Sh - the Bourne Shell

Последнее редактирование: Пт, 27 ноября, 09:44:54 2020

Ознакомьтесь с другими моими учебными пособиями на странице Unix и в моем блоге

Это мой учебник по старой оболочке Bourne. Я создал обновленную версию для оболочки POSIX, которая является более распространенной версией. Самое большое отличие заключается в том, что оболочка POSIX позволяет использовать $(.....) для подстановки команд.

Спасибо за предложения / исправления от:
Райан Пенн
Хельмут Нойяр
DJ Фазик
Дастин Кинг
пратик гоял

Содержание

Нажмите на тему в этой таблице, чтобы перейти туда. Нажмите на название темы, чтобы вернуться к оглавлению.

Оригинальная версия написана в 1994 году и опубликована в Sun Observer

> Как создавать собственные сложные команды из простых команд в UNIX toolbox.
> В этом учебном пособии обсуждается программирование Bourne shell, описывающее особенности оригинальной Bourne Shell. Новые оболочки POSIX имеют больше возможностей. Я впервые написал это руководство до того, как оболочки POSIX были стандартизированы. Таким образом, информация, описанная здесь, должна работать в оболочках POSIX, поскольку она является подмножеством спецификаций POSIX.
> Вы не получите максимальной отдачи от UNIX, если не умеете писать программы в оболочке!

Оболочка Bourne и расширение имени файла

В этом разделе рассматривается оболочка Bourne. Страницы руководства занимают всего 10 страниц, так что это не должно быть сложно освоить, верно? Ну, видимо, я ошибаюсь, потому что большинство людей, которых я знаю, изучили одну оболочку для настройки своей среды и с тех пор остались с оболочкой C. Я понимаю ситуацию. Достаточно сложно выучить **один** язык оболочки, и после некоторой борьбы с одной оболочкой они не решаются изучать другую оболочку. После нескольких сценариев новый пользователь решает, что оболочка C "достаточно хороша на данный момент", и на этом все заканчивается. Они никогда не делают следующий шаг и не изучают Bourne shell. Что ж, возможно, эта глава поможет.

Оболочка Bourne считается основной оболочкой в сценариях. Все системы UNIX имеют это, прежде всего. Во-вторых, оболочка маленькая и быстрая. В нем нет интерактивных

функций оболочки C, но что с того? Интерактивные функции не очень полезны в сценариях. Оболочка Bourne также обладает некоторыми функциями, которых нет в оболочке C. В целом, многие считают Bourne shell лучшей оболочкой для написания переносимых UNIX-скриптов.

Концепции оболочки

Что такое оболочка, в любом случае? На самом деле все просто. Операционная система UNIX представляет собой сложный набор файлов и программ. UNIX не требует какого-либо одного метода или интерфейса. Можно использовать много разных техник. Самый старый интерфейс, который находится между пользователем и программным обеспечением, - это оболочка. Двадцать пять лет назад у многих пользователей даже не было видеотерминала. У некоторых был только шумный, большой и медленный печатный терминал. Оболочка была интерфейсом к операционной системе. Оболочка, слой, интерфейс - все эти слова описывают одну и ту же концепцию. По соглашению, оболочка - это пользовательская программа, основанная на ASCII, которая позволяет пользователю указывать операции в определенной последовательности.

В оболочке UNIX есть четыре важных понятия:

- Пользователь взаимодействует с системой с помощью оболочки.
- Последовательность операций может быть написана по сценарию или автоматически, путем размещения операций в файле сценария.
- Оболочка - это полнофункциональный язык программирования с переменными, условными операторами и возможностью выполнения других программ. Он может использоваться и используется для создания прототипов новых программ.
- Оболочка позволяет легко создать новую программу, которая не является "гражданином второго сорта", а вместо этого является программой со всеми привилегиями любой другой программы UNIX.

Последние два пункта важны. DOS не имеет оболочки, которая имеет столько функций, сколько оболочка Bourne. Кроме того, невозможно написать сценарий DOS, который эмулирует или заменяет существующие команды.

Позвольте мне привести пример. Некоторые программы DOS понимают метасимвол "*". То есть команде "ПЕРЕИМЕНОВАТЬ" можно сообщить

```
ПЕРЕИМЕНОВАТЬ *.OLD *.NEW
```

Файлы "A.OLD" и "B.OLD" будут переименованы в "A.NEW" и "B.NEW". Команды DOS "КОПИРОВАТЬ" и "ПЕРЕМЕСТИТЬ" также понимают символ "*". Однако команды "TYPE" и "MORE" этого не делают. Если бы вы хотели создать новую команду, она тоже не поняла бы, что звездочка используется для сопоставления имен файлов. Видите ли, каждой программе DOS дается бремя понимания расширения имени файла. Следовательно, многие программы этого не делают.

UNIX - это совсем другая история. На оболочку возлагается бремя расширения имен файлов. Программа, которая видит имена файлов, не знает об исходном шаблоне. Это означает, что все программы действуют согласованно, поскольку расширение имени файла можно использовать с любой командой. Это также означает, что сценарий оболочки может легко заменить программу на C, насколько это касается пользователя. Если вам не нравится название утилиты UNIX, легко создать новую утилиту для замены текущей программы. Если вы хотите создать программу под названием "DIR", вы можете просто создать файл, содержащий

```
#!/bin/sh
ls $*
```

Оболочка выполняет самую сложную часть.

Другое различие между командным файлом DOS и оболочкой UNIX заключается в богатстве языка оболочки. Возможно выполнять разработку программного обеспечения, используя оболочку в качестве верхнего уровня программы. Это не только возможно, но и поощряется. Философия разработки программ в UNIX заключается в том, чтобы начинать со сценария оболочки. Получите желаемую функциональность. Если конечный результат обладает всеми функциональными возможностями и достаточно быстр, то все готово. Если это недостаточно быстро, рассмотрите возможность замены части (или всего) скрипта программой, написанной на другом языке (например, C, Perl). То, что программа UNIX является сценарием оболочки, не означает, что это не "настоящая" программа.

Проверка того, какую оболочку вы используете

Поскольку доступно много оболочек, важно научиться различать разные оболочки. Ввод команд для одной оболочки при использовании другой обязательно вызовет путаницу. Я знаю по личному опыту. Это усугублялось тем фактом, что во многих книгах, которые я использовал для изучения UNIX, никогда не упоминалось о существовании других оболочек. Поэтому первый шаг - убедиться, что вы используете правильную оболочку.

Вы можете выполнить следующую команду, чтобы определить свою оболочку по умолчанию (команда, которую вы вводите, выделена **жирным** шрифтом):

```
% echo $SHELL
/bin/csh
```

Хотя это определяет вашу оболочку по умолчанию, это не точно определяет оболочку, которую вы используете в настоящее время. Я дам вам лучший способ узнать позже. Поскольку в этой колонке обсуждается оболочка Bourne, все обсуждаемые здесь команды будут работать правильно, только если вы используете оболочку Bourne. У вас есть два варианта: поместите эти команды в сценарий Bourne shell. То есть создайте файл, сделайте так, чтобы первая строка считывалась

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

Сделайте так, чтобы вторая строка и следующие за ней содержали команды, которые вы хотите протестировать. Затем сделайте его исполняемым, набрав

```
chmod +x имя файла
```

Как только вы это сделаете, вы можете протестировать сценарий, набрав

```
./имя файла
```

Второй способ - создать новое окно (если хотите). Затем введите

```
sh
```

Вы можете увидеть изменение символов, которые оболочка выдает вам в качестве подсказки, как в примере ниже:

```
% /bin/sh
$
```

Bourne shell будет выполнять каждую введенную вами строку, пока не будет найден конец файла. Другими словами, когда вы вводите Control-D, оболочка Bourne завершается и возвращает вас к оболочке, которую вы использовали ранее. Это то же самое действие,

которое оболочка выполняет при выполнении файла сценария и достижении конца файла сценария.

Основы оболочки

Основные действия оболочки просты. Он читает строку. Это либо из файла, скрипта, либо от пользователя. Во-первых, "обрабатываются" мета-символы. Во-вторых, найдено имя исполняемого файла. В-третьих, аргументы передаются программе. В-четвертых, выполняется настройка перенаправления файлов. Наконец, программа выполняется.

Расширение мета-символов и имени файла

Поскольку оболочка считывает каждую строку, она "обрабатывает" любые специальные символы. Это включает в себя вычисление переменных (переменные начинаются с "$)" и расширение имени файла. Расширение имен файлов происходит, когда символы "*", "?" или "[" встречаются в слове. Знак вопроса соответствует одному символу. Звездочка соответствует любому количеству символов, включая ни одного. Квадратные скобки используются для указания диапазона или определенной комбинации символов. Внутри квадратных скобок дефис используется для указания диапазона или символов. Кроме того если первый символ внутри квадратных скобок является восклицательным знаком, используется дополнение диапазона. Позвольте мне привести несколько примеров:

Таблица 1

Примеры расширения имени файла Bourne shell

Шаблон	Матчи
*	Каждый файл в текущем каталоге
?	Файлы, состоящие из одного символа
??	Файлы, состоящие из двух символов
??*	Файлы, состоящие из двух или более символов
[abcdefg]	Файлы, состоящие из одной буквы от a до g.
[gfedcba]	То же, что и выше
[a-g]	То же, что и выше
[a-cd-g]	То же, что и выше
[a-zA-Z0-9]	Файлы, состоящие из одной буквы или цифры
[!a-zA-Z0-9]	Файлы, состоящие из одного символа, а не буквы или цифры
[a-zA-Z]*	Файлы, начинающиеся с буквы
?[a-zA-Z]*	Файлы, второй символ которых соответствует букве.
*[0-9]	Файлы, которые заканчиваются номером
?[0-9]	Двухсимвольное имя файла, заканчивающееся цифрой
*.[0-9]	Файлы, которые заканчиваются точкой и цифрой

Как вы можете видеть, точка не является специальным символом. Имена файлов могут содержать или не содержать точку. Программисты UNIX используют точку для стандартизации типа исходного кода каждого файла, но это всего лишь соглашение. Существует еще одно соглашение, которое касается оболочки:

> **Файлы, имя которых начинается с точки, обычно не отображаются в списке.**

Опять же, это соглашение, но *ls*, *find* и различные оболочки следуют этому соглашению. Это позволяет некоторым файлам быть "секретными" или, возможно, невидимыми по умолчанию. Вы должны явно запросить эти файлы, включив точку как часть имени файла. Шаблон ".*" соответствует всем скрытым файлам. Помните, что в каждом каталоге всегда находятся два скрытых файла: ".,", которые указывают на текущий каталог, и "..,", который указывает на каталог над текущим каталогом. Если вы хотите сопоставить все скрытые

файлы, кроме этих двух каталогов, нет простого способа указать шаблон, который всегда будет сопоставлять все файлы, кроме двух каталогов. Я использую

```
.??
```

или

```
.[a-zA-Z]*
```

Как я уже сказал, это не соответствует всем комбинациям, но работает большую часть времени. Хакеры (или взломщики, если хотите) взламывают компьютеры и часто используют странные имена файлов, такие как ". " или ".. ", чтобы скрыть свои следы. Возможно, вы не заметили, но в этих именах файлов был пробел. Ссылки на файлы с пробелами в именах требуют кавычек, о которых я расскажу позже. Лично все мои скрытые файлы сопоставляются с ".[a-z] *", а все мои скрытые каталоги сопоставляются с ".[A-Z] *". Это работает, потому что я составил свое собственное соглашение и всегда следую ему.

Косая черта также является особенной, поскольку она используется для указания пути к каталогу. Расширение имени файла не расширяется, чтобы соответствовать косой черте, потому что косая черта никогда не может быть частью имени файла. Кроме того, те же правила для расширения имени файла скрытых файлов применяются, если шаблон следует за косой чертой. Если вы хотите сопоставить скрытые файлы в подкаталоге, вы должны указать явный шаблон. В таблице 2 перечислены некоторые примеры.

Таблица 2

Расширение имени файла с помощью каталогов

Шаблон	Матчи
*	Все невидимые файлы
азбука/*	Все невидимые файлы в каталоге abc
азбука/.*	Все невидимые файлы в каталоге abc
/	Все невидимые файлы во всех подкаталогах ниже
/.	Все невидимые файлы во всех подкаталогах ниже

Расширения имени файла основаны на текущем каталоге, если только имя файла не начинается с косой черты.

