`. Это простой способ получения псевдослучайных чисел в диапазоне от 1 до `$biggest`.

## Запуск сценария

Отлаживая игру, прежде всего проверим и убедимся, что генерируемое число достаточно случайно. Для этого получим PID оболочки, в которой выполняется сценарий, и приведем его к требуемому диапазону, используя операцию `%` извлечения остатка от деления нацело ❶. Для проверки операции введите в командной строке следующие команды:

```
$ echo $(( $$ % 100 ))
5
$ echo $(( $$ % 100 ))
5
$ echo $(( $$ % 100 ))
5
```

Операция работает, но числа не выглядят случайными. Если немного поразмыслить, становится понятно, почему так происходит: когда команда выполняется непосредственно в командной строке, она всегда получает одно и то же значение PID; но внутри сценария команда каждый раз будет выполняться в другой оболочке, с другим значением PID.

Еще один способ получить случайное число — воспользоваться переменной окружения `$RANDOM`. Это не простая переменная! При каждом обращении к ней вы будете получать разные значения. Чтобы получить число в диапазоне от 1 до `$biggest`, используйте в строке ❶ выражение `$(( $RANDOM % $biggest + 1 ))`.

Следующий шаг — добавление основной логики игры. В ❶ генерируется случайное число в диапазоне от 1 до 100; в ❷ пользователь делает попытку угадать это число; затем пользователю сообщается, что число слишком большое ❸ или слишком маленькое ❹, пока он наконец не угадает правильное значение. После ввода всего основного кода можно попробовать запустить сценарий и посмотреть, как он работает. Ниже демонстрируется проверка работы сценария из листинга 1.30:

```
$ hilow
./013-hilow.sh: line 19: unexpected EOF while looking for matching '"'
./013-hilow.sh: line 22: syntax error: unexpected end of file
```

Опля! Мы столкнулись с проклятием разработчиков сценариев: неожиданный конец файла (EOF). Сообщение говорит, что ошибка находится в строке 19, но это не означает, что она действительно там. На самом деле строка 19 не содержит ошибок:

```
$ sed -n 19p hilow
echo "Right!! Guessed $number in $guesses guesses."
```

Чтобы понять причину ошибки, вспомните, что строки в кавычках могут содержать символы перевода строки. То есть, встретив кавычки, по ошибке не закрытые как следует, командная оболочка просто продолжит читать сценарий, стараясь найти парную закрывающую кавычку, и останавливается, только встретив самую последнюю и обнаружив, что в сценарии что-то неправильно.

Следовательно, проблема должна находиться где-то выше. В сообщении об ошибке есть единственная полезная деталь — оно указывает, какой символ не был найден. То есть можно попробовать с помощью `grep` извлечь все строки, содержащие кавычки, и затем отфильтровать те из них, что содержат по две кавычки, как показано ниже:

```
$ grep '"' 013-hilow.sh | egrep -v '.*".*'.*'
echo "... smaller!"
```

Вот и все! В строке ❹, сообщающей, что число, предложенное пользователем, слишком мало, отсутствует закрывающая кавычка. Добавим ее в конец строки и повторим попытку запустить сценарий:

```
$ hilow
./013-hilow.sh: line 7: unexpected EOF while looking for matching ')'
./013-hilow.sh: line 22: syntax error: unexpected end of file
```

Не вышло. Еще одна проблема. Выражений в круглых скобках в сценарии немного, поэтому мы можем просто посмотреть и увидеть, что в выражении, вычисляющем случайное число, отсутствует закрывающая скобка:

```
number=$(( $RANDOM % $biggest ) # Случайное число от 1 до $biggest
```

Исправим эту ошибку, добавив закрывающую круглую скобку в конец выражения, но перед комментарием. А теперь игра заработает? Давайте попробуем:

```
$ hilow
Guess? 33
... bigger!
Guess? 66
... bigger!
Guess? 99
... bigger!
Guess? 100
... bigger!
Guess? ^C
```

Почти получилось. Но при попытке ввести максимально возможное значение 100 появляется ответ, что загаданное число больше (*bigger*), значит, в логике игры допущена ошибка. Искать такие ошибки особенно сложно, потому что никакая, даже самая замысловатая команда `grep` или `sed` не поможет выявить проблему. Вернитесь к коду и попробуйте найти ошибку самостоятельно.

Чтобы упростить поиск, можно добавить несколько команд `echo`, вывести значение, выбранное пользователем, и проверить, какое число введено и какое проверяется. Соответствующий раздел кода начинается в строке ❷, но для удобства приведем эти строки еще раз:

```
/bin/echo -n "Guess? " ; read answer
if [ "$guess" -lt $number ] ; then
```

Изменив команду `echo` и исследовав эти две строки, мы заметили ошибку: ввод пользователя читается в переменную `answer`, а проверяется переменная `guess`. Глупая, но не такая уж редкая ошибка (особенно если имеются переменные с необычными для вас именами). Чтобы исправить ошибку, нужно заменить `read answer` на `read guess`.

## Результаты

Наконец сценарий работает правильно, как показано в листинге 1.31.

**Листинг 1.31.** Сценарий `hilow` работает без ошибок

```
$ hilow
Guess? 50
... bigger!
Guess? 75
... bigger!
Guess? 88
... smaller!
Guess? 83
... smaller!
Guess? 80
... smaller!
Guess? 77
... bigger!
Guess? 79
Right!! Guessed 79 in 7 guesses.
```

## Усовершенствование сценария

Самая досадная ошибка, кроющаяся в этом маленьком сценарии, — отсутствие проверки ввода. Попробуйте ввести произвольную строку вместо числа, и сценарий завершится с сообщением об ошибке. Мы легко могли бы добавить элементарную проверку, включив следующие строки в цикл `while`:

```
if [ -z "$guess" ] ; then
    echo "Please enter a number. Use ^C to quit"; continue;
fi
```

Но непустой ввод еще не означает, что введено число, и, если ввести произвольную строку, например `hi`, сценарий все еще будет завершаться с ошибкой. Чтобы исправить эту проблему, добавьте вызов функции `validint` из сценария № 5.

## Глава 2. Усовершенствование пользовательских команд

Типичная система Unix или Linux по умолчанию включает сотни команд, которые, с учетом многообразия флагов и способов сочетания команд посредством каналов, дают миллионы разных вариантов работы в командной строке.

Прежде чем двинуться дальше, взгляните на листинг 2.1, в котором приводится премиальный сценарий, подсчитывающий количество команд, доступных в списке каталогов PATH.

**Листинг 2.1.** Подсчет количества выполняемых и невыполняемых файлов в текущем списке PATH

```
#!/bin/bash

# Подсчет количества команд: простой сценарий для подсчета количества выполняемых
# команд в каталогах из списка PATH

IFS=":"
count=0 ; nonex=0
for directory in $PATH ; do
    if [ -d "$directory" ] ; then
        for command in "$directory"/* ; do
            if [ -x "$command" ] ; then
                count=$(( $count + 1 ))
            else
                nonex=$(( $nonex + 1 ))
            fi
        done
    fi
done

echo "$count commands, and $nonex entries that weren't executable"

exit 0
```

Этот сценарий подсчитывает не просто файлы, а выполняемые файлы, и может использоваться для оценки количества команд и невыполняемых файлов в каталогах из списка PATH в разных системах (табл. 2.1).



**Таблица 2.1.** Типичное количество команд в разных ОС

Операционная система	Команд	Невыполняемых файлов
Ubuntu 15.04 (включая все библиотеки для разработки)	3156	5
OS X 10.11 (со всеми установленными инструментами для разработки)	1663	11
FreeBSD 10.2	954	4
Solaris 11.2	2003	15

Очевидно, что разные версии Linux и Unix предлагают разное количество команд и сценариев. Почему их так много? Ответ заключается в основополагающей философии Unix: всякая команда должна делать что-то одно и делать это хорошо. Текстовый процессор, включающий функции проверки орфографии, поиска файлов и работы с электронной почтой, возможно, хорошо подходит для мира Windows и Mac, но в командной строке все эти функции должны существовать и быть доступны по отдельности.

Философия Unix имеет много преимуществ, и самое большое заключается в том, что каждая функция способна расширяться и совершенствоваться независимо от других, предоставляя новые возможности всем приложениям, использующим ее. Для решения практически любой задачи в Unix обычно достаточно объединить какие-нибудь команды, которые легко справятся с работой, загрузить новую утилиту, которая расширит возможности системы, создать несколько псевдонимов или написать свой сценарий командной оболочки.

Сценарии, демонстрирующиеся в книге, полезны не только как учебные примеры, но также как логическое расширение философии Unix. В конце концов, лучше дополнять и расширять, чем создавать сложные, несовместимые версии команд для личного использования.

Сценарии, рассмотренные в данной главе, похожи на сценарий в листинге 2.1 тем, что добавляют интересные и полезные средства и возможности без лишних сложностей. Некоторые сценарии поддерживают различные флаги для большей гибкости, а некоторые демонстрируют, как создаются *обертки* для программ, позволяющие пользователям указывать команды или флаги в привычной форме и затем преобразующие эти флаги в вид, соответствующий требованиям фактической команды.

## № 14. Форматирование длинных строк

Если вам повезло, в вашей системе Unix имеется команда `fmt` — программа, особенно удобная для работы с обычным текстом. `fmt` — утилита,

с которой действительно стоит познакомиться. Ее можно использовать для форматирования электронных писем или выравнивания по ширине строк в документах.

Однако в некоторых системах Unix команда `fmt` отсутствует. В особенности это относится к устаревшим системам, часто имевшим минимальную реализацию.

Как оказывается, команда `nroff`, входившая в состав Unix с самого начала, является сценарием-оберткой и может использоваться для переноса длинных строк и заполнения коротких строк для их выравнивания, как показано в листинге 2.2.

## Код

**Листинг 2.2.** Сценарий `fmt` для форматирования длинных текстовых строк

```
#!/bin/bash

# fmt -- утилита форматирования текста, действующая как обертка для nroff
# Добавляет два флага: -w X, для задания ширины строк,
# и -h, для расстановки переносов и улучшения выравнивания
❶ while getopts "hw:" opt; do
    case $opt in
        h ) hyph=1          ;;
        w ) width="$OPTARG" ;;
        esac
    done
❷ shift $((OPTIND - 1))
❸ nroff << EOF
❹ .ll ${width:-72}
   .na
   .hy ${hyph:-0}
   .pl 1
❺ $(cat "$@")
EOF

exit 0
```

## Как это работает

Этот короткий сценарий реализует поддержку двух дополнительных флагов: `-w X`, для ограничения ширины строк `X` символами (по умолчанию 72), и `-h`, разрешающий разрывать слова и расставлять переносы. Обратите внимание на проверку флагов в ❶. Цикл `while` вызывает `getopts`, чтобы прочитать каждый параметр, переданный сценарию, а внутренний блок `case` решает, что делать с ними. После анализа флагов сценарий вызывает `shift` в строке ❷, чтобы отбросить проанализированные параметры, для чего используется переменная

`$OPTIND` (хранящая индекс следующего аргумента, который должна была бы прочитать функция `getopts`), и оставляет прочие аргументы для последующей обработки.

В сценарии также используется встроенный документ (обсуждался в сценарии № 9, в главе 1) — особый блок кода, который можно использовать для передачи нескольких строк на вход команды. Используя это удобное средство, сценарий в ❸ передает сценарию `nrff` все команды, необходимые для получения желаемого результата. В этом документе используется типичный для `bash` прием подстановки значения вместо неопределенной переменной ❹, чтобы передать разумное значение по умолчанию, если пользователь не указал свое. Наконец, сценарий вызывает команду `cat` с именами файлов, подлежащих обработке. Для выполнения поставленной задачи вывод команды `cat` передается команде `nrff` ❺. Этот прием часто будет встречаться в данной книге.

## Запуск сценария

Этот сценарий можно запустить непосредственно из командной строки, но вероятнее всего он станет частью внешнего конвейера, запускаемого редактором, таким как `vi` или `vim` (например, `!}fmt`), для форматирования абзаца текста.

## Результаты

Команда в листинге 2.3 разрешает расстановку переносов и задает максимальную ширину 50 символов.