Оболочка Bourne отличается от оболочки C, если мета-символы не совпадают ни с одним файлом. Если это произойдет, шаблон передается в программу без изменений. (Оболочка C либо сделает это, либо сгенерирует ошибку, в зависимости от переменной).

Если вы не уверены, как что-то будет расширяться, используйте команду *echo* для проверки. Он генерирует выходные данные более компактные, чем *ls*, и не будет отображать содержимое каталогов, как *ls*. Вы также заметите, что выходные данные отсортированы в алфавитном порядке. Оболочка не только расширяет имена файлов, но и сортирует их для всех приложений.

Поиск исполняемого файла

Как только оболочка расширяет командную строку, она разбивает строку на слова и принимает первое слово в качестве команды, которая должна быть выполнена. (Специальная переменная Bourne "IFS" содержит символы, используемые для "разбиения" строки. Обычно эта переменная содержит пробел и символ табуляции.) После этого первое слово используется в качестве имени программы. Если команда указана без явного пути к каталогу, оболочка затем выполняет поиск в разных каталогах, указанных переменной "PATH", пока не найдет указанную программу.

Если вы следовали пунктам, которые я изложил, вас не должно удивлять, что допустимая команда UNIX может быть

```
*
```

Содержимое каталога определяет результаты, но если вы создали файл с именем "0", который содержит

```
#!/bin/sh
echo Привет! Вы забыли указать команду!
```

Нажмите здесь, чтобы получить файл: 0.sh
и если это первый файл (в алфавитном порядке) в вашем каталоге, то выполнение "*" выдаст вам сообщение об ошибке. (При условии, что текущий каталог был в вашем пути поиска).

Итак, вы видите, расширение имени файла может быть где угодно в командной строке. Вы можете запускать программы с длинными именами, не вводя полное имя. Однако расширение имени файла работает только в том случае, если файл находится в указанном вами каталоге. Если вы хотите выполнить программу "my_very_own_program", не вводя полное имя файла, вы можете ввести

```
мои аргументы
```

до тех пор, пока "my_ *" расширяется до уникального имени программы. Если это было в другом каталоге, то вам нужно указать путь к каталогу:

```
/usr/local/bin/my_* аргументы
```

Понимание взаимосвязи между оболочкой и программами позволяет добавлять функции. Некоторые люди создают файл с именем "-i" в каталоге. Если кто-то затем набирает

```
rm *
```

находясь в этом каталоге, первым аргументом, вероятно, будет "-i". Это имя файла передается программе *rm*, которая предполагает, что дефис указывает на аргумент. Поэтому *rm* запускается с интерактивной опцией, защищая программы от случайного удаления.

И последнее замечание. Многие пользователи DOS жалуются, что они не могут выполнить

```
ПЕРЕИМЕНОВАТЬ *.OLD *.NEW
```

Я признаю, что это немного неудобно делать в UNIX. Я хотел бы сказать две вещи в защиту. Во-первых, вышеупомянутое использование расширения имени файла является "неестественным действием" с точки зрения философии UNIX. У оболочки, обрабатывающей расширение имени файла, есть много преимуществ, и, возможно, один недостаток в одном случае. Я считаю, что преимущества философии UNIX намного перевешивают недостатки.
Во-вторых, я все равно не считаю это недостатком. Подобное переименование файлов является неправильным. Единственная причина для этого заключается в том, что DOS делает это таким образом, и вы должны это сделать, потому что вы ограничены 11 символами. Если вы должны переименовать их, добавьте строку в конец вместо изменения исходного имени файла. Это UNIX. У вас могут быть имена файлов длиной 256 символов, поэтому такой подход не является проблемой. Поэтому, если вы должны переименовать их, используйте

```
для i в * .OLD
сделайте
```

```
mv $ i $i.NEW
готово
```

Это позволяет вам отменить то, что вы сделали, и сохранить исходное имя файла. Еще лучше переместить файлы в другой каталог, сохранив их первоначальное имя. Я бы посоветовал вам ввести

```
mkdir Старый
mv *.СТАРЫЙ старый
```

Это упрощает отмену вашего действия и работает для файлов с любым именем, а не только "* .OLD".

Цитирование с помощью Bourne Shell

Первая проблема, с которой сталкиваются программисты оболочки, - это кавычки. Стандартная клавиатура имеет три кавычки. Каждый из них имеет свое назначение, и только два используются для цитирования строк. Зачем вообще цитировать и что я подразумеваю под цитированием? Ну, оболочка понимает много специальных символов, называемых мета-символами. У каждого из них есть цель, и их так много, что новички часто страдают от мета-ити. Пример: знак доллара является метасимволом и сообщает оболочке, что следующее слово является переменной. Если вы хотите использовать знак доллара в качестве обычного символа, как вы можете сообщить оболочке, что знак доллара не указывает на переменную? Ответ: знак доллара должен быть заключен в кавычки. Почему? Заключенные в кавычки символы не имеют особого значения. Позвольте мне повторить это с акцентом.

Заключенные в кавычки символы не имеют особого значения

Удивительное количество символов имеют особые значения. Скромное пространство, часто забываемое во многих книгах, является чрезвычайно важным мета-персонажем. Рассмотрим следующее:

```
rm -i file1 file2
```

Оболочка разбивает эту строку на четыре слова. Первое слово - это команда или программа для выполнения. Следующие три слова передаются программе в качестве трех аргументов. Слово "-i" является аргументом, как и "file1". Оболочка обрабатывает аргументы и параметры одинаково и не знает разницы между ними. Другими словами, программа рассматривает аргументы, начинающиеся с дефиса, как специальные. Оболочке все равно, за исключением того, что она следует соглашению. В этом случае *rm* просматривает первый аргумент, понимает, что это опция, потому что он начинается с дефиса, и обрабатывает следующие два аргумента как имена файлов. Затем программа начинает удалять указанные файлы, но сначала запрашивает у пользователя разрешение из-за опции "-i". Использование дефиса для обозначения опции является соглашением. Нет причин, по которым вы не можете написать программу для использования другого символа. Вы могли бы использовать косую черту, как это делает DOS, для обозначения дефиса, но тогда ваша программа не сможет отличить параметр от пути к имени файла, первые символы которого являются косой чертой.

Может ли файл иметь пробел в имени? Абсолютно. Это UNIX. Существует несколько ограничений в именах файлов. Что касается операционной системы, у вас не может быть имени файла, содержащего косую черту или нуль. Оболочка - это отдельная история, и я не планирую ее обсуждать.

Обычно аргументы обозначаются пробелом. Чтобы включить пробел в имя файла, вы должны заключить его в кавычки. Другой глагол, используемый в документации UNIX, - это "escape"; обычно это относится к одному символу. Вы "заключаете" строку в кавычки, но

"экранируете" мета-символ. В обоих случаях все специальные символы обрабатываются как обычные символы.

Предположим на мгновение, что у вас есть файл с именем "file1 file2", это один файл с пробелом между "1" и "f". Если этот файл должен быть удален, один из способов заключить пробел в кавычки:

```
rm 'file1 file2'
```

Есть и другие способы сделать то же самое. Большинство людей считают, что кавычки - это то, что вы помещаете в начало и конец строки. Более точное описание процесса цитирования - это переключатель, или тумблер. Все следующие варианты эквивалентны:

```
rm 'file1 file2'
rm file1' 'file2
rm файл f'ile1'2
```

Другими словами, при чтении сценария оболочки вы должны помнить текущее "состояние цитирования". Цитата переключает состояние. Поэтому, если вы видите кавычки в середине строки, это может означать включение или выключение переключателя. Вы должны начать с начала и сгруппировать кавычки попарно.

Есть две другие формы или цитаты. Во втором используется обратная косая черта "\", которая действует только для "экранирования" следующего символа. Двойные кавычки похожи на одинарные кавычки, использованные выше, но слабее. Я объясню сильные и слабые цитаты позже. Вот более ранний пример, на этот раз с использованием других форм цитирования:

```
rm "file1 file2"
rm file1 file2
rm file1" "file2
```

Вложенные цитаты

Очень запутанная проблема заключается в размещении кавычек внутри кавычек. Это можно сделать, но это не всегда последовательно или логично. Цитирование двойных кавычек, возможно, является упрощением и делает то, что вы ожидаете. Каждая из этих команд выводит двойную кавычку:

```
echo '"'
эхо "\""
эхо \ "
```

Обратная косая черта отличается. Рассмотрим три способа вывода обратной косой черты

```
echo '\'
эхо "\\"
эхо \\
```

Как вы можете видеть, одинарные и двойные кавычки ведут себя по-разному. Двойная кавычка слабее и не заключает в кавычки обратную косую черту. Одинарные кавычки снова отличаются. Вы можете экранировать их с помощью обратной косой черты или заключить их в двойные кавычки:

```
echo \'
эхо "'"
```

Следующее **не** работает:

```
echo '\"
```

Он идентичен

```
эхо '
```

Оба примера запускают операцию цитирования, но не завершают действие. Другими словами, функция кавычек останется включенной и будет продолжаться до тех пор, пока не будет найдена другая одинарная кавычка. Если ничего не найдено, оболочка будет читать остальную часть скрипта, пока не будет найден конец файла.

Сильное или слабое цитирование

Ранее я описывал одинарные кавычки как сильные кавычки, а двойные кавычки - как слабые кавычки. В чем разница? Строгие кавычки не позволяют символам иметь особые значения, поэтому, если вы поместите символ в одинарные кавычки, вы увидите то, что получите. Поэтому, если вы не уверены, является ли символ специальным символом или нет, используйте сильные кавычки.

Слабые кавычки обрабатывают большинство символов как обычные символы, но позволяют некоторым символам (или, скорее, мета-символам) иметь особое значение. Как показано в предыдущем примере, обратная косая черта в двойных кавычках **является** специальным метасимволом. Он указывает, что следующий символ не является, поэтому его можно использовать перед обратной косой чертой и перед двойными кавычками, избегая специального значения. Есть два других мета-символа, которые разрешены внутри двойных кавычек: знак доллара и обратная кавычка.

Знаки доллара обозначают переменную. Одной из важных переменных является "HOME", которая указывает ваш домашний или начальный каталог. Следующие примеры иллюстрируют разницу:

$ **echo '$HOME'**
$HOME
$ **echo '$HOME'**
$HOME
$ **echo "$HOME"**
/home/barnett
$ **echo "$HOME"**
$HOME

Обратная кавычка выполняет подстановку команд. Строка между обратными кавычками выполняется, и результаты заменяют строку с обратными кавычками:

$ **echo 'Текущий каталог `pwd`**
Текущий каталог - `pwd`
$ **echo 'Текущий каталог - `pwd \`**
Текущий каталог - \ `pwd \`
$ **echo "Текущий каталог - `pwd`"**
Текущий каталог - /home/barnett
$ **echo "Текущий каталог - "pwd""**
Текущий каталог - `pwd`

Цитирование в нескольких строках

Существует большая разница между оболочкой C и оболочкой Bourne, когда кавычка больше строки. Оболочка C лучше всего подходит для интерактивных сеансов. Из-за этого предполагается, что цитата заканчивается концом строки, если второй символ запроса не найден. Оболочка Bourne не делает никаких предположений и прекращает заключать в кавычки только тогда, когда вы указываете вторую кавычку. Если вы используете эту оболочку в интерактивном режиме и вводите кавычки, обычное приглашение изменяется,

указывая, что вы находитесь внутри цитаты. Это смутило меня в первый раз, когда это произошло. Следующий пример Bourne shell иллюстрирует это:

```
$ echo 'Не делай этого'
> ls
> pwd
> '
Не делайте этого
ls
pwd
$
```

Это небольшое неудобство при интерактивном использовании оболочки, но большое преимущество при написании сценариев оболочки, содержащих несколько строк текста в кавычках. Я использовал оболочку C для своих первых сценариев, но вскоре понял, насколько неудобной была оболочка C, когда я включил многострочный *awk*-скрипт вместо сценария оболочки C. Обработка *awk*-скриптов Bourne shell была намного проще:

```
#!/bin/sh
# Выведите предупреждение, если какой-либо диск больше
# заполнено более чем на 95%.
/usr/ucb/df | tr -d '%' | awk '
# смотрите только на строки, где первое поле содержит "/"
$ 1 ~ / \// { если ($ 5> 95) {
 printf("Предупреждение, диск %s заполнен % 4.2f%% \ n", $ 6,$ 5);
 }
}'
```

Нажмите здесь, чтобы получить файл: diskwarn.sh

Смешивание кавычек

Наличие двух типов кавычек упрощает многие проблемы, если вы помните, как ведут себя мета-символы. Вы обнаружите, что самый простой способ избежать кавычек - это использовать другую форму кавычек.

```
эхо "Не забудь!"
эхо "Предупреждение! Отсутствует ключевое слово: "end" '
```

Кавычки внутри кавычек - возьмите два

Ранее я показал, как включить цитату в кавычки того же типа. Как вы помните, вы не можете поместить одинарную кавычку в строку, заканчивающуюся одинарными кавычками Самое простое решение - использовать кавычки другого типа. Но бывают случаи, когда это невозможно. Существует способ сделать это, но он не очевиден для многих людей, особенно для тех, у кого большой опыт работы с компьютерными языками. Как видите, большинство языков используют специальные символы в начале и конце строки и имеют escape для вставки специальных символов в середину строки. Кавычки в Bourne shell **не** используются для определения строки. Используются для **отключения** или **включения** интерпретации мета-символов. Вы должны понимать, что следующие эквивалентны:

```
echo abcd
echo 'abcd'
echo ab'c'
d echo a"b" cd
echo 'a'"b" 'c'"d"
```

Последний пример защищает каждую из четырех букв от специальной интерпретации и переключается между сильными и слабыми кавычками для каждой буквы. Буквы не нужно

заключать в кавычки, но я хотел простой пример. Если бы я хотел включить одинарную кавычку в середину строки, заключенной в одинарные кавычки, я бы переключился на другую форму кавычек, когда встречается этот конкретный символ. То есть я бы использовал форму

```
'string1"'string2"'string3'
```

где string2 - символ одинарной кавычки. Вот реальный пример:

```
$ echo 'Сильные кавычки используют '"'"', а слабые кавычки используют "'
Используйте сильные кавычки ' и используйте слабые кавычки "
```

Это сбивает с толку, но если вы начнете с самого начала и пройдете до конца, вы увидите, как это работает.

Размещение переменных в строках

Изменение кавычек в середине потока также очень полезно, когда вы вставляете переменную в середину строки. Вы могли бы использовать слабые кавычки:

```
echo "Мой домашний каталог - $ HOME, а моя учетная запись - $ USER"
```

Вы обнаружите, что эта форма также полезна:

```
echo 'Мой домашний каталог - '$ HOME', а моя учетная запись - '$ USER
```

Когда вы пишете свой первый многострочный сценарий *awk* или *sed* и обнаруживаете, что хотите передать значение переменной в середину сценария, вторая форма легко решает эту проблему.

Переменные

Оболочка Bourne имеет очень простой синтаксис для переменных:

```
переменная= значение
```

Символы, используемые для имен переменных, ограничены буквами, цифрами и символом подчеркивания. Оно не может начинаться с числа. В отличие от оболочки C, пробелы важны при определении переменных Bourne shell. Пробелы (пробелы, табуляции или новые строки) завершают значение. Если вы хотите использовать пробелы в переменной, она должна быть заключена в кавычки:

```
вопрос = 'Каково имя файла?'
```

В одной строке можно разместить несколько назначений:

```
A = 1 B = 2 C = 3 D = 4
```

Не ставьте пробел после знака равенства. Это завершает значение. Команда

```
a= дата
```

устанавливает переменную "a" равной "date", но команда

```
a= дата
```

устанавливает "a" в качестве пустой строки и **выполняет** команду date . Команда "дата"? ДА. Которое представляет...

Тонкий момент

Обратите внимание, как в одной строке выполняются две команды: изменяется переменная и выполняется программа "дата". Не очевидно, что это действительно так. На странице руководства об этом не упоминается. Еще более странно то, что некоторые команды могут выполняться, а другие - нет. Команда "дата" является внешней программой. То есть команда не встроена в оболочку, а является внешним исполняемым файлом. Другие команды являются внутренними командами, встроенными в оболочку. "Echo" и "export" являются встроенными командами оболочки и могут следовать за назначением переменной. Вы можете увидеть переменную среды, определенную следующим образом:

```
VAR=/usr/lib; экспортировать переменную
```

Но следующее работает так же хорошо:

```
VAR=/usr/lib экспорт переменной
```

Некоторые встроенные команды не могут находиться в одной строке, например "for" или "if". Команда "echo" выполняет, но может не выполнять то, что вы думаете. Не верите мне? Я приведу вам пример, и вы должны угадать, какими будут результаты. Я подозреваю, что 99,9999% из вас ошиблись бы. Наденьте свои мыслительные колпачки. Вам это понадобится. **ОБНОВЛЕНИЕ - эта информация дает другие результаты, чем когда я писал ее в 1996 году. Смотрите ниже**

Готовы?

Что делают следующие команды Bourne shell?

```
a = один; echo $ a
a = два echo $ a
a = три echo $ a > $ a
```

Я должен быть честным. Я сам провалил тест. Ну, я получил частичный кредит. Но я написал тест. Первая строка проста: "один" выводится на экран. Вторая строка ведет себя по-другому. Значение переменной "a" устанавливается равным "два", но команда echo выводит "один". Помните, что оболочка считывает строки, разворачивает метасимволы, а затем передает их программам. Оболочка обрабатывает встроенные команды как внешние команды и расширяет метасимволы перед выполнением встроенных команд. Поэтому вторая строка эффективно

```
a = два эхо один
```

и **затем** выполняется команда, которая изменяет значение переменной **после** ее использования.

Готовы к игре с кривым мячом? Что делает последняя строка? Он создает файл. Файл содержит слово "два". За 64 000 долларов и поездку в Силиконовую долину, как называется созданный файл? Для тех, кто думал, что ответ "два", мне ужасно жаль, вы не выиграли большую цену. У нас есть хорошая домашняя версия этой игры и годовой запас зубной пасты. Правильный ответ - "три". Другими словами

```
echo $ a >$ a
```

интерпретируется как

```
второе эхо> третье
```

Я не обманываю вас. Переменная "$ a" имеет два разных значения в одной строке! Кстати оболочка C этого не делает.

Обновление 2011 года - я только что попробовал это на нескольких системах, чтобы посмотреть, что получится. В старой системе Sun он вел себя так, как я заметил. Обратите внимание, что это была оболочка Bourne, а не Bash.

Я попробовал это с помощью оболочки Bash в системе 2.2 Linux. Третья строка создала файл с именем "three", но он содержал строку "one"!.

Я также попробовал это в системе Ubuntu 10.04, и третья строка выдала ошибку

```
bash: $a: ambiguous redirect
```

Обновление 2014 года: я только что попробовал его в системе Ubuntu 14.04 с bash 4.3.11, и в третьей строке был создан файл с именем "one" с содержимым "one".

Просто рассмотрите этот пример, когда поведение непредсказуемо. Теперь давайте продолжим в учебном пособии.]

Оболочка Bourne дважды оценивает метасимволы: один раз для команд и аргументов, а второй раз для перенаправления файлов. Возможно, мистер Борн спроектировал оболочку так, чтобы она вела себя таким образом, потому что он чувствовал

a= вывод> $ a

следует использовать **новое** значение переменной, а не значение **до** выполнения строки, которое может быть неопределенным и, безусловно, приведет к нежелательным результатам. Хотя настоящая причина заключается в том, что приведенная выше команда обрабатывает переменную "a" как переменную среды и устанавливает переменную, помечает ее для экспорта, а затем выполняет команду. Поскольку переменная "передается" команде средой, оболочка просто устанавливает стандартный вывод в соответствующий файл, а затем обрабатывает строку для переменных и, наконец, выполняет команду в строке. Подробнее об этом позже.

Команда set

Если вы хотите проверить значения всех ваших текущих переменных, используйте команду "set:"

$ **set**
ОТОБРАЖЕНИЕ =:0.0
HOME=/home/barnett
IFS=

LD_LIBRARY_PATH=/usr/openwin/lib:/usr/openwin/lib/
ЛОГ-ИМЯ сервера = barnett
MAILCHECK= 600
OPENWINHOME=/usr/openwin
PATH=/home/barnett/bin:/usr / ucb:/usr/bin
PS1 = $
PS2=>
PWD=/home/ barnett
SHELL=/bin/csh
ТЕРМИН =
ПОЛЬЗОВАТЕЛЬ vt100= барнетт
$

Обратите внимание на алфавитный порядок переменных и символ равенства между переменной и значением. Команда "set" - это один из способов определить, какую оболочку вы используете в данный момент. (Оболочка C помещает пробелы между переменной и значением.) Также обратите внимание на уже определенный набор переменных. Это переменные среды.

Переменные среды

UNIX предоставляет механизм для передачи информации всем процессам, созданным родительским процессом, с помощью переменных среды. Когда вы входите в систему, вам предоставляется небольшое количество предопределенных переменных. Вы можете добавить в этот список файлы запуска вашей оболочки. Каждая программа, которую вы выполняете, наследует эти переменные. Но поток информации односторонний. Новые пользователи UNIX находят это запутанным и не могут понять, почему сценарий оболочки не может изменить их текущий каталог. Представьте это так: предположим, вы выполнили сотни программ, и все они хотели изменить свое окружение на другое значение. Должно быть очевидно, что все они не могут управлять одной и той же переменной. Представьте себе сотни программ, пытающихся изменить каталог, который вы используете в данный момент! Возможно, эти переменные следует называть наследственными, а не средовыми. Дочерние процессы наследуют эти значения от родителей, но никогда не могут изменить способ создания родителей. Для этого потребуется перемещение во времени, функция, недоступная в коммерческих системах UNIX.

Как я упоминал ранее, команда оболочки "экспорт" используется для обновления переменных среды. Команда

```
экспорт a b c
```

помечает переменные "a", "b" и "c", и все дочерние процессы наследуют текущее значение переменной. Без аргументов в нем перечислены те переменные, которые отмечены таким образом. Необходима команда "экспорт". Изменение значения переменной среды не означает, что это изменение будет унаследовано. Пример:

```
ГЛАВНАЯ СТРАНИЦА =/
myprogram
```

При выполнении "myprogram" значение переменной "HOME" не изменяется. Однако в этом примере:

```
HOME=/
экспортировать домашнюю
myprogram
```

программа **действительно** получает измененное значение. Другой способ проверить это - запустить новую копию оболочки и выполнить команду "экспорт". Переменные не сообщаются.

Команда помечает переменную. Оно не копирует текущее значение в специальное место в памяти. Переменная может быть помечена для экспорта перед ее изменением. Это,

```
экспорт HOME
HOME=/
myprogram
```

работает нормально.

Специальные переменные среды

Есть несколько специальных переменных, используемых оболочкой, и есть специальные переменные, которые система определяет для каждого пользователя. Системы SunOS и Solaris используют разные переменные среды. Если вы сомневаетесь, проверьте страницы руководства. Я опишу некоторые важные переменные Solaris.

PATH - задает путь поиска

В переменной среды "PATH" перечислены каталоги, содержащие команды. При вводе произвольной команды поиск в перечисленных каталогах выполняется в указанном порядке. Двоеточие используется для разделения имен каталогов. Пустая строка соответствует текущему каталогу. Поэтому путь поиска

```
:/usr/bin:/usr/ucb
```

содержит три каталога, поиск в которых выполняется в первую очередь. Это опасно, так как кто-то может создать программу с именем "ls", и если вы измените свой текущий каталог на тот, который содержит эту программу, вы запустите этого троянского коня. Если необходимо включить текущий каталог, поместите его последним в пути поиска.

```
/usr/bin:/usr/ucb:
```

ГЛАВНАЯ страница - ваш домашний каталог

Переменная "HOME" определяет, куда отправляется "cd", когда он выполняется без каких-либо аргументов. Переменная HOME evvironment устанавливается процессом входа в систему.