**Листинг 2.3.** Форматирование текста с помощью сценария `fmt` путем расстановки переносов и ограничения ширины текста 50 символами

```
$ fmt -h -w 50 014-ragged.txt
```

```
So she sat on, with closed eyes, and half believed
herself in Wonderland, though she knew she had but
to open them again, and all would change to dull
reality--the grass would be only rustling in the
wind, and the pool rippling to the waving of the
reeds--the rattling teacups would change to tin-
kling sheep-bells, and the Queen's shrill cries
to the voice of the shepherd boy--and the sneeze
of the baby, the shriek of the Gryphon, and all
the other queer noises, would change (she knew) to
the confused clamour of the busy farm-yard--while
the lowing of the cattle in the distance would
take the place of the Mock Turtle's heavy sobs.
```

Сравните содержимое в листинге 2.3 (обратите внимание, как был выполнен перенос слова **tinkling**, выделенного жирным в строках 6 и 7) с выводом в листинге 2.4, полученным с использованием ширины по умолчанию и запрещенными переносами.

**Листинг 2.4.** Форматирование по умолчанию без переносов, осуществляемое сценарием `fmt`

```
$ fmt 014-ragged.txt
```

```
So she sat on, with closed eyes, and half believed herself in
Wonderland, though she knew she had but to open them again, and all
would change to dull reality--the grass would be only rustling in the
wind, and the pool rippling to the waving of the reeds--the rattling
teacups would change to tinkling sheep-bells, and the Queen's shrill
cries to the voice of the shepherd boy--and the sneeze of the baby, the
shriek of the Gryphon, and all the other queer noises, would change (she
knew) to the confused clamour of the busy farm-yard--while the lowing of
the cattle in the distance would take the place of the Mock Turtle's
heavy sobs.
```

## № 15. Резервное копирование файлов при удалении

Одна из распространенных проблем, с которыми часто сталкиваются пользователи Unix, — сложность восстановления удаленных по ошибке файлов или каталогов. В Unix нет приложения, такого же удобного, как Undelete 360, WinUndelete или утилита для OS X, которое позволяло бы просматривать и восстанавливать удаленные файлы щелчком на кнопке. Как только вы нажмете клавишу `enter` после ввода команды `rm filename`, файл станет историей.

Чтобы решить эту проблему, нужно организовать тайное и автоматическое архивирование файлов и каталогов в архив `.deleted-files`. Немного подумав, можно написать сценарий (представленный в листинге 2.5), который сделает все это почти незаметно для пользователя.

### Код

**Листинг 2.5.** Сценарий `newrm`, копирующий файлы перед удалением с диска

```
#!/bin/bash

# newrm -- замена существующей команды rm.
# Этот сценарий предоставляет простую возможность восстановления, создавая и
# используя новый каталог в домашнем каталоге пользователя. Может обрабатывать
# каталоги и отдельные файлы. Если пользователь добавляет флаг -f, файлы
# удаляются БЕЗ архивирования.
```

```

# Важное предупреждение: возможно, вам понадобится создать задание для cron или
# нечто подобное для очистки удаленных каталогов и файлов через некоторое
# время. Иначе файлы не будут удаляться из системы и вы рискуете исчерпать
# дисковое пространство!

archivedir="$HOME/.deleted-files"
realrm="$(which rm)"
copy="$(which cp) -R"

if [ $# -eq 0 ] ; then # Позволить 'rm' вывести сообщение о порядке
использования.
    exec $realrm # Our shell is replaced by /bin/rm.
fi

# Проверить все параметры на наличие флага '-f'

flags=""

while getopts "dfiPRrvW" opt
do
    case $opt in
        f ) exec $realrm "$@" ;; # exec позволяет покинуть сценарий немедленно.
        * ) flags="$flags -$opt" ;; # Другие флаги предназначены команде rm.
    esac
done
shift $(( $OPTIND - 1 ))

# НАЧАЛО ОСНОВНОГО СЦЕНАРИЯ
# =====

# Гарантировать наличие каталога $archivedir.

❶ if [ ! -d $archivedir ] ; then
    if [ ! -w $HOME ] ; then
        echo "$0 failed: can't create $archivedir in $HOME" >&2
        exit 1
    fi
    mkdir $archivedir
❷ chmod 700 $archivedir # Ограничить доступ к каталогу.
fi

for arg
do
❸ newname="$archivedir/$(date "+%S.%M.%H.%d.%m").$(basename "$arg")"
    if [ -f "$arg" -o -d "$arg" ] ; then
        $copy "$arg" "$newname"
    fi
done
❹ exec $realrm $flags "$@" # Текущий сценарий будет вытеснен командой realrm.

```

## Как это работает

В этом сценарии есть много интересных аспектов, в основном связанных с необходимостью скрыть его работу от пользователя. Например, сценарий не генерирует сообщений об ошибках в ситуациях, когда обнаруживает, что не может продолжить работу; он просто позволяет команде `realrm` самой сгенерировать такое сообщение, вызывая (обычно) `/bin/rm` с иногда ошибочными параметрами. Вызов `realrm` производится с помощью команды `exec`, которая замещает текущий процесс новым, выполняющим указанную команду. Сразу после вызова команды `exec realrm` ❹ текущий сценарий фактически прекращает работу, и в вызывающую командную оболочку передается код возврата, генерируемый процессом `realrm`.

Поскольку сценарий втайне создает в домашнем каталоге пользователя новый каталог ❶, он должен гарантировать, что хранимые в нем файлы не окажутся доступны для других только из-за неправильно настроенного значения `umask`. (Значение `umask` определяет привилегии доступа по умолчанию для создаваемых файлов и каталогов.) Чтобы избежать непреднамеренного открытия доступа к резервируемым файлам, сценарий вызывает в строке ❷ команду `chmod`, дающую право на доступ к каталогу только для текущего пользователя.

Наконец, в строке ❸ сценарий использует `basename` для удаления любой информации о каталоге из пути к файлу и добавляет в имя файла дату и время удаления в формате: *секунды.минуты.часы.день.месяц.имя\_файла*:

```
newname="$archivedir/${date "+%S.%M.%H.%d.%m").$(basename "$arg")"
```

Обратите внимание на использование нескольких элементов `$( )` для формирования нового имени файла. Хотя это немного усложняет сценарий, тем не менее такое решение эффективно. Напомним, что содержимое, заключенное между `$( )` выполняется в подоболочке, а результат замещает выражение в скобках.

Но зачем усложнять реализацию добавлением даты и времени в имя резервируемого файла? Чтобы дать возможность сохранять несколько копий удаляемого файла с одним и тем же именем. После архивирования файла сценарием нельзя будет отличить `/home/oops.txt` от `/home/subdir/oops .txt` иначе как по времени удаления. Если стирание одноименных файлов произойдет одновременно (или в течение одной секунды), резервные копии файлов, удаленных первыми, будут затерты. Для решения этой проблемы можно организовать добавление абсолютных путей к оригинальным файлам в имена резервных копий.

## Запуск сценария

Чтобы установить сценарий, добавьте псевдоним — тогда при вводе команды `rm` действительно будет вызываться этот сценарий, а не команда `/bin/rm`. В командных оболочках `bash` и `ksh` псевдонимы определяются так:

```
alias rm=yourpath/newrm
```

## Результаты

Результаты работы этого сценария преднамеренно скрыты (как показывает листинг 2.6), так что обратим все внимание на каталог *.deleted-files*.

### Листинг 2.6. Тестирование сценария `newrm`

```
$ ls ~/.deleted-files
ls: /Users/taylor/.deleted-files/: No such file or directory
$ newrm file-to-keep-forever
$ ls ~/.deleted-files/
51.36.16.25.03.file-to-keep-forever
```

Что и требовалось получить. Файл был удален из локального каталога и скрытно перемещен в каталог *.deleted-files*. Добавление префикса с временем удаления позволяет сохранять в каталоге одноименные файлы, удаленные в разное время, не затирая их.

## Усовершенствование сценария

Как одно из усовершенствований можно предложить изменить префикс со временем, чтобы упростить вывод списка копий удаленных файлов командой `ls` в обратном хронологическом порядке. Ниже показана строка из сценария, подлежащая изменению:

```
newname="$archivedir/${date "+%S.%M.%H.%d.%m"}.${basename "$arg"}"
```

Можно изменить порядок следования компонентов в новом имени на противоположный, чтобы исходное имя файла следовало первым, а за ним — дата удаления в секундах. Далее, поскольку время измеряется с точностью до секунды, может так получиться, что при одновременном удалении одноименных файлов из разных каталогов (например, `rm test testdir/test`) произойдет затирание одной копии удаленного файла другой. Поэтому, как еще одно полезное усовершенствование, можно добавить в имя архивируемого файла его прежнее местоположение, чтобы в результате получить, например, файлы *timestamp.test* и *timestamp.testdir.test*, явно отличающиеся друг от друга.

## № 16. Работа с архивом удаленных файлов

Теперь, когда в домашней папке пользователя появился скрытый каталог с удаленными файлами, пригодился бы сценарий, позволяющий выбирать для восстановления одну из нескольких удаленных версий. Однако эта задача сложна тем, что нам придется предусмотреть все вероятные проблемы: от невозможности найти требуемый файл до обнаружения нескольких копий, соответствующих заданному критерию. Например, если обнаружится несколько совпадений, какую копию должен восстановить сценарий — самую старую или самую новую? Или он должен вывести сообщение об ошибке, указав в нем количество найденных совпадений? Или вывести список версий и предложить пользователю выбрать нужную? Давайте посмотрим, как решаются эти проблемы на практике, изучив сценарий 2.7, в котором приводится сценарий командной оболочки `unrm`.

### Код

**Листинг 2.7.** Сценарий `unrm` для восстановления файлов из резервных копий

```
#!/bin/bash

# unrm -- отыскивает в архиве удаленных файлов требуемый файл или
# каталог. Если найдено более одного совпадения, выводит список
# результатов поиска, упорядоченных по времени, и предлагает
# пользователю выбрать нужный для восстановления.

archivedir="$HOME/.deleted-files"
realrm="$(which rm)"
move="$(which mv)"

dest=$(pwd)

if [ ! -d $archivedir ] ; then
    echo "$0: No deleted files directory: nothing to unrm" >&2
    exit 1
fi

cd $archivedir

# Если сценарий запущен без аргументов, просто вывести список
# удаленных файлов.
❶ if [ $# -eq 0 ] ; then
    echo "Contents of your deleted files archive (sorted by date):"
❷ ls -FC | sed -e 's/\([[[:digit:]]\)[[:digit:]]\.\.\){5}\//g' \
    -e 's/^/ /'
    exit 0
fi
```



```

# Иначе принять шаблон для поиска, предложенный пользователем.
# Проверить наличие в архиве нескольких совпадений с шаблоном

❶ matches="$(ls -d *"$1" 2> /dev/null | wc -l)"
if [ $matches -eq 0 ] ; then
    echo "No match for \"$1\" in the deleted file archive." >&2
    exit 1
fi

❷ if [ $matches -gt 1 ] ; then
    echo "More than one file or directory match in the archive:"
    index=1
    for name in $(ls -td *"$1")
    do
        datetime="$(echo $name | cut -c1-14| \
❸      awk -F. '{ print $5/"$4" at "$3":"$2":"$1 }')'"
        filename="$(echo $name | cut -c16-)"
        if [ -d $name ] ; then
❹      filecount="$(ls $name | wc -l | sed 's/^[[:digit:]]//g')'"
            echo " $index) $filename (contents = ${filecount} items,\" \
                " deleted = $datetime)"
        else
❺      size="$(ls -sd1 $name | awk '{print $1}')"
            echo " $index) $filename (size = ${size}Kb, deleted = $datetime)"
        fi
        index=$(( $index + 1))
    done
    echo ""
    /bin/echo -n "Which version of $1 should I restore ('0' to quit)? [1] : "
    read desired
    if [ ! -z "$(echo $desired | sed 's/[[[:digit:]]//g')" ] ; then
        echo "$0: Restore canceled by user: invalid input." >&2
        exit 1
    fi

    if [ ${desired:=1} -ge $index ] ; then
        echo "$0: Restore canceled by user: index value too big." >&2
        exit 1
    fi

    if [ $desired -lt 1 ] ; then
        echo "$0: Restore canceled by user." >&2
        exit 1
    fi

❻ restore="$(ls -td1 *"$1" | sed -n "${desired}p")"

❼ if [ -e "$dest/$1" ] ; then
    echo "\"$1\" already exists in this directory. Cannot overwrite." >&2
    exit 1
fi

```

```

/bin/echo -n "Restoring file \"$1\" ..."
$move "$restore" "$dest/$1"
echo "done."

❶ /bin/echo -n "Delete the additional copies of this file? [y] "
read answer

if [ ${answer:=y} = "y" ] ; then
    $realrm -rf *"$1"
    echo "Deleted."
else
    echo "Additional copies retained."
fi
else
if [ -e "$dest/$1" ] ; then
    echo "\"$1\" already exists in this directory. Cannot overwrite." >&2
    exit 1
fi

restore="$(ls -d *"$1")"

/bin/echo -n "Restoring file \"$1\" ... "
$move "$restore" "$dest/$1"
echo "Done."
fi

exit 0

```

## Как это работает

Первый фрагмент кода в ❶, блок в условной инструкции `if [ $# -eq 0 ]`, выполняется, если сценарий запущен без аргументов. Он выводит содержимое архива удаленных файлов. Однако тут есть одна загвоздка: нам нужно вывести имена файлов без префикса со временем удаления, потому что он предназначен только для внутреннего использования. Префикс только ухудшил бы читаемость списка. Для решения этой задачи применяется команда `sed` в ❷, которая удаляет первые пять вхождений шаблона «цифра цифра точка» из каждой строки в выводе команды `ls`.

Пользователь может указать в аргументе имя файла или каталога для восстановления. Следующий шаг в ❸ — проверка количества совпадений с именем, указанным пользователем.

Необычное применение вложенных двойных кавычек в этой строке (вокруг `$1`) позволяет команде `ls` находить совпадения с именами файлов, содержащими пробелы, а шаблонный символ `*` разрешает совпадения с именами, включающими произвольные префиксы с временем удаления. Последовательность `2> /dev/null` нужна, чтобы скрыть любые сообщения об ошибках от пользователя,

выводимые командой. С наибольшей вероятностью будет скрыто сообщение об ошибке «No such file or directory» («Нет такого файла или каталога»), которое выводит команда `ls`, когда не может найти файл с указанным именем.

При наличии нескольких совпадений с указанным именем файла или каталога выполняется самая сложная часть сценария — блок в инструкции `if [ $matches -gt 1 ]` ④, который выводит все результаты. Флаг `-t` в команде `ls`, вызываемой в главном цикле `for`, обеспечивает перебор файлов в архиве в обратном хронологическом порядке — от более новых к более старым, а вызов команды `awk` в ⑤ преобразует префикс в имени файла в дату и время удаления в круглых скобках. В строке ⑦ определяется размер файла в килобайтах, для чего вызывается команда `ls` с флагом `-k`.

Вместо размера записи, соответствующей каталогу в структуре файловой системы, сценарий выводит более полезную информацию — количество файлов в каждом совпавшем каталоге. Вычисляется оно очень просто. В ⑥ просто подсчитывается количество строк в выводе команды `ls` и отбрасываются любые пробелы из вывода команды `wc`.

Когда пользователь выберет одно из совпадений, команда в ⑧ получит точное имя файла для восстановления. Эта команда чуть иначе использует `sed`. Здесь с помощью флага `-n` строчному редактору `sed` передается номер строки (`${desired}`) и команда `p` (`print` — печать), что позволяет быстро извлечь из потока ввода указанную строку. Хотите увидеть только строку с номером 37? Команда `sed -n 37p` сделает это.

Далее, в строке ⑨, сценарий `unrm` проверяет, не затрет ли он существующий файл, и затем восстанавливает файл или каталог вызовом команды `/bin/mv`. После этого в ⑩ пользователю дается возможность удалить все остальные (вероятно, избыточные) копии файла, и сценарий завершается.

Обратите внимание, что команда `ls` с шаблоном `*"$1"` найдет все файлы, имена которых оканчиваются значением параметра `$1`, поэтому список с «совпавшими файлами» может содержать не только файл, который пользователь хотел бы восстановить. Например, если удаляемый каталог содержал файлы `11.txt` и `111.txt`, команда `unrm 11.txt` сообщит, что найдено несколько совпадений и вернет список с обоими файлами, `11.txt` и `111.txt`. На первый взгляд в этом нет ничего страшного, но как только пользователь выберет файл для восстановления (`11.txt`) и ответит утвердительно на предложение удалить другие копии, сценарий удалит также файл `111.txt`. Такое поведение по умолчанию в некоторых случаях может оказаться нежелательным. Однако это легко исправить, используя шаблон `???.???.???.??"$1"`, если в сценарии `newrm` сохранен формат префикса в именах копий.

## Запуск сценария

Сценарий можно запустить двумя способами. Если запустить его без аргументов, он выведет список всех файлов и каталогов в архиве удаленных файлов.

Если передать сценарию аргумент с именем файла, он попытается восстановить этот файл или каталог (если найдет только одно совпадение) или выведет список найденных кандидатов на восстановление и предложит пользователю выбрать нужную версию файла или каталога.

## Результаты

При запуске без аргументов сценарий выведет список всех файлов и каталогов в архиве удаленных файлов, как показано в листинге 2.8.

**Листинг 2.8.** При запуске без аргументов сценарий `unrm` выведет список файлов и каталогов, доступных для восстановления

```
$ unrm
Contents of your deleted files archive (sorted by date):
  detritus              this is a test
  detritus              garbage
```

Получив аргумент с именем файла, сценарий выведет больше информации о файлах, если найдет несколько совпадений с указанным именем, как показано в листинге 2.9.

**Листинг 2.9.** При запуске с единственным аргументом сценарий `unrm` попытается восстановить файл

```
$ unrm detritus
More than one file or directory match in the archive:
  1) detritus (size = 7688Kb, deleted = 11/29 at 10:00:12)
  2) detritus (size = 4Kb, deleted = 11/29 at 09:59:51)

Which version of detritus should I restore ('0' to quit)? [1] : 0
unrm: Restore canceled by user.
```

## Усовершенствование сценария

Используйте этот сценарий внимательно, потому что в нем не выполняется никаких проверок и отсутствуют всякие ограничения. Объем архива с удаленными файлами будет расти без всяких ограничений. Чтобы избежать исчерпания дискового пространства, создайте задание для `cron`, вызывающее команду `find`, для очистки удаленных файлов, с флагом `-mtime`, чтобы выявить файлы,

остававшиеся невостребованными в течение нескольких недель. 14-дневного срока хранения в архиве, вероятно, будет вполне достаточно и для большинства пользователей, и для того, чтобы предотвратить исчерпание дискового пространства.

Можно также внести ряд других усовершенствований, которые сделают сценарий более дружественным для пользователя. Например, добавить флаг `-l` для восстановления последней (latest) копии и флаг `-D` для удаления дополнительных копий файла. Подумайте, какие еще флаги вы добавили бы, чтобы упростить работу со сценарием?

## № 17. Журналирование операций удаления файлов

Вместо архивирования удаляемых файлов иногда достаточно просто фиксировать факты удаления. В листинге 2.10 приводится сценарий, который журналирует вызовы команды `rm` в отдельном файле, ни о чем не извещая пользователя.

Такого эффекта можно добиться, используя сценарий в роли обертки. Основная идея любой обертки состоит в том, что она располагается между фактической командой Unix и пользователем, предлагая дополнительные возможности, недоступные в оригинальной команде.

---

### ПРИМЕЧАНИЕ

Обертки — мощная концепция, и в этой книге вы еще не раз встретитесь с ней.

---

### Код

#### Листинг 2.10. Сценарий `logrm`

```
#!/bin/bash

# logrm -- журналирует все операции удаления файлов, если вызывается без флага -s

removelog="/var/log/remove.log"

❶ if [ $# -eq 0 ] ; then
    echo "Usage: $0 [-s] list of files or directories" >&2
    exit 1
fi

❷ if [ "$1" = "-s" ] ; then
```

```
# Запрошена операция без журналирования...
shift
else
❸ echo "$(date): ${USER}: $@" >> $removelog
fi

❹ /bin/rm "$@"

exit 0
```

## Как это работает

Первая условная инструкция в ❶ проверяет ввод пользователя и показывает сообщение, описывающее порядок использования сценария, если он вызван без аргументов. Затем, в строке ❷, сценарий проверяет, не содержит ли аргумент \$1 флаг -s; если содержит, сценарий пропустит операцию журналирования. В заключение сценарий записывает текущее время, имя пользователя и текст команды в файл *\$removelog* ❸, и передает свои параметры фактической программе */bin/rm* ❹.

## Запуск сценария

Обычно при установке программ-оберток, таких как сценарий *logrm*, обертываемые команды переименовываются, а оберткам присваиваются имена оригинальных команд. Если вы решите пойти этим путем, убедитесь, что обертка вызывает переименованную программу, а не саму себя! Например, если вы переименовали */bin/rm* в */bin/rm.old*, а сценарий сохранили с именем */bin/rm*, тогда в предпоследней строке сценария замените вызов */bin/rm* на */bin/rm.old*.

Как вариант, можно определить псевдоним, чтобы заменить стандартный вызов *rm* вызовом команды *logrm*:

```
alias rm=logrm
```

В любом случае вам потребуются права доступа к каталогу */var/log* на выполнение и запись, что может не соответствовать настройкам системы по умолчанию.

## Результаты

Давайте создадим несколько файлов, удалим их и затем заглянем в журнал *remove.log*, как показано в листинге 2.11.

**Листинг 2.11.** Тестирование сценария `logrm`

```
$ touch unused.file ciao.c /tmp/junkit
$ logrm unused.file /tmp/junkit
$ logrm ciao.c
$ cat /var/log/remove.log
Thu Apr 6 11:32:05 MDT 2017: susan: /tmp/central.log
Fri Apr 7 14:25:11 MDT 2017: taylor: unused.file /tmp/junkit
Fri Apr 7 14:25:14 MDT 2017: taylor: ciao.c
```

Отлично! Обратите внимание, что пользователь `susan` удалил файл `/tmp/central.log` во вторник.

**Усовершенствование сценария**

В сценарии может возникнуть проблема с правами доступа к файлу журнала. Файл `remove.log` либо будет доступен всем для записи, и тогда любой пользователь сможет удалить его содержимое, например, командой `cat /dev/null > /var/log/remove.log`, или он вообще не будет доступен для записи, и тогда сценарий просто не станет журналировать события. Можно, конечно, попробовать установить привилегию `setuid`, чтобы сценарий запускался с правами суперпользователя `root`, открывающими доступ к файлу журнала. Но тут есть две проблемы. Во-первых, это очень плохая идея! Никогда не давайте сценариям привилегию `setuid`! Она позволяет выполнить команду с правами определенного пользователя, независимо от того, кто ее вызывает, что ухудшает безопасность системы. Во-вторых, можно оказаться в ситуации, когда пользователи имеют право удалять свои файлы, но сценарий не дает сделать этого, потому что действующий идентификатор пользователя, установленный привилегией `setuid`, будет унаследован командой `rm`, что нарушит ее работу. Может возникнуть большой конфуз, если обнаружится, что пользователи не имеют права удалять даже свои собственные файлы!

Для файловых систем `ext2`, `ext3` и `ext4` (используются по умолчанию в большинстве дистрибутивов Linux), существует другое решение — с помощью команды `chattr` установить на файл журнала специальное разрешение «только для добавления», что сделает его доступным для записи всем пользователям без всякой опасности. Еще одно решение: записывать сообщения в системный журнал с помощью замечательной команды `logger`. Журналирование операций с командой `rm` в этом случае будет выглядеть так:

```
logger -t logrm "${USER:-LOGNAME}: $*"
```

Эта команда добавит в поток данных системного журнала, недоступный рядовым пользователям для изменения, запись с меткой `logrm`, именем пользователя и выполненной командой.

**ПРИМЕЧАНИЕ**

Если вы решите использовать команду **logger**, прочитайте страницу справочного руководства **syslogd(8)**, где написано, как убедиться, что ваша конфигурация не отбрасывает события с приоритетом **user.notice**. Обычно эта настройка находится в файле */etc/syslogd.conf*.