CDPATH - путь поиска компакт-диска

Когда вы выполняете команду "cd" и указываете каталог, оболочка выполняет поиск этого каталога внутри текущего рабочего каталога. Вы можете добавить дополнительные каталоги в этот список. Если оболочка не может найти каталог в текущем каталоге, она будет искать в списке каталогов внутри этой переменной. Полезно добавить домашний каталог и каталог над текущим каталогом:

```
CDPATH=$HOME:..
экспортировать CDPATH
```

IFS - разделитель внутренних полей

В переменной "IFS" перечислены символы, используемые для завершения слова. Я кратко обсуждал это ранее. Обычно слова разделяются пробелом, и эта переменная содержит пробел, символ табуляции и новую строку. Хакеры находят эту переменную интересной, потому что ее можно использовать для взлома компьютерных систем. Плохо написанная программа может небрежно выполнить "/bin/ps". Хакер может переопределить переменную PATH и определить IFS как "/". Когда программа выполняет "/bin/ ps", оболочка будет обрабатывать это как "bin ps". Другими словами, программа "bin" выполняется с "ps" в качестве аргумента. Если хакер поместил программу с именем "bin" в путь поиска, то хакер получает привилегированный доступ.

PS1 - обычная подсказка

Переменная "PS1" определяет приглашение, выводимое перед каждой командой. Обычно это "$" . Текущий каталог не может быть помещен в это приглашение. Ну, некоторые люди шутят и говорят новому пользователю поместить точку внутри этой переменной. "." означает текущий каталог, однако большинство пользователей предпочитают фактическое имя.

PS2 - вторичная подсказка

Переменная среды "PS2" определяет вторичное приглашение, это приглашение, которое вы видите при выполнении многострочной команды, такой как "для" или "если". Вы также видите его, когда забываете завершить кавычку. Значение по умолчанию - ">" .

ПОЧТА - входящая почта

Переменная "MAIL" указывает, где находится ваш почтовый ящик. Оно устанавливается в процессе входа в систему.

ПРОВЕРКА ПОЧТЫ - как часто проверять почту

Переменная "MAILCHECK" указывает, как часто проверять почту в секундах. Значение по умолчанию - 600 секунд (10 минут). Если вы установите его равным нулю, каждый раз, когда оболочка вводит запрос, она будет проверять почту.

SHACCT - файл учета

Эта переменная определяет файл учета, используемый командами *acctcom* и *acctcms*.

MAILPATH - путь поиска для почтовых папок

В переменной "MAILPATH" перечислены имена файлов, разделенные двоеточием. Вы можете добавить "%" после имени файла и указать специальное приглашение для каждого почтового ящика.

Кроме того, в процессе входа в систему задается несколько переменных среды. "TERM" определяет тип терминала, а "USER" или "LOGNAME" определяет ваш идентификатор пользователя. "SHELL" определяет вашу оболочку по умолчанию, а "TZ" указывает ваш часовой пояс. Проверьте страницы руководства и протестируйте свою собственную среду, чтобы убедиться в этом наверняка. Внешняя программа "env" печатает все текущие переменные среды.

Переменные Bourne Shell - альтернативные форматы

Ранее я обсуждал простые переменные в Bourne shell. Сейчас самое время углубиться в детали. Предположим, вы хотите добавить строку к переменной. То есть, предположим, у вас была переменная "X" со значением "Accounts", но вы хотели добавить строку типа ".old" или "_new", создав "Accounts.old" или "Accounts_new", возможно, в попытке переименовать файл. Первый из них прост. Второе требует специального действия. В первом случае просто добавьте строку

```
mv $ X $ X. старый
```

Однако второй пример не работает:

```
mv $ X $ X_new # НЕПРАВИЛЬНО!
```

Причина? Ну, символ подчеркивания является допустимым символом в имени переменной Поэтому во втором примере вычисляются две переменные: "X" и "X_new". Если второе значение не определено, переменная будет иметь значение nothing, и оболочка преобразует его в

```
Учетные записи mv
```

Команда *mv* примет предложенные аргументы и пожалуется, поскольку ей всегда нужны две или более переменных. Аналогичная проблема возникнет, если вы захотите добавить букву или цифру к значению переменной.

Использование кавычек и переменных оболочки

Существует несколько решений. Первое - использовать кавычки оболочки. Помните, что цитирование запускает и останавливает обработку оболочки вложенной строки из интерпретации. Все, что нужно, это запустить или остановить условие кавычки между двумя строками, переданными в оболочку. Поместите переменную в одну строку, а константу в другую. Если переменная "x" имеет значение "home", и вы хотите добавить "run" в конец, все следующие комбинации равны "homerun":

```
$ x"запустить"
$ x'запустить'
$ x \ запустить
$ x"запустить
$ x""запустить
" $ x"выполнить
```

Использование фигурных скобок с переменными

Существует другое решение, использующее фигурные скобки:

```
${x}выполнить
```

Это обычное соглашение в программах UNIX. Оболочка C также использует ту же функцию. Утилита *make* UNIX использует это в файлах makefile и требует фигурных скобок для всех ссылок на переменные длиной более одной буквы. (*Make* использует либо фигурные скобки, либо круглые скобки).

Эта форма для переменных очень полезна. Вы могли бы стандартизировать его как соглашение. Но реальное применение получают четыре варианта этой базовой формы, кратко описанные ниже:

Форма	Значение
${переменная?слово}	Пожаловаться, если не определено
${переменная-слово}	Используйте новое значение, если оно не определено
${переменная + слово}	Противоположное приведенному выше
${переменная=слово}	Используйте новое значение, если оно не определено, и переопределите

Почему эти формы полезны? Если вы пишете сценарии оболочки, рекомендуется корректно обрабатывать необычные условия. Что произойдет, если переменная "d" не определена, а вы используете приведенную ниже команду?

```
d = `выражение $ d + 1`
```

Вы получаете "выражение: синтаксическая ошибка"
. Способ исправить это - выдать ошибку, если "d" не определено.

```
d=`выражение "${d?'не определено'}" + 1`
```

"?" генерирует ошибку: "sh: d: не определено"
Если вместо этого вы хотели, чтобы он молча использовал ноль, используйте

```
d= `выражение "${d-0}" + 1`
```

Здесь используется "0", если "d" не определено.
Если вы хотите задать значение, если оно не определено, используйте "="

```
echo $z
echo ${z=23}
echo $ z
```

Первое echo выводит пустую строку. Следующие 2 команды "echo" выводят "23". Обратите внимание, что вы не можете использовать

```
new=`выражение "${old=0}" + 1`
```

чтобы изменить значение "old", потому что команда expr выполняется как скрипт дочерней оболочки, и изменение значения "old" в этой оболочке не изменяет значение в родительской оболочке.
Я видел, как многие скрипты завершаются ошибкой со странными сообщениями, если определенные переменные не определены. Предотвратить это очень просто, как только вы освоите эти четыре метода обращения к переменной Bourne shell. Позвольте мне описать их более подробно.

${переменная?значение} - пожаловаться, если не определено

Первый вариант используется, когда происходит что-то необычное. Я думаю об этом как о опции "A ???", а вопросительный знак действует как мнемоника для этого действия. В качестве примера предположим, что выполняется следующий скрипт:

```
#!/bin/sh
cat ${HOME}/Добро пожаловать
```

Но предположим, что переменная среды "HOME" не установлена. Без вопросительного знака вы можете получить странную ошибку. В этом случае программа *cat* будет жаловаться, говоря, что файл "/ Welcome" не существует. Измените сценарий, чтобы

```
#!/bin/sh
cat ${HOME?}/Добро пожаловать
```

и выполните его, и вместо этого вы получите следующее сообщение:

```
сценарий: HOME: параметр null или не установлен
```

Как вы можете видеть, изменение всех переменных вида "$variable" на "$ {variable?}" предоставляет простой метод улучшения отчетов об ошибках. Еще лучше - это сообщение в котором пользователю сообщается, как устранить проблему. Это делается путем указания слова после вопросительного знака. Слово? Да, на страницах руководства написано одно слово. В типичном UNIX-подобном смысле это слово очень важно. Вы можете поместить одно слово после вопросительного знака. Но только одно слово. Идеально подходит для односложных оскорблений тем, кто забывает задавать переменные:

```
cat ${HOME?Пустышка}/Добро пожаловать
```

Это идеальное сообщение об ошибке, если вы хотите создать репутацию. Некоторые программисты, однако, предпочитают сохранять свою работу и друзей. Если вы попадаете в эту категорию, вы можете предпочесть выдать сообщение об ошибке, в котором пользователю сообщается, как устранить проблему. Как вы можете сделать это с помощью одного слова?

Помните мое обсуждение ранее о цитировании? И как оболочка будет считать пробел концом слова, если оно не заключено в кавычки? Решение должно быть очевидным. Просто заключите строку в кавычки, что сделает результаты одним словом:

```
cat $ {HOME?"Пожалуйста, определите HOME и повторите попытку"} /Добро пожаловать
```

Простой, но это делает сценарий оболочки более удобным для пользователя.

${переменная-по умолчанию} - использовать значение по умолчанию, если не определено

Следующий вариант не генерирует ошибку. Он просто предоставляет переменную, которая не имеет значения. Вот пример, с пользовательскими командами, выделенными жирным шрифтом:

$ **echo Y - это $ Y**
Y - это
$ **echo Y - это $ {Y-по умолчанию}**
Y по умолчанию
$ **Y=new**
$ **echo Y - это $ {Y-по умолчанию}**
Y является новым
$

Подумайте о дефисе как о мнемонике для необязательного значения, поскольку дефис используется для указания параметра в командной строке UNIX. Как и в другом примере, слово может состоять из нескольких слов. Вот несколько примеров:

$ {b-string}
$ {b- $variable}
$ {b - "фраза с пробелами"}
$ {b- "Сложная фраза с переменными, такими как $HOME или `date`"}
$ {b-`command`}
$ {b-`wc -l </etc/passwd`}
${b-`ypcat passwd | wc -l`}

Любая команда в этой фразе выполняется только при необходимости. В последних двух примерах подсчитывается количество строк в файле паролей, что может указывать на максимальное количество пользователей. Помните - вы можете использовать эти формы переменных вместо простой ссылки на переменную. Итак, вместо команды

echo Максимальное количество пользователей - $MAXUSERS

измените его на

echo Максимальное количество пользователей - ${MAXUSERS-`wc -l

Если переменная установлена, то файл паролей никогда не проверяется.

${переменная + значение} - изменить, если определено

В третьем варианте используется знак плюс вместо минуса. Мнемоника гласит: "плюс противоположен минусу". Это уместно, поскольку команда действует противоположно предыдущей. Другими словами, если переменная установлена, игнорируйте текущее значение и используйте новое значение. Это можно использовать в качестве вспомогательного средства для отладки в сценариях Bourne shell. Предположим, вы хотели узнать, когда была установлена переменная и каково ее текущее значение. Простой способ сделать это - использовать команду *echo* и ничего не повторять, когда переменная не имеет значения, используя:

echo ${A+"Текущее значение A равно $ A"}

Эта команда выводит пустую строку, если A не имеет значения. Чтобы устранить это, используйте либо версию echo в Беркли, либо версию echo System V.:

```
/usr/bin/echo ${A+"A = $ A"}"\c"
/usr/ucb/echo -n $ {A +"A = $ A"}
```

${переменная=значение} - переопределить, если не определено

Не забывайте, что эти варианты используются при ссылке на переменную и не изменяют значение переменной. Ну, четвертый вариант отличается тем, что он изменяет значение переменной, если переменная не определена. Он действует как дефис, но при использовании переопределяет переменную. Мнемоника для этого действия? Знак равенства. Это должно быть легко запомнить, потому что знак равенства используется для присвоения значений переменным:

```
$ echo Y - это $ Y
Y - это
$ echo Y - это $ {Y= по умолчанию}
Y
- это $ echo по умолчанию, Y - это $ Y
Y - значение по умолчанию
$
```

Определение переменных

При использовании этих функций вы можете захотеть протестировать поведение. Но как вы отменяете определение переменной, которая определена? Если вы попытаетесь установить его в пустую строку:

```
A =
```

вы обнаружите, что приведенные выше тесты не помогают. Что касается их, переменная определена. Он просто имеет значение nothing или null, как это называется в руководстве. Чтобы отменить определение переменной, используйте команду *unset*:

```
отмените установку
```

или, если вы хотите отменить несколько переменных

```
сбросить A B C D E F G
```

формы ${x:-y}, ${x:=y}, ${x:?y}, ${x:+y}

Как вы можете видеть, существует разница между переменной, имеющей значение null, и переменной, которая не определена. Хотя может показаться, что все, что кого-то волнует, определено или не определено, жизнь редко бывает такой простой. Рассмотрим следующее:

```
A = $ B
```

Если B не определено, является ли A также неопределенным? Нет. Помните, что оболочка оценивает переменные, а затем обрабатывает результаты. Итак, вышесказанное совпадает с

```
A =
```

который определяет переменную, но присваивает ей пустое или нулевое значение. Я думаю, что большинству сценариев не нужно знать разницу между неопределенными и нулевыми переменными. Им просто важно, имеют ли переменные реальное значение или нет. Это имеет такой смысл, что более поздние версии Bourne shell упростили тестирование для обоих случаев, создав небольшое изменение четырех форм, описанных ранее: после имени переменной добавляется двоеточие:

Форма	Значение
${переменная:?слово}	Пожаловаться, если не определено или равно нулю
${переменная:-word}	Используйте новое значение, если оно не определено или равно нулю
${переменная:+слово}	Противоположное приведенному выше
${переменная:=слово}	Используйте новое значение, если оно не определено или равно нулю, и переопределите.

Обратите внимание на разницу между "$ {b-2}" и "$ {b: -2}" в следующем примере:

$ # **a не определено**
$ **b =""**
$ **c ="Z"**
$ **echo a=${a-1}, b=$ {b-2}, c=${c-3}**
a=1, b=, c= Z
$ **echo a=${a:-1}, b=$ {b:-2}, c=$ {c:-3}**
a = 1, b = 2, c = Z

Порядок оценки

И последнее замечание - специальное слово в одном из этих форматов вычисляется только при необходимости. Поэтому команды *cd* и *pwd* в следующем: выполняется только в том случае, если выполняется слово:

```
echo ${x-`cd $HOME;pwd`}
```

Кроме того, оценка выполняется в текущей оболочке, а не в вложенной оболочке. Приведенная выше команда изменит текущий каталог, а приведенная ниже - нет, поскольку она выполняет команды в новой оболочке, которая затем завершается.

```
echo `cd $HOME;pwd`
```

Специальные переменные в Bourne Shell

Ранее я обсуждал переменные Bourne shell и различные способы их использования. До сих пор я дал вам только основы программирования на оболочке. Пришло время обсудить специальные переменные Bourne shell, которые позволят вам писать полезные скрипты. Эти специальные переменные обозначаются знаком доллара и другим символом. Если символ является числом, это позиционный параметр. Если это не буква или цифра, это переменная специального назначения.

Позиционные параметры $1, $2, ..., $9

Наиболее важной концепцией в сценариях оболочки является передача аргументов скрипту. Сценарий без параметров более ограничен. Синтаксис Bourne shell для этого прост и похож на другие оболочки и *awk*. Как всегда, знак доллара указывает на переменную. Число после знака доллара указывает позицию в командной строке. То есть "$ 1" указывает на первый параметр, а "$ 2" указывает на второй. Предположим, вы хотите создать скрипт с именем *rename*, который принимает два аргумента. Просто создайте файл с таким именем, который содержит следующее:

```
#!/bin/sh
# переименовать: - переименовать файл
# Использование: переименовать старое
имя новое имя mv $1 $2
```

Нажмите здесь, чтобы получить файл: rename0.sh

Затем выполните "chmod + x rename", и у вас будет новая программа UNIX. Если вы хотите добавить в этот скрипт простую проверку синтаксиса, используя методы, которые я обсуждал ранее, измените последнюю строку на чтение:

```
mv $ {1?"отсутствует: исходное имя файла"} $ {2?"отсутствует новое имя файла"}
```

Это не очень удобно для пользователя. Если вы не укажете первый аргумент, скрипт сообщит:

```
переименовать: 1: отсутствует: исходное имя файла
```

Как вы можете видеть, сообщается об отсутствующей переменной, в данном случае "1", что немного сбивает с толку. Второй способ справиться с этим - присвоить позиционным переменным новые имена:

```
#!/bin/sh
# переименовать: - переименовать файл
# Использование:
переименовать старое имя новое имя старое имя = $
1 новое имя = $ 2
mv $ {старое имя:?"отсутствует"} $ {новое имя:?"отсутствует"}
```

Нажмите здесь, чтобы получить файл: rename.sh

Это сообщит об ошибке следующим образом:

```
переименовать: старое имя: отсутствует
```

Обратите внимание, что мне пришлось добавить двоеточия перед вопросительным знаком. Ранее я упоминал, как вопросительный знак проверяет наличие неопределенных параметров, в то время как двоеточие перед вопросительным знаком указывает на пустые параметры, а также на неопределенные параметры. В противном случае команда *mv* могла пожаловаться на то, что у нее недостаточно аргументов.

Оболочка Bourne может иметь любое количество параметров. Однако **переменные** позиционных параметров ограничены номерами от 1 до 9. Можно ожидать, что $ 10 относится к десятому аргументу, но это эквивалентно значению первого аргумента с добавлением нуля в конце значения. Другой формат переменной, $ {10}, должен работать, но не работает. Оболочка Korn поддерживает синтаксис $ {10}, но оболочка Bourne требует обходных путей. Одним из них является команда *shift* . При выполнении этой команды первый аргумент удаляется из списка и теряется. Поэтому один из способов обработки трех аргументов заключается в следующем:

```
#!/bin/sh
arg1=$ 1;shift;
arg2=$ 1;shift;
arg3= $ 1;shift;
echo первые три аргумента - это $ arg1, $ arg2 и $ arg3
```

Команда *shift* может сдвигать более одного аргумента; Приведенный выше пример может быть:

```
#!/bin/sh
arg1=$ 1
arg2=$ 2
arg3= $ 3; shift 3
echo первые три аргумента - $ arg1, $ arg2 и $ arg3
```

Этот метод упрощает добавление аргументов, но сообщение об ошибке недружелюбно. Все, что вы получаете, это "невозможно сдвинуть" как ошибку. Правильный способ

обработки синтаксических ошибок требует лучшего понимания тестирования и ветвления, поэтому я отложу эту проблему на потом.

$ 0 - имя скрипта

В нулевом местоположении есть специальный позиционный параметр, который содержит имя скрипта. Это полезно при составлении отчетов об ошибках:

```
echo $ 0: ошибка
```

сообщит "переименовать: ошибка", когда сценарий *переименования* выполнит его. Команда *shift* не влияет на эту переменную.

$ * - все позиционные параметры

Другим способом устранения невозможности указания параметров 10 и выше является переменная "$ *". Символ "*" похож на метасимвол имени файла, поскольку он соответствует всем аргументам. Предположим, вы хотите написать скрипт, который перемещал бы любое количество файлов в каталог. Если первым аргументом является каталог, будет работать следующий сценарий:

```
#!/bin/sh
# имя_скрипта: moveto
# использование:
# файлы каталога moveto .....
directory=${1:?"Отсутствует"};shift
mv $* $каталог
```

Нажмите здесь, чтобы получить файл: moveto.sh

Если этот скрипт назывался "moveto", то команда

```
moveto / tmp *
```

может легко перемещать сотни файлов в указанный каталог. Однако, если какой-либо из файлов содержит пробел в имени, сценарий не будет работать. Однако есть решение, использующее переменную $@.

$@ - все позиционные параметры с пробелами

Переменная "$ @" очень похожа на переменную "$ *". Тем не менее, есть тонкое, но важное различие. В обоих случаях перечислены все позиционные параметры, начиная с $ 1, разделенные пробелами. Если внутри переменных есть пробелы, то "$ @" сохраняет пробелы, а "$ *" - нет. Поможет пример. Вот скрипт, называемый *EchoArgs*, который повторяет его аргументы:

```
#!/bin/sh
# Имя_скрипта: EchoArgs
# Он повторяет аргументы
#Сначала убедитесь, что мы используем стиль Berkeley echoes
PATH=/usr/ucb:$path;ПУТЬ экспорта
E="echo -n"
# повторите имя скрипта
${E} $ 0:
#теперь повторите каждый аргумент, но поставьте пробел
```

```
# перед аргументом и поместите одинарные кавычки
# вокруг каждого аргумента
${E} " '${1-"?"}'"
${ E} " '${2-"?"}'"
${ E} " '${3-"?"}'"
${ E} " '${4-"?"}'"
${ E} " '${5-"?"}'"
${ E} " '${6-"?"}'"
${E} " '${7-"?"}'"
эхо
```

Нажмите здесь, чтобы получить файл: EchoArgs.sh

Во-вторых, вот скрипт, который проверяет разницу:

```
#!/bin/sh
EchoArgs $*
EchoArgs $@
EchoArgs "$*"
EchoArgs "$ @"
```

Нажмите здесь, чтобы получить файл: TestEchoArgs.sh

Теперь давайте выполним скрипт с аргументами, содержащими пробелы:

```
./TestEcho "a b c" 'd e' f g
```

Скрипт выводит следующее:

```
./ EchoArgs: 'a' 'b' 'c' 'd' 'e' 'f' 'g'
./EchoArgs: 'a' 'b' 'c' 'd' 'e' 'f' 'g'
./EchoArgs: 'a b c d ef g' '?' '?' '?' '?' '?' '?'
./ Отголоски: 'a b c' 'd e' 'f' 'g' '?' '?' '?'
```

Как вы можете видеть, $ * и $ @ действуют одинаково, когда они не заключены в двойные кавычки. Но в двойных кавычках переменная $ * обрабатывает пробелы внутри переменных и пробелы между переменными одинаково. Переменная $@ сохраняет пробелы. В большинстве случаев $ * - это нормально. Однако, если в ваших аргументах когда-либо будут пробелы, тогда $@ требуется.

$# - количество параметров

Переменная "$#" равна количеству аргументов, переданных скрипту. Если *newscript* вернул $# в качестве результата, то оба

```
newscript a b c d
```

and

```
newscript "a b c" 'd e' f g
```

would report 4. The command

```
shift $#
```

"erases" all parameters because it shifts them away, so it is lost forever.

$$ - Current process ID

Переменная "$$" соответствует идентификатору процесса текущей оболочки, в которой выполняется сценарий. Поскольку никакие два процесса не имеют одинакового идентификационного номера, это полезно при выборе уникального временного имени файла. Следующий скрипт выбирает уникальное имя файла, использует его, а затем удаляет:

```
#!/bin/sh
filename=/tmp/ $0.$$
cat "$@" | wc -l>$filename
были найдены строки echo `cat $filename`
/bin/rm $filename
```

Нажмите здесь, чтобы получить файл: CountLines0.sh

Другое использование этой переменной - разрешить одному процессу останавливать второй процесс. Предположим, что первый процесс выполнен

```
echo $$ >/tmp/job.pid
```

Второй скрипт может уничтожить первый, предполагая, что у него есть разрешения, используя

```
kill -HUP `cat /tmp/job.pid`
```

Команда *kill* отправляет указанный сигнал указанному процессу. В приведенном выше случае сигналом является отбой или сигнал HUP. Если вы вошли в систему из дома, и ваш модем потерял соединение, ваша оболочка получит сигнал HUP.