## № 18. Вывод содержимого каталогов

Нам всегда казался бессмысленным один из аспектов команды **ls**: для каталогов она либо выводит список содержащихся в них файлов, либо показывает количество блоков по 1024 байта, необходимых для хранения данных. Ниже показано, как выглядит типичный элемент списка, возвращаемого командой **ls -l**:

```
drwxrwxr-x  2 taylor  taylor  4096 Oct 28 19:07 bin
```

Но в этой информации мало проку! В действительности нам хотелось бы знать, сколько файлов находится в каталоге. Именно это делает сценарий в листинге 2.12. Он генерирует многоколоночный список файлов и каталогов, показывая для файлов их размеры, а для каталогов — количество содержащихся в них записей.

### Код

**Листинг 2.12.** Сценарий **formatdir** для получения более информативных списков каталогов

```
#!/bin/bash

# formatdir -- выводит содержимое каталога в дружелюбном и информативном виде

# Обратите внимание: необходимо, чтобы "scriptbc" (сценарий № 9) находился
# в одном из каталогов, перечисленных в PATH, потому что он неоднократно
# вызывается в данном сценарии.

scriptbc=$(which scriptbc)

# Функция для преобразования размеров из KB в KB, MB или GB для
# большей удобочитаемости вывода
❶ readablesizе()
{
    if [ $1 -ge 1048576 ] ; then
        echo "$($scriptbc -p 2 $1 / 1048576)GB"
    elif [ $1 -ge 1024 ] ; then
        echo "$($scriptbc -p 2 $1 / 1024)MB"
    else
```



```

    echo "${1}KB"
fi
}

#####
## КОД ОСНОВНОГО СЦЕНАРИЯ

if [ $# -gt 1 ] ; then
    echo "Usage: $0 [dirname]" >&2
    exit 1
❷ elif [ $# -eq 1 ] ; then # Указан определенный каталог, не текущий?
    cd "$@"                # Тогда перейти в него.
    if [ $? -ne 0 ] ; then # Или выйти, если каталог не существует.
        exit 1
    fi
fi

for file in *
do
    if [ -d "$file" ] ; then
        size=$(ls "$file" | wc -l | sed 's/^[^:digit:]*//g')
        if [ $size -eq 1 ] ; then
            echo "$file ($size entry)|"
        else
            echo "$file ($size entries)|"
        fi
    else
        size="$(ls -sk "$file" | awk '{print $1}')"
        ❸ echo "$file ($(readablesize $size))|"
        fi
done | \
❹ sed 's/ /^^^/g' | \
    xargs -n 2 | \
    sed 's/\\^\\^\\^/ /g' | \
❺ awk -F\| '{ printf "%-39s %-39s\n", $1, $2 }'

exit 0

```

## Как это работает

Одним из наиболее интересных элементов сценария является функция `readablesize` ❶, которая принимает число в килобайтах и выводит килобайты, мегабайты или гигабайты, в зависимости от наиболее подходящей единицы измерения. Например, для файла очень большого размера она выведет 2.08GB вместо 2,083,364KB. Обратите внимание, что `readablesize` вызывается с применением конструкции `$( )` ❷:

```
echo "$file ($(readablesize $size))|"
```

Подоболочки автоматически наследуют все функции, объявленные в родительской оболочке, поэтому подоболочка, запущенная конструкцией `$()`, получит доступ к функции `readablesizes`. Очень удобно.

Ближе к началу сценария ❷ проверяется, был ли указан какой-то другой каталог, отличный от текущего, и затем производится смена текущего рабочего каталога выполняющегося сценария с помощью простой команды `cd`.

Основная логика сценария занимается организацией вывода в две колонки, выровненные по вертикали. Одна из проблем, возникающих при этом, состоит в том, что пробелы в потоке вывода нельзя просто заменить символами перевода строки, потому что имена файлов и каталогов сами могут содержать пробелы. Чтобы решить эту проблему, сценарий в ❸ сначала замещает каждый пробел последовательностью из трех «крышек» (`^^^`). Затем с помощью команды `xargs` объединяет строки попарно, чтобы каждая пара строк превратилась в одну, разделенную вертикальной чертой на два поля. Наконец, в ❹ вызывается команда `awk` для вывода полей с требуемым выравниванием.

Обратите внимание, как просто в ❺ подсчитывается количество (не скрытых) элементов внутри каталога с помощью команд `wc` и `sed`:

```
size=$(ls "$file" | wc -l | sed 's/^[^:digit:]*//g')
```

## Запуск сценария

Чтобы получить список содержимого сценария, запустите сценарий без аргументов, как показано в листинге 2.13. Чтобы получить информацию о другом каталоге, передайте имя этого каталога сценарию в виде единственного аргумента командной строки.

## Результаты

### Листинг 2.13. Тестирование сценария `formatdir`

```
$ formatdir ~
Applications (0 entries)      Classes (4KB)
DEMO (5 entries)             Desktop (8 entries)
Documents (38 entries)       Incomplete (9 entries)
IntermediateHTML (3 entries) Library (38 entries)
Movies (1 entry)             Music (1 entry)
NetInfo (9 entries)          Pictures (38 entries)
Public (1 entry)             RedHat 7.2 (2.08GB)
Shared (4 entries)           Synchronize! Volume ID (4KB)
X Desktop (4KB)              automatic-updates.txt (4KB)
bin (31 entries)             cal-liability.tar.gz (104KB)
cbhma.tar.gz (376KB)         errata (2 entries)
```

fire aliases (4KB)	games (3 entries)
junk (4KB)	leftside navbar (39 entries)
mail (2 entries)	perinatal.org (0 entries)
scripts.old (46 entries)	test.sh (4KB)
testfeatures.sh (4KB)	topcheck (3 entries)
tweakmktargs.c (4KB)	websites.tar.gz (18.85MB)

## Усовершенствование сценария

С данным сценарием может возникнуть проблема, если в системе имеется пользователь, обожающий последовательности из трех «крышек» в именах файлов. Конечно, это весьма маловероятно — из 116 696 файлов в нашей тестовой системе Linux не нашлось ни одного, имя которого содержало хотя бы один символ крышки, — но если такое случится, вывод сценария окажется испорченным. Если вас волнует эта проблема, попробуйте преобразовывать пробелы в другую последовательность символов, еще менее вероятную в именах файлов. Четыре «крышки»? Пять?

## № 19. Поиск файлов по именам

В системах Linux имеется очень практичная команда `locate`, которая не всегда присутствует в других разновидностях Unix. Эта команда выполняет поиск в предварительно созданной базе данных имен файлов по регулярному выражению, указанному пользователем. Нужно быстро найти мастер-файл `.cshrc`? Ниже показано, как это сделать с помощью `locate`:

```
$ locate .cshrc
/.Trashes/501/Previous Systems/private/etc/csh.cshrc
/OS9 Snapshot/Staging Archive:/home/taylor/.cshrc
/private/etc/csh.cshrc
/Users/taylor/.cshrc
/Volumes/110GB/WEBSITES/staging.intuitive.com/home/mdella/.cshrc
```

Как видите, в системе OS X мастер-файл `.cshrc` находится в каталоге `/private/etc`. Версия `locate`, которую мы напишем, будет просматривать все файлы на диске и конструировать их внутренний список для быстрого поиска, где бы они ни находились — в корзине, на отдельном томе. В списке окажутся даже скрытые файлы, имена которых начинаются с точки. Как вы вскоре поймете, это одновременно достоинство и недостаток новой команды.

## Код

Описываемый метод поиска файлов прост в реализации и предполагает создание двух сценариев. Первый (в листинге 2.14) создает базу данных всех имен

файлов, вызывая команду `find`, а второй (в листинге 2.15) — просто вызывает команду `grep` для поиска в новой базе данных.

#### Листинг 2.14. Сценарий `mklocatedb`

```
#!/bin/bash

# mklocatedb -- создает базу данных для locate с использованием find.
# Для запуска этого сценария пользователь должен обладать привилегиями
# суперпользователя root.

locatedb="/var/locate.db"

❶ if [ "$(whoami)" != "root" ] ; then
    echo "Must be root to run this command." >&2
    exit 1
fi

find / -print > $locatedb

exit 0
```

Второй сценарий еще короче.

#### Листинг 2.15. Сценарий `locate`

```
#!/bin/sh

# locate -- выполняет поиск в базе данных по заданному шаблону

locatedb="/var/locate.db"

exec grep -i "$@" $locatedb
```

## Как это работает

Сценарий `mklocatedb` должен запускаться с привилегиями суперпользователя `root`, чтобы он смог увидеть все файлы во всей системе, поэтому в строке ❶ он проверяет свои привилегии с помощью команды `whoami`. Однако запуск сценария с привилегиями `root` влечет за собой проблему безопасности, потому что, если каталог закрыт для рядовых пользователей, база данных `locate` не должна хранить информацию о нем или его содержимом. Эта проблема будет решена в главе 5, в новом, более безопасном сценарии `locate`, который учитывает правила защищенности и безопасности (сценарий № 39). А пока данный сценарий просто имитирует поведение стандартной команды `locate` из Linux, OS X и других дистрибутивов.

Не удивляйтесь, если сценарию `mklocatedb` потребуется несколько минут или больше; он выполняет обход всей файловой системы, что требует значительного

времени, даже для систем среднего размера. Результат также может получиться весьма впечатляющим. В одной из наших тестовых систем OS X файл *locate.db* содержал более 1,5 миллиона записей и занимал 1874,5 Мбайт дискового пространства.

После создания базы данных сам сценарий *locate* выглядит очень простым; он просто вызывает команду *grep* со всеми аргументами, полученными от пользователя.

## Запуск сценария

Прежде чем воспользоваться сценарием *locate*, необходимо запустить *mklocatedb*. Когда он завершит работу, вызов *locate* почти мгновенно будет находить совпадения в файловой системе с любыми заданными шаблонами.

## Результаты

Сценарий *mklocatedb* не принимает аргументов и ничего не выводит, как показано в листинге 2.16.

**Листинг 2.16.** Запуск сценария *mklocatedb* с помощью команды *sudo* для получения привилегий *root*

```
$ sudo mklocatedb
Password:
...
Много времени спустя
...
$
```

С помощью *ls* можно быстро узнать размер получившейся базы данных, как показано ниже:

```
$ ls -l /var/locate.db
-rw-r--r-- 1 root wheel 174088165 Mar 26 10:02 /var/locate.db
```

Теперь все готово к поиску файлов с помощью *locate*:

```
$ locate -i solitaire
/Users/taylor/Documents/AskDaveTaylor image folders/0-blog-pics/vista-search-solitaire.png
/Users/taylor/Documents/AskDaveTaylor image folders/8-blog-pics/windows-play-solitaire-1.png
/usr/share/emacs/22.1/lisp/play/solitaire.el.gz
/usr/share/emacs/22.1/lisp/play/solitaire.elc
```

```
/Volumes/MobileBackups/Backups.backupdb/Dave's MBP/2014-04-03-163622/BigHD/  
Users/taylor/Documents/AskDaveTaylor image folders/0-blog-pics/vista-search-  
solitaire.png  
/Volumes/MobileBackups/Backups.backupdb/Dave's MBP/2014-04-03-163622/BigHD/  
Users/taylor/Documents/AskDaveTaylor image folders/8-blog-pics/windows-play-  
solitaire-3.png
```

С помощью этого сценария можно извлекать другую интересную информацию о системе, например, количество файлов с исходным кодом на языке C:

```
$ locate '\.c$' | wc -l  
1479
```

---

## ПРИМЕЧАНИЕ

Обратите внимание на использованное здесь регулярное выражение. Команда **grep** требует экранировать символ точки (.), иначе она будет соответствовать любому одному символу. Кроме того, символ **\$** обозначает конец строки или, в данном случае, конец имени файла.

---

Приложив чуть больше усилий, мы могли бы передать каждый из найденных файлов команде **wc** и подсчитать общее количество строк исходного кода на языке C в системе, но это будет, пожалуй, перебор.