Я надеюсь, вы не возражаете против краткого обсуждения сигналов, но эти понятия тесно связаны, поэтому стоит рассмотреть их вместе. Любой скрипт профессионального качества должен завершаться корректно. То есть, если вы завершите работу скрипта, не должно остаться никаких дополнительных файлов, и все процессы должны завершиться одновременно. Большинство людей просто помещают все временные файлы в каталог */tmp* и надеются, что в конечном итоге эти файлы будут удалены. Они будут, но иногда временные файлы большие и могут заполнить диск */ tmp*. Также некоторые люди не возражают, если для завершения работы скрипта требуется некоторое время, но если это приводит к замедлению работы системы или отправке большого количества сообщений об ошибках на терминал, тогда вам следует остановить все дочерние процессы вашего скрипта, когда ваш скрипт прерывается. Это делается с помощью команды *trap*, которая принимает одну строку и любое количество сигналов в качестве аргумента. Поэтому сценарий, который уничтожает второй сценарий, может быть написан с использованием:

```
#!/bin/sh
# выполнить скрипт, который создает /tmp/job.pid
newscript и
ловушку 'kill -HUP`cat /tmp/job.pid` 0 HUP INT TERM
# продолжить, ожидая завершения другого
<остальная часть скрипта удалена>
```

Сигналы - это очень грубая форма межпроцессного взаимодействия. Вы можете отправлять сигналы только процессам, запущенным под вашим именем пользователя. HUP соответствует зависанию, INT - это прерывание, подобное control-C, а TERM - команда завершения. При желании вы можете использовать числа, связанные с этими сигналами, которые равны 1, 2 и 15. Нулевой сигнал является особенным. Это означает, что сценарий завершен. Поэтому установка ловушки на нулевой сигнал - это способ убедиться, что некоторые команды выполняются в конце сценария. Сигнал 1, или сигнал HUP, обычно считается самым слабым сигналом. Многие программы используют это, чтобы указать, что программа должна перезапустить себя. Другие сигналы обычно

означают скорую остановку (15), в то время как самый сильный сигнал (9) не может быть захвачен, поскольку он означает, что процесс должен немедленно прекратиться. Поэтому, если вы уничтожите сценарий оболочки с помощью signal 9, он не сможет очистить какие-либо временные файлы, даже если бы захотел.

$! - идентификатор фонового задания

Предыдущий пример с $$ требует, чтобы процесс создал специальное имя файла. В этом нет необходимости, если ваш скрипт запустил другой скрипт. Эта информация возвращается в переменной "$!". Он указывает идентификатор процесса процесса, выполняемого с помощью амперсанда, который можно назвать асинхронным или фоновым процессом. Вот способ запустить фоновый процесс, выполнить что-то еще и дождаться завершения фонового задания:

```
#!/bin/sh
newscript &
trap "kill -TERM $!" 0 1 2 15
# сделай что-нибудь еще
, подожди $!
```

Я использовал цифры вместо названий сигналов. Я использовал двойные кавычки, чтобы переменная $! оценивается должным образом. Я также использовал команду *wait* , которая переводит оболочку в спящий режим до завершения этого процесса. Этот скрипт будет запускать два процесса оболочки одновременно, но если пользователь нажимает control-C, оба процесса умирают. В большинстве случаев программисты оболочки не беспокоятся об этом. Однако, если вы запускаете несколько процессов, и один из них никогда не завершается (например, "tail -f)", тогда требуется такой контроль. Другое применение - убедиться, что скрипт не запускается в течение длительного времени. Вы можете запустить команду в фоновом режиме и перейти в режим ожидания на определенное время. Если фоновый процесс к тому времени не завершится, завершите его:

```
#!/bin / sh
newscript и
sleep 10
убивают СРОК $!
```

Переменная $! изменяется только при выполнении задания с символом "&" в конце. Оболочка C не имеет эквивалента переменной $! . Это одна из причин, по которой оболочка C не подходит для высококачественных сценариев оболочки. Другая причина заключается в том, что оболочка C имеет команду, похожую на *trap*, но она использует одну команду для всех сигналов, в то время как оболочка Bourne позволяет выполнять разные действия для разных сигналов.

Команде wait не нужен аргумент. Если выполняется без аргументов, он ожидает завершения всех процессов. Вы можете запускать несколько заданий одновременно, используя

```
#!/bin/sh
job1 &
pid=$!
задание 2 &
pid = "$ pid $!"
задание 3 &
pid ="$ pid $!"
ловушка "убить -15 $ pid" 0 1 2 15
подождите
```

$? - состояние ошибки

Переменная "$?" равна возвращаемой ошибке предыдущей программы. Вы можете запомнить эту переменную, распечатать ее или выполнить другие действия, основанные на различных ошибках. Вы можете использовать это для передачи информации обратно в вызывающую оболочку, завершив сценарий оболочки с номером в качестве аргумента. Пример:

```
#!/bin/sh
# это
выход script1 12
```

Затем следующий сценарий

```
#!/bin/sh
script1
echo $?
```

напечатал бы 12.

$- Набор переменных

Переменная "$-" соответствует определенным внутренним переменным внутри оболочки. Я расскажу об этом дальше.

Параметры и отладка

Команда Bourne Shell *set* несколько необычна. Оно преследует две цели: установка определенных параметров оболочки и установка позиционных параметров. Я упоминал позиционные параметры ранее. Это аргументы, передаваемые сценарию оболочки. Вы можете представить их как массив, из которого вы можете видеть только первые девять значений, используя специальные переменные от $ 1 до $ 9. Как я упоминал ранее, вы можете использовать команду *shift*, чтобы отменить первую и переместить $ 2 на $ 1 и т. Д. Если вы хотите просмотреть все свои переменные, команда

```
установить
```

отобразит их все, включая те, которые помечены для экспорта. Вы не можете определить, какие из них отмечены. Но внешняя команда *env* сделает это.

Вы также можете явно задать эти переменные, используя команду *set* . Следовательно, оболочка Bourne имеет один массив, но только один. Вы можете поместить что угодно в этот массив, но вы потеряете старые значения. Однако вы можете сохранить их, выполнив простое назначение:

```
old= $ @
set a b c
# переменная $ 1 теперь равна "a", $ 2 = b и $ 3 = c
set $ old
# переменная $ 1 теперь имеет исходное значение, как и $ 2 и т.д.
```

Это не идеально. Если какой-либо аргумент содержит пробел внутри, эта информация не сохраняется. То есть, если первый аргумент равен "a b", а второй - "c", то впоследствии первым аргументом будет "a", вторым "b" и третьим "c". Возможно, вам придется явно обрабатывать каждый из них:

```
один = $ 1; два = $ 2; три = $ 3
установите b c
# аргумент $ 1 равен "a" и т. Д.
Установите "$ один" "$ два" "$ три"
# аргументы $ 1, $ 2 и $ 3 восстанавливаются
```

Если вы хотите очистить все позиционные параметры, попробуйте следующее:

```
установить x; сдвиг
```

Специальные опции

Как вы помните, знак доллара является специальным символом в Bourne shell. Обычно он используется для идентификации переменных. Если переменная начинается с буквы, это обычная переменная. Если он начинается с числа, это позиционный параметр, используемый для передачи параметров сценарию оболочки. Ранее я обсуждал $*, $@, $# $$, и специальные переменные $!. Но есть и другой класс переменных, или, возможно, правильный термин - flags или options . Они не читаются. То есть вы не используете их в строках, тестах, именах файлов или чем-то подобном. Эти переменные являются логическими переменными и являются внутренними для оболочки. То есть они либо истинны, либо ложны. Вы не можете присваивать им произвольные значения, используя символ "=". Вместо этого вы используете команду *set* . Кроме того, вы можете установить их и очистить, но вы не можете их прочитать. По крайней мере, не так, как другие переменные. Вы читаете их, изучая переменную "$-", которая показывает вам, какие из них установлены.

Извините, но я собираюсь поститься. Я учу вас бегать, прежде чем объясню ходьбу. Давайте обсудим первый флаг и почему он полезен.

Флаг X - Bourne Shell echo

Если у вас возникли проблемы с пониманием того, как работает сценарий оболочки, вы можете изменить сценарий, добавив команды echo, чтобы вы могли видеть, что происходит. Другим решением является выполнение скрипта с флагом "x". Существует три способа установить этот флаг. Первый и, возможно, самый простой - указать параметр при выполнении скрипта: для демонстрации предположим, что файловый *скрипт*:

```
#!/bin/sh
a=$1
echo a равно $ a
```

Затем, если вы введете

```
sh -x script abc
```

сценарий будет распечатан

```
a= abc
+ echo a - это abc
a - это abc
```

Обратите внимание, что отображаются встроенные команды, в то время как внешние команды отображаются со знаком "+" перед каждой строкой. Если у вас есть несколько команд, разделенных точкой с запятой, каждая часть будет отображаться в отдельной строке.

Переменная "x" показывает вам каждую строку перед ее **завершением**. Второй способ включить эту переменную - изменить первую строку скрипта, т.е.:

```
#!/bin/sh -x
```

Как вы можете видеть, первый способ удобен, если вы хотите запустить скрипт один раз с переменной set, в то время как второй полезен, если вы планируете повторить это несколько раз подряд. Однако большой и сложный скрипт сложно отлаживать, когда нужно

просмотреть сотни строк. Решение состоит в том, чтобы включать и выключать переменную по мере необходимости внутри скрипта. Команда

```
set -x
```

включает его, в то время как

```
установить +x
```

снова отключает флаг. Поэтому вы можете включить или выключить флаг "echo before execute", когда это удобно.

V - подробный флаг Bourne Shell

Похожим флагом является "v", или подробный флаг. Это также полезно при отладке сценариев. Разница заключается в следующем: флаг "v" повторяет строку при ее чтении, в то время как флаг "x" вызывает повторение каждой команды при ее выполнении. Давайте рассмотрим это более подробно. Учитывая сценарий:

```
#!/bin/sh
# комментарий
a= ${1:-`whoami`};b=${2:- `hostname`}
пользователь echo $ a использует компьютер $ b
```

ввод "sh -x script" вызывает:

```
+ whoami
a = barnett
+ имя хоста
b = grymoire
+ пользователь echo barnett использует компьютер grymoire
пользователь barnett использует компьютер grymoire
```

Однако "sh -v script" сообщает

```
#!/bin / sh
# комментарий
a = $ {1:-`whoami`}; b = $ {2:-`hostname`}
пользователь echo $ a использует компьютер $ b
пользователь barnett использует компьютер grymoire
```

Как вы можете видеть, комментарии повторяются с помощью флага verbose. Кроме того, каждая строка повторяется перед вычислением переменных и команд в обратных кавычках. Также обратите внимание, что команда "x" повторяет присвоение переменным a и b в двух строках, в то время как флаг verbose повторяет одну строку. Возможно, лучший способ понять разницу - флаг verbose повторяет строку до того, как оболочка что-либо с ней сделает, в то время как флаг "x" заставляет оболочку повторять каждую **команду.**Подумайте об этом как о примере до и после.

Объединение флагов

Вы можете комбинировать флаги, если хотите. Выполните скрипт с

```
скрипт sh -x -v
```

или более кратко

```
скрипт sh -xv
```

Внутри вы можете использовать любую из этих команд

```
набор -x -v
набор -xv
набор + x + v
набор +xv
```

В первой строке скрипта есть исключение. Вы можете использовать формат

```
#!/bin/sh -xv
```

но следующее не будет работать:

```
#!/bin/sh -x -v
```

Системы UNIX передают интерпретатору только первый аргумент. В приведенном выше примере оболочка никогда не видит опцию "-v".

U - сбросить переменные

Другим полезным флагом для отладки является флаг "u". Ранее я упоминал, как переменная формы "$ {x:?}" сообщает об ошибке, если переменная имеет значение null или не установлена. Что ж, вместо того, чтобы переводить каждую переменную в эту форму, просто используйте флаг "-u", который сообщит об ошибке для **любой** неустановленной переменной.

N - флаг невыполнения Bourne Shell

Простой способ проверить сложный сценарий оболочки - это опция "-n". Если установлено, оболочка будет считывать сценарий и анализировать команды, но не выполнять их. Если вы хотите проверить наличие синтаксических ошибок, но не выполнить сценарий, используйте эту команду.

Флаг выхода E - Bourne Shell

Я не особо обсуждал статус выхода. Каждая внешняя программа или сценарий оболочки завершается со статусом. Нулевой статус является нормальным. Любое положительное значение обычно является ошибкой. Обычно я проверяю статус, когда мне нужно, и игнорирую его, когда мне все равно. Вы можете игнорировать ошибки, просто не глядя на статус ошибки, который является переменной "$?", о которой я упоминал в прошлый раз. (Если программа печатает сообщения об ошибках, вам нужно перенаправить сообщения в другое место). Тем не менее, у вас может быть случай, когда скрипт работает не так, как вы ожидаете. Для этого можно использовать переменную "-e": если возникает какая-либо ошибка, сценарий оболочки немедленно завершается. Это также можно использовать, чтобы убедиться, что все ошибки известны и ожидаемы. Это было бы очень важно, если бы вы хотели изменить некоторую информацию, но только в том случае, если ошибок не произошло. Вы же не хотите повредить какую-то важную базу данных, не так ли? Предположим, что выполняется следующий скрипт:

```
#!/bin/sh
word=$1
grep $word my_file >/tmp/count
count=`wc -l
```

Скрипт выполняет поиск шаблона внутри файла и выводит, сколько раз этот шаблон был найден. Однако программа *grep* завершается со статусом ошибки, если слова не найдены. Если установлен параметр "e", оболочка завершает работу перед выполнением программы *подсчета*. Если вас беспокоили ошибки, вы могли бы установить опцию "e" в начале скрипта. Если позже вы обнаружите, что хотите проигнорировать ошибку, заключите ее в скобки с инструкциями по отключению этой опции:

```
set +e # игнорировать ошибки
grep $word my_file >/tmp/count
set -e
```

T - Bourne Shell проверяет один флаг команды

Еще один способ быстро завершить работу скрипта - использовать опцию "t". Это приводит к тому, что оболочка выполняет еще одну строку, а затем завершает работу. Это было бы полезно, если вы хотите проверить наличие сценария, но не хотите, чтобы он завершался. Возможно, выполнение скрипта занимает много времени, и вам просто важно, есть ли он там. В этом случае выполнение

```
сценарий sh -t
```

сделает это за вас.

A - метка Bourne Shell для флага экспорта

Ранее я упоминал, что вам нужно было явно экспортировать переменную, чтобы поместить ее в среду, чтобы другие программы могли ее найти. То есть, если вы выполните эти команды

```
a=
новое значение newscript
```

Теперь скрипт *newscript* будет знать значение переменной "a".

в среде с

```
экспортировать
```

Второй способ сделать это - назначить переменную непосредственно перед выполнением скрипта:

```
a=новое значение newscript
```

Это необычная форма, и она не часто используется. В строке нет точки с запятой. Если бы между присваиванием и *myscript* была точка с запятой, переменная "a" не стала бы переменной среды.

Другой способ сделать это - установить параметр "a":

```
установить -a
```

Если задано, **все** переменные, которые были изменены или созданы, будут экспортированы. Это может быть очень полезно, если вы разделите один большой скрипт на два меньших скрипта и хотите убедиться, что все переменные, определенные в одном скрипте, известны другому.

Флаг ключевого слова K - Bourne Shell

Хотя многие из рассмотренных мною опций полезны для отладки или решения проблем, другие варианты решают тонкие проблемы. Неясным параметром является переключатель "k". Рассмотрим следующую команду Bourne shell

```
a = 1 myscript b = 2 c d = 3
```

При выполнении *myscript* программе передаются четыре фрагмента информации: переменная среды "a" имеет значение 1. Скрипту передаются три аргумента: "b=2", "c" и "d =3".

Любое назначение в той же строке, что и команда, становится переменной среды, но только если оно предшествует команде. Параметры "-k" изменяют это, все три назначения становятся переменными среды, и сценарий видит только один аргумент.

Флаг хэш-функций H - Bourne Shell

Я прочитал страницу руководства, и мне было неясно. Похоже, это способ ускорить выполнение программы путем предварительного сохранения путей для каждой команды. На справочной странице Bash указано, что это включено по умолчанию. опция "-h".

Переменная $

Как я уже упоминал, вы можете использовать команду *set* для изменения значения этих флагов. Однако его нельзя использовать для проверки значений. Существует специальная переменная, называемая "$-", которая содержит текущие параметры. Вы можете распечатать значения или протестировать их. У него есть другое применение. Предположим, у вас был сложный скрипт, который вызывал другие скрипты. Предположим вы хотите отладить все сценарии. Вы можете изменить каждый скрипт, добавить нужную опцию. То есть, предположим, *newscript* может содержать

```
#!/bin/sh
myscript arg1 arg2
```

Если это заменено на

```
#!/bin/sh
sh -$- myscript arg1 arg2
```

затем, если вы наберете "sh -x newscript", *myscript* также увидит опцию "-x".

- - Опция дефиса в Bourne Shell

Я должен упомянуть, что вы можете задавать параметры, а также позиционные параметрь в одной и той же команде *set* . То есть вы можете ввести

```
набор -xvua a b c
```

Существует еще одна специальная опция, которая на самом деле не является опцией. Вместо этого он решает особую проблему. Предположим, вы хотите, чтобы один из этих параметров начинался с дефиса? То есть предположим, что у вас есть следующий скрипт, называемый *myscript*:

```
#!/bin/sh
# запомнить старые параметры
old=$@
set a b c
# изменены $ 1, $2, $ 3.
# теперь верните их обратно
, установите $old
```

Выглядит просто. Но что произойдет, если вы выполните этот скрипт со следующими аргументами:

```
myscript -d abc
```

Вы можете видеть, что произойдет? Вы получите сообщение об ошибке, когда система сообщит

```
-d: плохой вариант (ы)
```

Команда *set* считает, что аргумент "-d" является параметром оболочки. Решение состоит в том, чтобы установить специальный флаг дефис-дефис. Это сообщает оболочке, что остальные аргументы являются не параметрами, а позиционными параметрами:

```
#!/bin/sh
# запомнить старые параметры
old=$@
set a b c
# изменены $ 1, $2, $ 3.
# теперь верните их обратно, ОБРАТИТЕ
ВНИМАНИЕ на набор изменений - $ old
```

Другие варианты

Существует от трех до пяти дополнительных параметров, которые могут быть переданы в оболочку, но не изменены командой *set* . Количество опций зависит от версии операционной системы и оболочки. Переменная "$-" отобразит некоторые из этих параметров. Я буду обсуждать их по отдельности.

Параметр команды C - Bourne Shell

Опция "-c" используется, если вы хотите, чтобы оболочка выполнила команду. Это может быть полезно, если вы обычно используете оболочку C. Просто введите

```
sh -c "команда"
```

Вы можете проверить, действительно ли опция Bourne shell "-a" приводит к экспорту переменных, введя:

```
sh -ac "a = 1; env"
```

Поскольку *env* печатает все переменные среды, вы увидите, что переменная "a" действительно экспортируется и видна для внешней программы *env*.

S - вариант сеанса оболочки Bourne Shell

При интерактивном использовании оболочки программы считывают данные из стандартного ввода, а не из файла. Если вы выполняете оболочку без аргументов, она обычно ведет себя следующим образом. Это,

```
echo "echo a" | sh
```

будет повторять "a", в то время как

```
echo "echo a" | sh myscript
```

будет игнорировать стандартный ввод. Опция "s" заставляет оболочку считывать стандартный ввод для команд. Это,

```
echo "echo a" | sh -s myscript
```

никогда не выполняет скрипт *myscript*. Вместо этого он повторяет "a", как в первом примере.

I - оболочка Bourne Shell -интерактивная опция

Обычно оболочка проверяет стандартный ввод и проверяет, является ли это терминалом или файлом. Если это терминал, то он игнорирует сигнал завершения, который связан с нулевым сигналом в команде *trap* . Также ПРЕРЫВАНИЕ игнорируется. Однако, если

оболочка выполняет чтение из файла, эти сигналы не игнорируются. Опция "-i" указывает оболочке не игнорировать эти ловушки.

R - ограниченная опция оболочки Bourne Shell

Добавление опции "-r" в некоторых системах Solaris делает оболочку ограниченной оболочкой, */usr/lib/rsh*. См. Страницу руководства rsh (1M).

Привилегированный вариант оболочки P - Bourne Shell

Опция "-p" не изменит действительного пользователя и группу на реального пользователя и группу.

сбросить настройки

Как только вы экспортируете переменную, она становится переменной среды. Единственный способ отменить это - использовать команду *unset*, за которой следует имя переменной. Поэтому

```
экспорт НЕЖЕЛАТЕЛЬНОЙ
ПОЧТЫ сброс НЕЖЕЛАТЕЛЬНОЙ почты
```

не имеет никакого эффекта, и две команды отменяются.

Bourne Shell: состояние, каналы и ответвления

Предположим, у вас есть каталог для всех локальных исполняемых файлов с именем */usr/local/bin* . Также предположим, что этот каталог используется несколькими компьютерами. Нет проблем, если все машины одного типа. Однако, если некоторые из них работают под управлением Solaris, а другие - под управлением SunOS, у вас могут возникнуть проблемы. Некоторые исполняемые файлы работают только для определенных типов архитектур (например, общедоступная программа *top*). У вас также могут быть разные типы UNIX-систем. Это может быть реальной проблемой. Одним из решений является создание сценария-оболочки, который вызывает соответствующий сценарий для машины. То есть предположим, что у вас есть каталог с именем */usr/local/SunOS/5.4/sun4m/bin*, который содержит исполняемые файлы для Solaris 2.4 (SunOS 5.4). Если вы переместите исполняемый файл из */usr/local/bin* и поместите его в соответствующий каталог, и замените его сценарием оболочки, который содержит:

```
#!/bin/sh
/usr/local/`uname -s`/`uname -r`/`uname -m`/bin/`базовое имя $ 0` "$@"
```

Ваша проблема будет решена. Вроде того. С этим скриптом есть четыре проблемы, или, скорее, четыре способа улучшить скрипт.

Ненужное выполнение процесса

Первая проблема заключается в том, что программа *uname* выполняется три раза при каждом выполнении этого скрипта. Простое решение - использовать переменные среды. То есть, если переменные среды заданы, то нет необходимости выполнять программу *uname*:

```
#!/bin/sh -a
OS=${OS= `uname -s`}
REV=${REV=`uname -r`}
ARCH=${ARCH= `uname -m`}
CMD=`базовое имя $ 0`
/usr/local/ $OS/ $REV/$ARCH/bin/$CMD "$@"
```

Опция Bourne shell "-a" указывает, что все измененные переменные должны быть помечены для автоматического экспорта. Этот скрипт проверяет три переменные среды, и если они не определены, он устанавливает эти переменные. Поэтому вы можете задать эти переменные один раз при входе в систему. После этого программа *uname* не должна выполняться.

$ @ против $ {1 + $ @}

Ранее я предлагал использовать "$ @" для передачи всех аргументов другому сценарию оболочки. Это работает на современных компьютерах SunOS и Solaris, но другие системы UNIX могут не работать. Если вы немного подумаете о том, что я сказал о цитировании, вь должны начать видеть, что происходит нечто волшебное. Если вы возьмете любые другие переменные и заключите их в двойные кавычки, вы получите один аргумент. Если переменная не была установлена, вы получите один аргумент, не содержащий ничего. Поэтому "$ @" не может работать корректно. Тем не менее, это так. Причина в том, что оболочка считает "$ @" особым случаем. Однако не все версии Bourne shell имеют одинаковое поведение. Некоторые преобразовывают

```
"$@"
```

Для

```
""
```

если аргументы не указаны. Для программы-оболочки общего назначения это неправильно. Программа *cat* будет жаловаться, что не может открыть предоставленный файл, и не будет распечатывать имя файла, потому что оно есть.

Исправление для этих систем заключается в использовании шаблона:

```
${1+"$@"}
```

Чтобы объяснить, если определено значение "$ 1", замените его на "$ @". Если оно не определено, ничего не делайте. Поэтому более переносимая форма:

```
/usr/local/$OS/$REV/$ARCH/bin/$CMD ${1+"$@"}
```

Состояние и потерянные процессы

В сценарии есть еще два недостатка: один незначительный и один серьезный. Во-первых, сценарий создает новый процесс без необходимости. Это небольшой момент, но если вы хотите оптимизировать часто выполняемый скрипт, он того стоит. Вторая проблема заключается в том, что скрипт неправильно возвращает статус выхода. Исправление простое. Поместите *exec* перед инструкцией:

```
exec /usr/local/$OS/$REV/$ARCH/bin/$CMD ${1+"$@"}
```

Обычно оболочка создает собственную копию для каждой строки, а затем выполняет команду в строке с новым процессом. Команда *exec* выполняет второй шаг, не требуя, чтобы оболочка копировала себя. Если команда *exec* выполнена успешно, оболочка никогда не выполняет следующую строку.

Вторая проблема со сценарием - это статус выхода. Версия без скрипта "exec" не выполняет команду *exit*. Следовательно, статус выхода равен нулю. Программа с командой "exec" никогда не завершается (если программа не найдена). Вместо этого программа, которую она выполняет, завершает работу, и статус выхода передается сценарию, который вызвал сценарий-оболочку. Почему это важно? Ну, значение статуса является основой управления потоком в Bourne shell.