## Усовершенствование сценария

Чтобы обеспечить своевременное обновление базы данных, можно создать задание для **cron**, вызывающее **mklocatedb** в ночные часы раз в неделю, как это организовано в большинстве систем со встроенной командой **locate** или даже чаще, в зависимости от особенностей использования системы. Как и в случае с другими сценариями, действующими с привилегиями **root**, позаботьтесь о том, чтобы сделать сценарий недоступным для редактирования рядовым пользователям.

Еще одно усовершенствование, которое можно добавить в сценарий **locate**, — проверка и завершение с сообщением об ошибке при попытке запустить его без шаблона для поиска или в отсутствие файла базы данных **locate.db**. В текущей реализации сценарий просто выведет стандартное сообщение об ошибке от команды **grep**, которое может оказаться неинформативным для обычного пользователя. Еще более важной, как обсуждалось выше, является проблема безопасности: доступность рядовым пользователям имен всех файлов в системе, включая те, что должны быть скрыты от их глаз. Усовершенствования, касающиеся безопасности, мы добавим в сценарии № 39, в главе 5.

## № 20. Имитация других окружений: MS-DOS

Хотя в повседневной практике это едва ли понадобится, но с точки зрения освоения некоторых понятий командной оболочки будет интересно и показательно попробовать создать версии классических команд MS-DOS, таких как DIR, в виде сценариев, совместимых с Unix. Конечно, можно просто определить псевдоним и отобразить команду DIR в Unix-команду ls:

```
alias DIR=ls
```

Но такое отображение не имитирует фактического поведения команды; оно просто помогает забывчивым пользователям заучить новые названия команд. Если вам доводилось использовать древние способы взаимодействий с компьютером, вы наверняка вспомните, что флаг /W требует использовать широкий формат вывода. Но если передать флаг /W команде ls, она сообщит, что каталог /W не найден. Следующий сценарий DIR, представленный в листинге 2.17, напротив, написан так, что принимает и обрабатывает флаги, начинающиеся с символа слеша.

### Код

**Листинг 2.17.** Сценарий DIR, имитирующий DOS-команду DIR в Unix

```
#!/bin/bash
# DIR -- имитирует поведение команды DIR в DOS, принимает некоторые
# стандартные флаги команды DIR и выводит содержимое указанного каталога

function usage
{
cat << EOF >&2
Usage: $0 [DOS flags] directory or directories
Where:
    /D sort by columns
    /H show help for this shell script
    /N show long listing format with filenames on right
    /OD sort by oldest to newest
    /O-D sort by newest to oldest
    /P pause after each screenful of information
    /Q show owner of the file
    /S recursive listing
    /W use wide listing format
EOF
    exit 1
}

#####
```

```
### ОСНОВНОЙ СЦЕНАРИЙ
```

```
postcmd=""
flags=""

while [ $# -gt 0 ]
do
  case $1 in
    /D      ) flags="$flags -x" ;;
    /H      ) usage              ;;
    ❶ /[/NQW] ) flags="$flags -l" ;;
    /OD     ) flags="$flags -rt" ;;
    /O-D    ) flags="$flags -t"  ;;
    /P      ) postcmd="more"     ;;
    /S      ) flags="$flags -s"  ;;
    *       ) # Неизвестный флаг: возможно, признак конца команды DIR;
              # поэтому следует прервать цикл while.
  esac
  shift      # Флаг обработан; проверить -- есть ли что-то еще.
done

# Обработка флагов завершена; теперь выполнить саму команду:
if [ ! -z "$postcmd" ] ; then
  ls $flags "$@" | $postcmd
else
  ls $flags "$@"
fi

exit 0
```

## Как это работает

Этот сценарий демонстрирует, что инструкция `case` в языке командной оболочки фактически проверяет регулярное выражение. Как можно видеть в строке ❶, DOS-флаги `/N`, `/Q` и `/W` отображаются в один и тот же Unix-флаг `-l` в окончательном вызове команды `ls`, и все это достигается с помощью простого регулярного выражения `/[/NQW]`.

## Запуск сценария

Сохраните сценарий в файле с именем *DIR* (также желательно создать псевдоним `dir=DIR`, потому что командный интерпретатор DOS не различает регистр символов, в отличие от Unix). Теперь, вводя команду `DIR` с флагами, типичными для команды `DIR` в MS-DOS, пользователи будут получать осмысленные результаты (как показано в листинге 2.18), а не сообщение о том, что команда не найдена.



## Результаты

**Листинг 2.18.** Тестирование сценария DIR со списком файлов

```
$ DIR /OD /S ~/Desktop
```

```
total 48320
```

7720 PERP - Google SEO.pdf	28816 Thumbs.db
0 Traffic Data	8 desktop.ini
8 gofatherhood-com-crawllerrors.csv	80 change-lid-close-behavior-win7-1.png
16 top-100-errors.txt	176 change-lid-close-behavior-win7-2.png
0 \$RECYCLE.BIN	400 change-lid-close-behavior-win7-3.png
0 Drive Sunshine	264 change-lid-close-behavior-win7-4.png
96 facebook-forcing-pay.jpg	32 change-lid-close-behavior-win7-5.png
10704 WCSS Source Files	

Это список с содержимым указанного каталога, отсортированный в обратном хронологическом порядке, от более новых к более старым, и размерами файлов (для каталогов всегда выводится размер 0).

## Усовершенствование сценария

В наши дни трудно найти человека, который помнил бы командную строку MS-DOS, но основные принципы работы с ней стоят того, чтобы их знать. Как одно из усовершенствований можно было бы реализовать вывод эквивалентной команды в Unix или Linux перед фактическим выполнением, и затем, после нескольких вызовов, сценарий мог бы просто показывать эквивалентную команду, но не выполнять ее. В этом случае пользователь будет вынужден запоминать новые команды, чтобы добиться желаемого!

## № 21. Вывод времени в разных часовых поясах

Основное требование, предъявляемое к команде `date`, — отображение даты и времени для часового пояса, настроенного в системе. Но как быть пользователям в дальней поездке, пересекающим несколько часовых поясов? Или тем, у кого есть друзья и коллеги, живущие в других уголках планеты, и им хотелось бы знать, который сейчас час, например, в Касабланке, Ватикане или Сиднее?

Как оказывается, команда `date` в большинстве современных разновидностей Unix опирается в своей работе на базу данных часовых поясов. Обычно хранящаяся в каталоге `/usr/share/zoneinfo` эта база данных содержит информацию о более чем 600 регионах и соответствующих им смещениях относительно универсального скоординированного времени (Universal Coordinated Time, UTC — часто также называется *средним временем по Гринвичу*, *Greenwich Mean Time* или *GMT*).

Команда `date` учитывает значение переменной окружения `TZ`, определяющей часовой пояс, которой можно присвоить любой регион из базы данных, например:

```
$ TZ="Africa/Casablanca" date
Fri Apr 7 16:31:01 WEST 2017
```

Однако большинству пользователей неудобно временно подменять значения переменных окружения. Написав сценарий командной оболочки, можно реализовать более дружелюбный интерфейс к базе данных часовых поясов.

Большая часть сценария в листинге 2.19 связана с базой данных часовых поясов (которая обычно хранится в виде нескольких файлов в каталоге *zonedir*), точнее, с попыткой найти файл, соответствующий указанному шаблону. После обнаружения файла сценарий устанавливает найденный часовой пояс как текущий (в виде `TZ="Africa/Casablanca"` в данном примере) и с этими настройками вызывает команду `date` в подоболочке. Команда `date` определит часовой пояс по значению переменной `TZ`, и ей совершенно безразлично, хранит ли она временное значение или это тот часовой пояс, в котором вы проводите большую часть времени.

## Код

**Листинг 2.19.** Сценарий `timein` для вывода времени в определенном часовом поясе

```
#!/bin/bash

# timein -- выводит текущее время в указанном часовом поясе или
# географической области. При вызове без аргументов выводит время
# UTC/GMT. Используйте слово "list", чтобы вывести список всех известных
# географических областей.
# Обратите внимание, что сценарий может находить совпадения с каталогами
# часовых поясов (областей), но действительными спецификациями являются
# только файлы (города).

# Ссылка на базу данных часовых поясов: http://www.twinsun.com/tz/tz-link.htm

zonedir="/usr/share/zoneinfo"

if [ ! -d $zonedir ] ; then
    echo "No time zone database at $zonedir." >&2
    exit 1
fi

if [ -d "$zonedir/posix" ] ; then
    zonedir=$zonedir/posix # Modern Linux systems
fi

if [ $# -eq 0 ] ; then
    timezone="UTC"
    mixedzone="UTC"
```

```

❶ elif [ "$1" = "list" ] ; then
    ( echo "All known time zones and regions defined on this system:"
      cd $zonedir
      find -L * -type f -print | xargs -n 2 | \
        awk '{ printf " %-38s %-38s\n", $1, $2 }'
    ) | more
    exit 0
else

    region="$(dirname $1)"
    zone="$(basename $1)"

    # Заданный часовой пояс имеет прямое соответствие? Если да, можно продолжать.
    # Иначе следует продолжить поиск. Для начала подсчитать совпадения.

    matchcnt="$(find -L $zonedir -name $zone -type f -print |
      wc -l | sed 's/^[^:digit:]*//g' )"

    # Проверить наличие хотя бы одного совпадения.
    if [ "$matchcnt" -gt 0 ] ; then
        # И выйти, если совпадений несколько.
        if [ $matchcnt -gt 1 ] ; then
            echo "\"$zone\" matches more than one possible time zone record." >&2
            echo "Please use 'list' to see all known regions and time zones." >&2
            exit 1
        fi
        match="$(find -L $zonedir -name $zone -type f -print)"
        mixedzone="$zone"
    else # Может быть, удастся найти совпадение с регионом, а не
        # с конкретным часовым поясом.
        # Первый символ в названии области/пояса преобразовать в верхний
        # регистр, остальные -- в нижний
        mixedregion="$(echo ${region%${region#?}} \
          | tr '[:lower:]' '[:upper:]')\
          $(echo ${region#?} | tr '[:upper:]' '[:lower:]'))"
        mixedzone="$(echo ${zone%${zone#?}} | tr '[:lower:]' '[:upper:]') \
          $(echo ${zone#?} | tr '[:upper:]' '[:lower:]'))"

        if [ "$mixedregion" != "." ] ; then
            # Искать только указанный часовой пояс в заданной области,
            # чтобы позволить пользователям указывать уникальные пары, когда
            # возможны другие варианты (например, "Atlantic").
            match="$(find -L $zonedir/$mixedregion -type f -name $mixedzone -print)"
        else
            match="$(find -L $zonedir -name $mixedzone -type f -print)"
        fi

        # Если найден файл, точно соответствующий заданному шаблону
        if [ -z "$match" ] ; then
            # Проверить, не является ли шаблон слишком неоднозначным.
            if [ ! -z $(find -L $zonedir -name $mixedzone -type d -print) ] ; then
                echo "The region \"$1\" has more than one time zone." >&2
            else # Или полное отсутствие совпадений

```

❷

```

        echo "Can't find an exact match for \"${1}\". " >&2
    fi
    echo "Please use 'list' to see all known regions and time zones." >&2
    exit 1
fi
fi
fi
❷    timezone="${match}"
fi

nicetz=$(echo $timezone | sed "s|${zonedir}/||g") # Отформатировать вывод.

echo It\'s $(TZ=$timezone date '+%A, %B %e, %Y, at %l:%M %p') in $nicetz

exit 0

```

## Как это работает

Этот сценарий использует способность команды `date` выводить дату и время для указанного часового пояса независимо от текущих настроек окружения. Фактически, весь сценарий решает задачу идентификации часового пояса, чтобы вызов команды `date` в самом конце выполнялся без ошибок.

В основном сложность данного сценария обусловлена желанием определить часовой пояс по введенному пользователем названию области, для которого не найдено прямого совпадения в базе данных часовых поясов. Данные хранятся в ней в виде столбцов *timzonename* и *region/locationname*, и сценарий старается отобразить полезные сообщения об ошибках для наиболее типичных проблем, связанных с вводом, например, когда часовой пояс не может быть определен, потому что пользователь указал страну, которая делится на несколько часовых поясов (например, Бразилию).

Даже при том, что присваивание `TZ="Casablanca"` приводит к неудаче поиска географической области, город Casablanca (Касабланка) действительно существует в базе данных. Проблема в том, что для успешного определения часового пояса необходимо использовать правильное сочетание названия области и города *Africa/Casablanca*, как было показано во введении к этому сценарию.

С другой стороны, данный сценарий способен самостоятельно найти файл *Casablanca* в каталоге *Africa* и точно определить часовой пояс. Но одной только области *Africa* будет недостаточно, потому что сценарий найдет несколько подобластей в каталоге *Africa* и выведет сообщение, указывающее, что предоставленной информации недостаточно для уникальной идентификации часового пояса ❷. Можно также воспользоваться полным списком всех часовых поясов ❶ или передать сценарию точное название часового пояса ❸ (например, UTC или WET).

## ПРИМЕЧАНИЕ

---

Отличный справочник по часовым поясам можно найти по адресу: <http://www.twinsun.com/tz/tz-link.htm>.

---

## Запуск сценария

Чтобы узнать текущее время в географической области или в городе, передайте сценарию `timein` аргумент с названием области или города. Если вы знаете и область, и город, передайте их в формате *region/city* (например, `Pacific/Honolulu`). При вызове без аргументов сценарий `timein` выведет время UTC/GMT. В листинге 2.20 показаны примеры вызова сценария `timein` с разными часовыми поясами.

## Результаты

**Листинг 2.20.** Тестирование сценария `timein` с разными часовыми поясами

```
$ timein
It's Wednesday, April 5, 2017, at 4:00 PM in UTC
$ timein London
It's Wednesday, April 5, 2017, at 5:00 PM in Europe/London
$ timein Brazil
The region "Brazil" has more than one time zone. Please use 'list'
to see all known regions and time zones.
$ timein Pacific/Honolulu
It's Wednesday, April 5, 2017, at 6:00 AM in Pacific/Honolulu
$ timein WET
It's Wednesday, April 5, 2017, at 5:00 PM in WET
$ timein mycloset
Can't find an exact match for "mycloset". Please use 'list'
to see all known regions and time zones.
```

## Усовершенствование сценария

Возможность узнать время в любом часовом поясе по всему миру очень полезна, особенно для администраторов, управляющих глобальными сетями. Но иногда требуется всего лишь узнать *разницу* во времени между двумя часовыми поясами. Эту функциональность можно было бы добавить в сценарий `timein`. Или же написать новый сценарий, например, с именем `tzdiff`, использующий `timein`, который принимает два аргумента вместо одного.

Задействуя оба аргумента, сценарий мог бы определять текущее время в обоих часовых поясах и затем выводить разницу между ними. Но имейте в виду, что двухчасовая разница между двумя часовыми поясами может быть на два часа *вперед* или на два часа *назад*. Различать два этих случая особенно важно для создания по-настоящему полезного сценария.

## Глава 3. Создание утилит

Одна из основных целей создания сценариев командной оболочки — перенести сложные команды в файл, где их легко воспроизвести и изменить. Поэтому неудивительно, что на протяжении всей книги рассматриваются пользовательские команды. Но удивительно, что нам не требуется писать обертки для каждой отдельной команды в системах Linux, Solaris и OS X.

Linux/Unix — единственная из основных операционных систем, где можно решить, что флаги по умолчанию не отвечают вашим потребностям, и исправить положение несколькими нажатиями клавиш или симитировать поведение понравившейся утилиты из другой операционной системы, определив псевдоним или написав сценарий длиной в десяток строк. Именно это делает систему Unix такой дружелюбной, и именно это вдохновило нас написать книгу, которую вы держите в руках!

### № 22. Утилита для напоминания

В распоряжении пользователей Windows и Mac уже много лет имеются превосходные и простые утилиты, такие как Stickies, позволяющие сохранять короткие заметки и выводить напоминания на экран. Они прекрасно подходят для быстрой записи телефонных номеров или другой информации. К сожалению, в командной строке Unix нет аналогичной программы для создания заметок, но эту проблему легко решить парой сценариев.

Первый сценарий, `remember` (приводится в листинге 3.1), позволяет сохранить заметку в общем файле `rememberfile` в домашнем каталоге. Если вызвать этот сценарий без аргументов, он будет читать стандартный ввод, пока не встретит символ конца файла (^D), который вводится комбинацией `ctrl-D`. Если вызвать сценарий с аргументами, он запишет их прямо в файл с данными.

Вторая половина описываемой двоицы — `remindme`, сопутствующий сценарий, представленный в листинге 3.2, который либо выводит все содержимое файла `rememberfile`, когда запускается без аргументов, либо отображает результаты поиска, используя аргументы как шаблон.

## Код

### Листинг 3.1. Сценарий remember

```
#!/bin/bash

# remember -- простой блокнот для записи заметок из командной строки

rememberfile="$HOME/.remember"

if [ $# -eq 0 ] ; then
    # Предложить пользователю ввести заметку и добавить ее в конец
    # файла rememberfile.
    echo "Enter note, end with ^D: "
    ❶ cat - >> $rememberfile
    else
        # Записать в конец файла .remember все полученные аргументы.
    ❷ echo "$@" >> $rememberfile
fi

exit 0
```

В листинге 3.2 приводится сопутствующий сценарий remindme.

### Листинг 3.2. Сценарий remindme, сопутствующий сценарию remember из листинга 3.1

```
#!/bin/bash

# remindme -- ищет в файле с данными совпадения с заданным шаблоном или, если
# запускается без аргументов, выводит все содержимое файла

rememberfile="$HOME/.remember"

if [ ! -f $rememberfile ] ; then
    echo "$0: You don't seem to have a .remember file. " >&2
    echo "To remedy this, please use 'remember' to add reminders" >&2
    exit 1
fi

if [ $# -eq 0 ] ; then
    # Вывести все содержимое rememberfile, если критерии поиска не заданы.
    ❸ more $rememberfile
else
    # Иначе выполнить поиск в файле по заданному критерию и вывести
    # результаты.
    ❹ grep -i -- "$@" $rememberfile | ${PAGER:-more}
fi

exit 0
```

## Как это работает

Сценарий `remember` в листинге 3.1 может действовать как интерактивная программа, предлагающая пользователю ввести текст заметки для запоминания, или как команда, сохраняющая свои аргументы командной строки. На случай, если пользователь запустит сценарий без аргументов, мы предусмотрели одну хитрость. После вывода сообщения с предложением ввести заметку, мы вызываем команду `cat`, чтобы прочитать ввод пользователя ❶:

```
cat - >> $rememberfile
```

В предыдущих главах нам доводилось использовать команду `read`, чтобы получить ввод пользователя. Здесь же команда `cat` читает текст из `stdin` (дефис - в команде является коротким обозначением `stdin` или `stdout`, в зависимости от контекста), пока пользователь не нажмет комбинацию `ctrl-D`, которая сообщит утилите `cat` о завершении файла. После этого `cat` выведет текст, прочитанный из `stdin`, и добавит его в конец файла `rememberfile`.

Однако, если сценарий запустить с аргументами, он просто добавит их все в конец `rememberfile` ❷.

Сценарий `remindme` в листинге 3.2 не может работать в отсутствие файла `rememberfile`, поэтому в самом начале, перед попыткой что-либо сделать, он проверяет его наличие. Если файл отсутствует, сценарий завершается с выводом сообщения о причине остановки.

Если сценарий запущен без аргументов, предполагается, что пользователь просто захотел увидеть содержимое `rememberfile`. Использование утилиты `more` позволяет организовать постраничный просмотр файла `rememberfile` ❸.

Если сценарий запущен с аргументами, вызывается утилита `grep`, чтобы найти совпадения с указанным шаблоном в `rememberfile` без учета регистра символов, а затем результаты выводятся с помощью утилиты постраничного просмотра ❹.

## Запуск сценария

Чтобы воспользоваться утилитой `remindme`, сначала нужно добавить несколько заметок в файл `rememberfile`, запустив сценарий `remember`, как показано в листинге 3.3. После этого можно с помощью `remindme` выполнить поиск в получившейся базе данных, передав сценарию искомый шаблон.



## Результаты

### Листинг 3.3. Тестирование сценария remember

```
$ remember Southwest Airlines: 800-IFLYSWA
$ remember
Enter note, end with ^D:
Find Dave's film reviews at http://www.DaveOnFilm.com/
^D
```

Затем, когда спустя несколько месяцев вам потребуется вспомнить текст заметки, вы сможете сделать это с помощью `reminder`, как показано в листинге 3.4.

### Листинг 3.4. Тестирование сценария remindme

```
$ remindme film reviews
Find Dave's film reviews at http://www.DaveOnFilm.com/
```

Или, если вы не можете быстро вспомнить номер телефона, из которого известны только цифры 800, листинг 3.5 демонстрирует, как выполнить поиск по частично известному номеру.

### Листинг 3.5. Поиск номера телефона по известной последовательности цифр с помощью сценария remindme

```
$ remindme 800
Southwest Airlines: 800-IFLYSWA
```

## Усовершенствование сценария

Конечно, не каждый сценарий демонстрирует чудеса программирования, но эти два сценария наглядно показывают, насколько легко расширить возможности командной строки Unix. Чтобы вы себе ни вообразили, наверняка найдется простой способ реализовать это.

В рассмотренные сценарии можно внести много разных усовершенствований. Например, ввести понятие *записей*: сценарий `remember` снабжает каждую запись датой и временем, многострочный текст сохраняется как одна запись, а поиск выполняется с использованием регулярных выражений. Такой подход позволит сохранять телефонные номера для групп людей и получать их, помня имя хотя бы одного члена группы. Если вы действительно задумаетесь над усовершенствованием сценария, можете добавить также функцию редактирования и удаления записей. Хотя, с другой стороны, файл `~/remember` легко отредактировать с помощью любого текстового редактора.

## № 23. Интерактивный калькулятор

Если вы помните, `scriptbc` (сценарий № 9 в главе 1) позволял вызывать калькулятор `bc` для вычисления выражений, передаваемых в виде аргументов командной строки. Следующий логичный шаг — написать сценарий-обертку, превращающую сценарий `scriptbc` в интерактивный калькулятор командной строки. Сценарий (приводится в листинге 3.6) получился действительно очень коротким! Но чтобы он заработал, не забудьте поместить сценарий `scriptbc` в один из каталогов из списка `PATH`.

### Код

**Листинг 3.6.** Сценарий калькулятора командной строки `calc`

```
#!/bin/bash

# calc -- калькулятор командной строки, который действует как интерфейс к bc

scale=2

show_help()
{
    cat << EOF
        In addition to standard math functions, calc also supports:

        a % b    remainder of a/b
        a ^ b    exponential: a raised to the b power
        s(x)     sine of x, x in radians
        c(x)     cosine of x, x in radians
        a(x)     arctangent of x, in radians
        l(x)     natural log of x
        e(x)     exponential log of raising e to the x
        j(n,x)   Bessel function of integer order n of x
        scale N  show N fractional digits (default = 2)
    EOF
}

if [ $# -gt 0 ] ; then
    exec scriptbc "$@"
fi

echo "Calc--a simple calculator. Enter 'help' for help, 'quit' to quit."

/bin/echo -n "calc> "

❶ while read command args
do
    case $command
```

```

in
    quit|exit) exit 0 ;;
    help|\?) show_help ;;
    scale) scale=$args ;;
    *) scriptbc -p $scale "$command" "$args" ;;
esac

/bin/echo -n "calc> "
done

echo ""

exit 0

```

## Как это работает

Самая интересная часть в этом сценарии — инструкция `while read` ❶, которая образует бесконечный цикл, отображающий приглашение `calc>`, пока пользователь не завершит работу вводом команды `quit` или признака конца файла (^D). Лаконичность сценария делает его особенно примечательным: сценарии командной строки должны быть простыми и практичными!

## Запуск сценария

Сценарий использует `scriptbc`, калькулятор, который мы написали в сценарии № 9, поэтому, прежде чем запускать его, не забудьте поместить `scriptbc` в один из каталогов, перечисленных в списке `PATH` (или добавьте в сценарий переменную, например `$scriptbc`, содержащую полный путь к сценарию). По умолчанию данный сценарий выполняется в интерактивном режиме, предлагая пользователю вводить выражения для вычисления. Если запустить его с аргументами, эти аргументы будут переданы непосредственно сценарию `scriptbc`. В листинге 3.7 показаны оба способа использования сценария.

## Результаты

### Листинг 3.7. Тестирование сценария `calc`

```

$ calc 150 / 3.5
42.85
$ calc
Calc -- a simple calculator. Enter 'help' for help, 'quit' to quit.
calc> help
    In addition to standard math functions, calc also supports:

    a % b      remainder of a/b
    a ^ b      exponential: a raised to the b power

```

```

s(x)      sine of x, x in radians
c(x)      cosine of x, x in radians
a(x)      arctangent of x, in radians
l(x)      natural log of x
e(x)      exponential log of raising e to the x
j(n,x)    Bessel function of integer order n of x
scale N   show N fractional digits (default = 2)
calc> 54354 ^ 3
160581137553864
calc> quit
$

```

## ВНИМАНИЕ

Вычисления с вещественными числами, даже простые для человека, могут быть сложными для компьютеров. К сожалению, команда **bc** иногда реагирует на такие сложности самым неожиданным образом. Например, запустите **bc** и введите **scale=0** и затем **7 % 3**. А теперь попробуйте вычислить то же выражение с **scale=4**. В результате вы получите **.0001**, что, очевидно, является ошибкой.

## Усовершенствование сценария

Все, что можно сделать в **bc**, можно сделать и в этом сценарии, с той лишь разницей, что **calc** не имеет памяти команд или состояний. Попробуйте добавить больше математических функций в справочное сообщение. Например, переменные **obase** и **ibase** позволяют определить основание системы счисления для вывода и ввода, однако из-за того, что сценарий не имеет памяти команд, вам придется изменить **scriptbc** (сценарий № 9 в главе 1) или научиться вводить настройки и выражения в одной строке.

## № 24. Преобразование температур

Сценарий в листинге 3.8 — первый в книге, выполняющий сложные математические вычисления, — может преобразовывать значение температуры в градусы Фаренгейта, Цельсия и Кельвина. В нем используется тот же трюк передачи выражений для вычисления калькулятору **bc**, что и в сценарии № 9, в главе 1.

## Код

### Листинг 3.8. Сценарий **convertatemp**

```

#!/bin/bash

# convertatemp -- сценарий преобразования температуры, позволяющий вводить
# температуру в градусах Фаренгейта, Цельсия или Кельвина и получать

```

```

# эквивалентную температуру в двух других шкалах

if [ $# -eq 0 ] ; then
    cat << EOF >&2
Usage: $0 temperature[F|C|K]
where the suffix:
    F      indicates input is in Fahrenheit (default)
    C      indicates input is in Celsius
    K      indicates input is in Kelvin
EOF
    exit 1
fi
❶ unit="$(echo $1|sed -e 's/[-[:digit:]]*/g' | tr '[:lower:]' '[:upper:]' )"
❷ temp="$(echo $1|sed -e 's/^[^[:digit:]]*/g' )"

case ${unit:=F}
in
F ) # Градусы Фаренгейта в градусы Цельсия:  $T_c = (F - 32) / 1.8$ 
    farn="$temp"
❸ cels="$(echo "scale=2;($farn - 32) / 1.8" | bc)"
    kelv="$(echo "scale=2;$cels + 273.15" | bc)"
    ;;

C ) # Градусы Цельсия в градусы Фаренгейта:  $T_f = (9/5)*T_c+32$ 
    cels=$temp
    kelv="$(echo "scale=2;$cels + 273.15" | bc)"
❹ farn="$(echo "scale=2;(1.8 * $cels) + 32" | bc)"
    ;;

❺ K ) # Градусы Цельсия = Kelvin - 273.15,
    # затем использовать формулу градусы Цельсия -> градусы Фаренгейта
    kelv=$temp
    cels="$(echo "scale=2; $kelv - 273.15" | bc)"
    farn="$(echo "scale=2; (1.8 * $cels) + 32" | bc)"
    ;;

*)
    echo "Given temperature unit is not supported"
    exit 1
esac

echo "Fahrenheit = $farn"
echo "Celsius = $cels"
echo "Kelvin = $kelv"

exit 0

```

## Как это работает

Большая часть сценария, вероятно, ясна, но давайте внимательнее рассмотрим математические вычисления и регулярные выражения, выполняющие основную работу. Многие плохо воспринимают математические формулы в таком

виде, поэтому ниже приводится формула преобразования температуры по Фаренгейту в температуру по Цельсию:

$$C = \frac{(F - 32)}{1,8}.$$

Преобразованную в последовательность для передачи калькулятору `bc` и вычисления, эту формулу можно видеть в строке ❸. Обратное преобразование из градусов Цельсия в градусы Фаренгейта реализовано в строке ❹. Этот сценарий также переводит температуру из градусов Цельсия в градусы Кельвина ❺. Он наглядно демонстрирует одну важную причину использовать мнемонические имена для переменных: код становится проще для чтения и отладки.

Еще один интересный аспект сценария — регулярные выражения, наиболее замысловатое из которых находится в строке ❶. Понять эту строку проще, если развернуть операцию подстановки, выполняемую `sed`. Подстановка всегда имеет вид `s/old/new/`; в данном случае шаблон *old* описывает строку, начинающуюся с нуля или более дефисов (-), за которыми следует любое количество цифр (как вы помните, `[:digit:]` — это форма записи класса символов в ANSI, представляющего собой произвольную цифру, а звездочка (\*) обозначает ноль или более вхождений предыдущего шаблона). Шаблон *new* описывает, чем заменить совпадение с шаблоном *old*, и в данном случае это всего лишь `//`, то есть пустой шаблон. Его удобно использовать, когда требуется просто удалить совпадения с шаблоном *old*. Данная операция подстановки фактически удаляет все цифры и дефисы так, что ввод `-31f` превращается в `f` и мы получаем возможность определить шкалу измерения температуры. После этого команда `tr` нормализует результат, преобразуя его в верхний регистр, то есть строка `-31f`, например, превращается в `F`.

Другое выражение `sed` выполняет противоположную операцию ❷: оно удаляет все, что не является частью числа, используя оператор `^` для инвертирования совпадения с любым символом в классе `[:digit:]`. (В большинстве языков программирования инвертирование выполняет оператор `!`.) В результате получается значение для преобразования с применением соответствующей формулы.

## Запуск сценария

Сценарий имеет простой и понятный формат входных данных, хотя и необычный для команд Unix. Сценарию передается числовое значение с обязательным символом в конце, обозначающим шкалу; в отсутствие этого символа предполагается, что значение температуры представлено в градусах Фаренгейта.

Чтобы узнать температуру в градусах Цельсия и Кельвина, эквивалентную 0° Фаренгейта, введите `0F`. Чтобы узнать температуру в градусах Цельсия и Фаренгейта, эквивалентную 100° Кельвина, введите `100K`. А чтобы узнать температуру в градусах Кельвина и Фаренгейта, эквивалентную 100° Цельсия, введите `100C`.

Похожий прием использования односимвольного обозначения в конце мы увидим в главе 7, в сценарии № 60, который выполняет преобразования между валютами.

## Результаты

В листинге 3.9 показано несколько примеров преобразования температур.

**Листинг 3.9.** Тестирование сценария `convertatemp` несколькими преобразованиями

```
$ convertatemp 212
Fahrenheit = 212
Celsius = 100.00
Kelvin = 373.15
$ convertatemp 100C
Fahrenheit = 212.00
Celsius = 100
Kelvin = 373.15
$ convertatemp 100K
Fahrenheit = -279.67
Celsius = -173.15
Kelvin = 100
```

## Усовершенствование сценария

В сценарий можно добавить поддержку нескольких флагов, чтобы ограничить вывод единственным результатом. Например, команда `convertatemp -c 100F` выводила бы только значение в градусах Цельсия, эквивалентное 100° Фаренгейта. Это помогло бы также упростить использование данного сценария внутри других.

## № 25. Вычисление платежей по кредиту

Другой распространенный вид вычислений, который наверняка пригодится пользователям — оценка платежей по кредиту. Сценарий в листинге 3.10 помогает также ответить на вопрос: «Куда потратить премию?», — и еще один, связанный с ним: «Могу ли я наконец позволить себе купить новую Tesla?».

Формула вычисления платежей, основанная на сумме кредита, процентах и его продолжительности, выглядит непростой, тем не менее грамотное использование переменных может помочь обуздать этого математического зверя и сделать вычисления на удивление простыми и понятными.

## Код

### Листинг 3.10. Сценарий loancalc

```
#!/bin/bash

# loancalc -- По заданной сумме кредита, процентной ставке
# и продолжительности (в годах), вычисляет суммы платежей

# Формула:  $M = P * (J / (1 - (1 + J)^{-N}))$ ,
# где P = сумма кредита, J = месячная процентная ставка, N = протяженность
# (месяцев).

# Обычно пользователи вводят P, I (годовая процентная ставка) и L (протяженность
# в годах).

❶ . library.sh # Подключить библиотечный сценарий.

if [ $# -ne 3 ] ; then
    echo "Usage: $0 principal interest loan-duration-years" >&2
    exit 1
fi

❷ P=$1 I=$2 L=$3
J="$(scriptbc -p 8 $I / \( 12 \* 100 \) )"
N="$(( $L * 12 ))"
M="$(scriptbc -p 8 $P \* \( $J / \(1 - \(1 + $J\) \^ -$N\) \) )"

# Выполнить необходимые преобразования значений:
❸ dollars="$(echo $M | cut -d. -f1)"
cents="$(echo $M | cut -d. -f2 | cut -c1-2)"

cat << EOF
A $L-year loan at $I% interest with a principal amount of $(nicenumber $P 1 )
results in a payment of \$$dollars.$cents each month for the duration of
the loan ($N payments).
EOF

exit 0
```

## Как это работает

Рассмотрение самих вычислений выходит за рамки этой книги, но обратите внимание, как сложную математическую формулу можно реализовать непосредственно в сценарии командной оболочки.



Другой способ выполнить все вычисления — передать один большой поток входных данных программе `bc`, потому что она поддерживает переменные. Однако возможность манипулировать промежуточными значениями внутри самого сценария доказывает, что он позволяет произвести часть вычислений без привлечения команды `bc`. Кроме того, деление формулы на несколько промежуточных вычислений ❷ упрощает отладку. Например, следующий код разбивает вычисленные месячные платежи на доллары и центы и гарантирует правильное форматирование денежных сумм:

```
dollars="$(echo $M | cut -d. -f1)"
cents="$(echo $M | cut -d. -f2 | cut -c1-2)"
```

Команда `cut` оказывается здесь особенно полезной ❸. Вторая строка в этом коде извлекает из суммы месячного платежа ту часть, которая следует за десятичной точкой, и затем отсекает все, что следует за вторым символом. Если вы пожелаете округлить число до центов в большую сторону, просто прибавьте 0,005 к результату вычислений перед усечением центов до двух цифр.

Обратите внимание, как в строке ❶ командой `. library.sh` подключается библиотечный сценарий, созданный в главе 1, что обеспечивает доступность всех функций (в данном сценарии используется функция `nicenumber()` из главы 1).

## Запуск сценария

Этот коротенький сценарий принимает три параметра: сумма кредита, процентная ставка и срок кредита (в годах).

## Результаты

Представьте, что вы узнали о выходе новой модели Tesla Model S и вам интересно узнать, сколько придется заплатить, если купить ее в кредит. Стоимость модели Model S начинается примерно с 69 900 долларов, а ставка по кредиту составляет 4,75% годовых. Допустим, что у вас уже есть автомобиль, за который вы выручите 25 000 долларов на вторичном рынке, и вам остается добавить 44 900. Недолго думая, вы можете сравнить суммы выплат по четырех- и пятилетнему автокредиту, просто воспользовавшись сценарием, показанным в листинге 3.11.

### Листинг 3.11. Тестирование сценария `loancalc`

```
$ loancalc 44900 4.75 4
```

```
A 4-year loan at 4.75% interest with a principal amount of 44,900
results in a payment of $1028.93 each month for the duration of
the loan (48 payments).
```

```
$ loancalc 44900 4.75 5
```

A 5-year loan at 4.75% interest with a principal amount of 44,900 results in a payment of \$842.18 each month for the duration of the loan (60 payments).

Если вы в состоянии потянуть выплаты по четырехлетнему автокредиту, вы погасите его быстрее, и общая сумма выплат (произведение суммы месячного платежа на количество месяцев) значительно уменьшится. Чтобы подсчитать экономию, можно воспользоваться интерактивным калькулятором из сценария № 23, как показано ниже:

```
$ calc '(842.18 * 60) - (1028.93 * 48)'
1142.16
```

1142,16 доллара — хорошая экономия, этих денег хватит на отличный ноутбук!

## Усовершенствование сценария

Этот сценарий мог бы запрашивать необходимые данные при запуске без параметров. Еще более полезная версия сценария могла бы предлагать пользователю ввести *любые* три параметра из четырех (сумма кредита, процентная ставка, срок и сумма месячных платежей) и автоматически вычислять четвертое значение. В этом случае, зная, что вы способны выплачивать только 500 долларов в месяц и максимальная ставка по пятилетнему автокредиту составляет 6%, вы сумели бы определить максимальную сумму доступного для вас кредита. Подобные вычисления можно выполнять, реализовав поддержку разных флагов, которые пользователи передавали бы сценарию.

## № 26. Слежение за событиями

Следующая пара сценариев реализует простую программу-календарь, похожую на утилиту напоминания из сценария № 22. Первый сценарий, `addagenda` (представлен в листинге 3.12), позволяет определить событие, повторяющееся (в определенные дни недели, месяца или года) или однократное (в конкретный день, месяц и год). Все даты проверяются и сохраняются вместе с однострочным описанием события в файле `.agenda`, в домашнем каталоге пользователя. Второй сценарий, `agenda` (представлен в листинге 3.13), просматривает все сохраненные события и отыскивает запланированные на текущую дату.

Этот инструмент особенно удобно использовать для запоминания дней рождений и годовщин. Если вы забываете про важные события, приведенная ниже пара сценариев поможет вам избежать конфуза!

## Код

### Листинг 3.12. Сценарий addagenda

```
#!/bin/bash

# addagenda -- предлагает пользователю добавить новое событие для сценария
agenda

agendafile="$HOME/.agenda"

isDayName()
{
    # Возвращает 0, если все в порядке, 1 -- в случае ошибки.
    case $(echo $1 | tr '[:upper:]' '[:lower:]') in
        sun*|mon*|tue*|wed*|thu*|fri*|sat*) retval=0 ;;
        * ) retval=1 ;;
    esac
    return $retval
}

isMonthName()
{
    case $(echo $1 | tr '[:upper:]' '[:lower:]') in
        jan*|feb*|mar*|apr*|may|jun*) return 0 ;;
        jul*|aug*|sep*|oct*|nov*|dec*) return 0 ;;
        * ) return 1 ;;
    esac
}

❶ normalize()
{
    # Возвращает строку с первым символом в верхнем регистре
    # и другими двумя -- в нижнем.
    /bin/echo -n $1 | cut -c1 | tr '[:lower:]' '[:upper:]'
    echo $1 | cut -c2-3 | tr '[:upper:]' '[:lower:]'
}

if [ ! -w $HOME ] ; then
    echo "$0: cannot write in your home directory ($HOME)" >&2
    exit 1
fi

echo "Agenda: The Unix Reminder Service"
/bin/echo -n "Date of event (day mon, day month year, or dayname): "
read word1 word2 word3 junk

if isDayName $word1 ; then
    if [ ! -z "$word2" ] ; then
        echo "Bad dayname format: just specify the day name by itself." >&2
        exit 1
    fi
fi
```

```

date="$(normalize $word1)"

else

if [ -z "$word2" ] ; then
    echo "Bad dayname format: unknown day name specified" >&2
    exit 1
fi

if [ ! -z "$(echo $word1|sed 's/[[:digit:]]//g')" ] ; then
    echo "Bad date format: please specify day first, by day number" >&2
    exit 1
fi

if [ "$word1" -lt 1 -o "$word1" -gt 31 ] ; then
    echo "Bad date format: day number can only be in range 1-31" >&2
    exit 1
fi

if [ ! isMonthName $word2 ] ; then
    echo "Bad date format: unknown month name specified." >&2
    exit 1
fi

word2="$(normalize $word2)"

if [ -z "$word3" ] ; then
    date="$word1$word2"
else
    if [ ! -z "$(echo $word3|sed 's/[[:digit:]]//g')" ] ; then
        echo "Bad date format: third field should be year." >&2
        exit 1
    elif [ $word3 -lt 2000 -o $word3 -gt 2500 ] ; then
        echo "Bad date format: year value should be 2000-2500" >&2
        exit 1
    fi
    date="$word1$word2$word3"
fi
fi

/bin/echo -n "One-line description: "
read description

# Данные готовы к записи в файл

❷ echo "$(echo $date|sed 's/ //g')|$description" >> $agendafile

exit 0

```

Второй сценарий, в листинге 3.13, короче, но используется чаще.

**Листинг 3.13.** Сценарий agenda, сопутствующий сценарию addagenda из листинга 3.12

```
#!/bin/sh

# agenda -- сканирует файл .agenda в поисках записей, относящихся
# к текущей дате

agendafile="$HOME/.agenda"

checkDate()
{
    # Создать значения по умолчанию для сопоставления с текущей датой.
    weekday=$1 day=$2 month=$3 year=$4
    ❸ format1="$weekday" format2="$day$month" format3="$day$month$year"

    # И выполнить поиск среди записей в файле...

    IFS="|" # Команда read автоматически разбивает
            # прочитанные строки по символам в IFS.

    echo "On the agenda for today:"

    while read date description ; do
        if [ "$date" = "$format1" -o "$date" = "$format2" -o \
            "$date" = "$format3" ]
        then
            echo " $description"
        fi
    done < $agendafile
}

if [ ! -e $agendafile ] ; then
    echo "$0: You don't seem to have an .agenda file. " >&2
    echo "To remedy this, please use 'addagenda' to add events" >&2
    exit 1
fi

# Получить текущую дату...
❹ eval $(date '+weekday=%a' month="%b" day="%e" year="%G")
❺ day="$(echo $day|sed 's/ //g')" # Удалить возможные пробелы в начале.

checkDate $weekday $day $month $year

exit 0
```

## Как это работает

Сценарии addagenda и agenda поддерживают три типа событий: еженедельные («каждую среду»), ежегодные («каждого 3 августа») и однократные («1 января 2017»). В процессе добавления записей в файл событий их даты

нормализуются и сжимаются так, что 3 August превращается в 3Aug, а Thursday превращается в Thu. Эта операция выполняется функцией `normalize` в сценарии `addagenda` ❶.

Данная функция отсекает все, что следует за третьим символом, и преобразует первый символ в верхний регистр, а два остальных — в нижний. Такой формат соответствует стандартным сокращенным названиям дней недели и месяцев в выводе команды `date`, что необходимо для правильной работы сценария `agenda`. Остальная часть сценария `addagenda` не содержит ничего сложного; большую его часть занимает проверка формата введенных данных.

Наконец, в строке ❷, он сохраняет нормализованные данные в скрытый файл. Отношение кода, связанного с проверкой ошибок, к коду, выполняющему фактическую работу, довольно типично для хорошо написанных программ: проверка и первичная обработка входных данных позволят сделать уверенные предположения об их формате в последующих приложениях.

Сценарий `agenda` проверяет события, преобразуя текущую дату в три возможных строковых представления (*день недели*, *число+месяц* и *день+месяц+год*) ❸. Затем он сравнивает каждую из этих строк с датами из записей в файле `.agenda`. Найденные совпадения выводятся на экран.

Самый, пожалуй, интересный прием в этой паре сценариев — использование команды `eval` для присваивания четырем переменным четырех значений, определяющих дату ❹:

```
eval $(date "+weekday=\"%a\" month=\"%b\" day=\"%e\" year=\"%G\"")
```

Можно было бы получить значения по одному (например, `weekday="$(date +%a)"`), но в очень редких случаях этот способ дает ошибочные результаты, если в ходе выполнения четырех вызовов `date` произойдет смена даты, так что краткая форма с единственным вызовом предпочтительнее. Плюс, это просто круто выглядит.

Так как `date` может вернуть день как число с нежелательным начальным пробелом, следующая строка ❺ удаляет его. А теперь посмотрим, как все это работает!

## Запуск сценария

Сценарий `addagenda` предлагает пользователю ввести дату нового события. Затем, если дата имеет допустимый формат, сценарий предлагает ввести однострочное описание события.

Сопутствующий сценарий `agenda` не имеет параметров и, когда вызывается, выводит список всех событий, запланированных на текущую дату.

## Результаты

Чтобы увидеть, как работает эта пара сценариев, добавим несколько новых событий, как показано в листинге 3.14.

**Листинг 3.14.** Тестирование сценария `addagenda` и добавление нескольких событий

```
$ addagenda
Agenda: The Unix Reminder Service
Date of event (day mon, day month year, or dayname): 31 October
One-line description: Halloween
$ addagenda
Agenda: The Unix Reminder Service
Date of event (day mon, day month year, or dayname): 30 March
One-line description: Penultimate day of March
$ addagenda
Agenda: The Unix Reminder Service
Date of event (day mon, day month year, or dayname): Sunday
One-line description: sleep late (hopefully)
$ addagenda
Agenda: The Unix Reminder Service
Date of event (day mon, day month year, or dayname): march 30 17
Bad date format: please specify day first, by day number
$ addagenda
Agenda: The Unix Reminder Service
Date of event (day mon, day month year, or dayname): 30 march 2017
One-line description: Check in with Steve about dinner
```

Теперь с помощью сценария `agenda` можно быстро вспомнить, что должно произойти сегодня, как показано в листинге 3.15.

**Листинг 3.15.** Использование сценария `agenda` для поиска событий на сегодня

```
$ agenda
On the agenda for today:
  Penultimate day of March
  sleep late (hopefully)
  Check in with Steve about dinner
```

Обратите внимание, что даты в совпавших событиях представлены в форматах: *день недели*, *число+месяц* и *день+месяц+год*. Для полноты картины в листинге 3.16 показано содержимое файла `.agenda` со всеми дополнительными записями:

**Листинг 3.16.** Содержимое файла `.agenda` со всеми записями

```
$ cat ~/.agenda
14Feb|Valentine's Day
25Dec|Christmas
3Aug|Dave's birthday
4Jul|Independence Day (USA)
31Oct|Halloween
30Mar|Penultimate day of March
Sun|sleep late (hopefully)
30Mar2017|Check in with Steve about dinner
```

## Усовершенствование сценария

Этот сценарий лишь слегка затронул сложную и интересную тему. Было бы неплохо включить в него возможность заглядывать на несколько дней вперед, добавив в сценарий `agenda` арифметические операции с датой. Если в системе используется GNU-версия команды `date`, выполнить такие операции будет проще простого. Если нет, тогда для операций с датой средствами командной оболочки придется написать довольно сложный код. Далее в книге мы еще вернемся к арифметике с датами, особенно в сценариях № 99, № 100 и № 101 в главе 15.

В качестве еще одного простого усовершенствования в сценарий `agenda` можно было бы добавить вывод сообщения «Nothing scheduled for today» («На сегодня ничего не запланировано») при отсутствии совпадений с текущей датой, вместо сбивающего с толку сообщения «On the agenda for today:» («В списке событий сегодня:»), за которым ничего не следует.

Этот сценарий можно было бы использовать на компьютере с ОС Unix для вывода общесистемных напоминаний о таких событиях, как запланированное создание резервных копий, корпоративные праздники и дни рождений сотрудников. Для этого нужно сначала установить на компьютеры пользователей сценарий `agenda` и убедиться, что общий файл `.agenda` доступен только для чтения. А затем добавить вызов сценария `agenda` в файл `.login` каждого пользователя или в аналогичный файл, запускаемый в момент входа.

### ПРИМЕЧАНИЕ

Просто удивительно, насколько сильно могут различаться реализации `date` в разных системах Unix и Linux, поэтому, попробовав реализовать что-то более сложное со своей командой `date` и потерпев неудачу, загляните в страницу справочного руководства `man`, чтобы увидеть, поддерживает ли она то, чего вы желаете добиться.