Простое управление потоком

Это другая тема для этого раздела. Есть несколько тонких моментов, так что потерпите меня. Простейшая форма - одна из этих вариаций

```
command1 && command2
command1 || command2
```

Как я упоминал ранее, у программы есть только два способа передать информацию другому процессу: файл / канал или статус завершения. Программа не может использовать переменные среды для передачи информации обратно процессу, который ее создал. Большую часть времени процесс выполняет файловый ввод-вывод. Статус выхода - еще один быстрый и удобный метод.

Статус представляет собой целое число от 0 до 255. Оболочка может либо проверять целочисленное значение статуса выхода, либо обрабатывать значение как логическое. Ноль равен true, все остальные значения равны false. Если вы не укажете статус завершения, система вернется со статусом последней выполненной команды.

UNIX поставляется с двумя программами с именами *true* и *false*, которые представляют собой просто команды "exit 0" и "exit 255". Ну, *настоящая* программа даже не имеет команды exit . Поскольку команды не выполняются, статус выхода равен нулю. Итак, по сути, команда *true* абсолютно ничего не делает, но занимает девять строк, включая уведомление об авторских правах, чтобы ничего не делать. Однако я должен предупредить вас, что если вы когда-нибудь создадите сценарий оболочки, который ничего не делает, вы рискуете вызвать юридическую ярость юридического отдела AT & T за обратное проектирование программы, на разработку которой ушло неисчислимое количество часов.

Это сенсация. Использование статуса дает вам управление потоком. "&&" часто называют оператором "и". Он выполняет следующую команду, если первая команда имеет значение true . Оператор "||" - это команда "или", которая выполняется, если команда имеет значение false. Для иллюстрации:

```
true && echo эта строка напечатана
ложно || echo эта строка напечатана
верно || echo эта строка НЕ напечатана
ложно && echo эта строка НЕ напечатана
```

Замените слова "и" или "или" в приведенных выше примерах и прочитайте их тихо про себя, чтобы понять, почему. При желании вы можете объединить эти операторы в одну строку:

```
команда command && echo выполнена успешно || ошибка команды echo
```

Позвольте мне сделать краткое обсуждение. Символ канала "|" является особенным. Несколько команд можно объединить с помощью каналов в нечто, называемое *конвейером*. Оболочка имеет пять различных механизмов для объединения конвейеров в список. Вы все знаете, что символ конца строки является одним из пяти. Вот примеры других четырех:

```
cmd1; cmd2; cmd3; cmd4
cmd1 & cmd2 & cmd3 & cmd4
cmd1 && cmd2 && cmd3 && cmd4
cmd1 || cmd2 || cmd3 || cmd4
```

Точка с запятой указывает оболочке работать последовательно. Сначала выполняется "cmd1", затем "cmd2" и т. Д. Каждая команда запускается и выполняется до тех пор, пока для них не требуется ввод из предыдущей команды. Команда "&" запускает каждый

процесс отдельно. Порядок не является последовательным, и вы не должны предполагать, что одна команда завершается раньше другой. Последние два примера, как и первый, выполняются последовательно, пока статус правильный. В примере "&&" выполняется "cmd4", если все три предыдущие команды выполнены. В примере "||" "cmd4" выполняется если первые три завершаются неудачно. "&&" и "||" имеют более высокий приоритет, чем ";" и "&", но ниже, чем "|". Поэтому

```
a | b && c; d || e | f ;
```

оценивается как

```
(a | (b && c)) ; ((d | | e) | f) ;
```

Команды || и && можно использовать для простого if-then-else . Обратите внимание, что

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

если cmd1 завершается успешно, а cmd2 завершается неудачно, то будет выполнен cmd3. Чтобы предотвратить это, можно использовать

```
cmd1 && {cmd2;выход 0; } || cmd3
```

Изменение приоритета

Если вы хотите изменить приоритет или порядок вычисления, вы можете использовать фигурные скобки или круглые скобки. Между этими двумя есть некоторые тонкие различия. Синтаксически есть два различия:

```
(cmd1; cmd2) | cmd3
{ cmd1; cmd2; } | cmd3
```

Обратите внимание на точку с запятой в конце списка в фигурной скобке. Также обратите внимание, что после первого "{" требуется пробел. Есть еще одно отличие: скобки заставляют оболочку выполнять новый процесс, в то время как фигурные скобки этого не делают. Вы можете задать переменные в фигурных скобках, и они будут известны за пределами фигурных скобок. В приведенном ниже примере первое echo выводит "СТАРОЕ", а второе - "НОВОЕ":

```
a= СТАРЫЙ
(a= НОВЫЙ); echo $a
{ a=НОВЫЙ;} ; echo $a
```

Собираем все это вместе

Я объяснил части. Теперь я попытаюсь собрать все воедино. Если вы хотите временно игнорировать ряд команд, не добавляя "#" перед каждой строкой, используйте следующее:

```
false && {
команда1
команда2
команда3
}
```

Измените значение *false* на *true*, и команды будут выполнены. Фигурные скобки и круглые скобки могут быть вложенными:

```
A && {
 B && {
echo "A и B оба прошли"
 } || {
 echo "A пройдено, B не удалось"
 }
} || echo "Сбой"
```

Скобки и фигурные скобки полезны, когда вы хотите объединить стандартный вывод нескольких команд. Например, предположим, что вы хотите добавить строку, содержащую "BEGIN", в середину конвейера перед командой "C":

```
A | B | C | D
```

Первое, что может попробовать новичок, это:

```
A | B | echo "НАЧАТЬ" | C | D
```

Это не работает. Команда *echo* не является фильтром. Любой ввод в команду *echo* отбрасывается. Поэтому программа "C" видит слово "BEGIN", но не видит вывод программы "B", который теряется. Изменение строки на

```
A | B | echo "НАЧАТЬ"; C | D
```

также не работает. Программа "C" не имеет своего ввода, подключенного к каналу. Поэтому оболочка использует пользовательский терминал для стандартного ввода. Исправление заключается в использовании одной из этих форм:

```
A | B | { echo "BEGIN" ; C;} | D
A | B | (echo "BEGIN"; C) | D
```

Требуется некоторое время, чтобы понять, когда это необходимо. Вы должны знать, какие команды считывают стандартный ввод, а какие генерируют вывод, и какие команды обладают гибкостью, позволяющей вам выполнять либо то, либо другое. Команда *cat* позволяет указать дефис в качестве опции, которая указывает на стандартный ввод. Поэтому другой способ сделать это:

```
echo BEGIN >/tmp/begin
A | B | cat /tmp/begin - | C | D
```

Скобки полезны, когда вы хотите изменить состояние процесса, не затрагивая другие процессы. Например, изменение текущего каталога. Обычный метод копирования деревьев каталогов может быть выполнен с помощью команды *tar*:

```
tar cvf . | ( cd newdirectory; tar xfBp - )
```

Скобки также можно использовать для изменения разрешений терминала. Если у вас есть принтер, подключенный к порту компьютера, и вы хотите изменить скорость передачи данных в бодах на 2400, во время печати файла одним из способов является ввод:

```
(stty 2400; файл cat) >/dev/printer
```

Одним из наиболее эффективных способов использования этих методов является объединение тестов с каналами. Видите ли, использование разных ветвей не приводит к автоматическому отключению конвейеров. Оболочка C не справляется с этим хорошо, но оболочка Bourne справляется. Предположим, вы хотите, чтобы фильтр считывал стандартные входные данные, и если входные данные содержат специальный шаблон, вы хотите, чтобы фильтр добавлял строку перед потоком

информации. В противном случае вы хотели добавить еще одну строку после входного потока. Простой способ сделать это - использовать функции оболочки:

```
#!/bin/sh
tee /tmp/file | \
(grep MATCH /tmp/file >/dev/null && cat header - || cat - trailer )
/bin/rm /tmp/file
```

Как вы можете видеть, создается временный файл, а затем программа *grep* ищет в копии шаблон. Если найдено, сначала выводится заголовок, а затем остальная часть входного потока. Если не найдено, выводится поток, а затем добавляется трейлер. Это работает, потому что команда *grep* считывает стандартный считывает стандартный ввод, если в качестве аргумента не указано имя файла. Однако, если указано имя файла, стандартный ввод игнорируется.

Я должен упомянуть, что большинство людей тестируют условия, используя команду *i* вместо операторов "&&" и "||". Структура может показаться необычной на первый взгляд, но чем больше вы ее используете, тем больше применений вы найдете для нее.

Команды управления потоком Bourne Shell: Если, пока и до

Некоторые могут найти последовательность этих руководств очень странной. Все эти слова и ничего в инструкции *if*. Странно, но необходимо, если основы правильно объяснены, что и является моей целью. Слишком много руководств касаются основ, и люди вскоре попадают в беду. Ранее я обсуждал статус, возвращаемый командами, и представил различные скобки, используемые для построения списков.

Что такое список? Существует три способа группировки команд:

Простая команда
Конвейер
Список

Простая команда представляет собой набор слов, разделенных пробелами.
Конвейер - это группа простых команд, разделенных символом "|". Более ранние версии оболочки использовали символ "^" вместо символа канала. У старых клавиатур не было символа pipe. Более новые версии оболочки используют оба. Статус завершения последней команды - это статус, возвращаемый конвейером.
Список представляет собой серию конвейеров, разделенных символами &, ;, && или || и заканчивающихся точкой с запятой, символом амперсанда или символом новой строки.

Команда может быть либо простой командой, либо сложной командой. Сложные команды могут быть одной из перечисленных ниже:

если *список* **, то** *список* **fi**
, если *список* **, то** *список* **else** *список* **fi**
, если *список* **, то** *список* **elif** *список* **, затем** *список* **fi**
, если *список* **, то** *список* **elif** *список* **, затем** *список* **elif** *список* **, затем** *список* **fi**
, если *список* **, затем** *список* **elif** *список* **, затем** *список* **else** *список* **fi**
, если *список* **, затем** *список* **elif** *список* **, затем** *список* **elif** *список* **else** *список* **fi**
, пока *списоксписок* **дел выполнен**
до тех пор, пока *список* **дел не будет выполнен**

Команды "if" могут быть вложенными. Поэтому выше перечислены только некоторые из комбинаций. Также обратите внимание, что *fi* - это *если* наоборот.

Команды, которые должны быть первыми в строке

Одним из моментов, который поначалу доставлял мне проблемы с Bourne shell, была концепция списка. Это простая концепция. Следующие слова должны быть первыми в строке:

```
если
тогда
else
elif
fi
case
esac
на
время
, пока
не
будет сделано
{
}
```

Строка не обязательно должна начинаться с фактического начала строки. Если перед словом стоит точка с запятой или амперсанд, это делает то же самое. Позвольте мне объяснить это по-другому. Один из способов написать оператор *if*:

```
если true
, то
echo это было true
fi
```

(Отступ сделан для удобства.) Та же сложная команда также может быть написана:

```
если true; тогда echo это было true; fi
```

Оба они эквивалентны описанию, которое я дал ранее:

если *список* **, то** *список* **fi**

Список - это просто команда, которая заканчивается точкой с запятой **или** символом новой строки.

Команда "if" выполняет список, если список после команды "if" имеет значение true. Список может содержать более одной команды. Последний используется для принятия решения:

```
если
false
false
true
то
echo это выведет
fi
```

Список "else" будет выполнен, если условие теста равно false. Добавление "elif" позволяет проводить несколько тестов. Кроме того, списки могут содержать дополнительные инструкции "if":

```
если testa; тогда
echo testa равно true



, если testb; тогда echo testb равно true, если testc; тогда
echo testc равно true
, иначе
```

```
echo все тесты завершаются
ошибкой fi
fi
```

Иногда требуется выполнить тест, если условие равно false. Вы можете попробовать следующее, но при этом возникнет синтаксическая ошибка:

```
если условие
, то
# игнорировать это
, иначе
условие echo равно false
fi
```

За операторами "if", "then" и "else" **должен** следовать список. Комментарий - это не список. Должна быть команда. В этом случае можно использовать команду ":", которая ничего не делает:

```
если условие; то :; иначе
условие echo равно false
fi
```

Вы можете заметить, как я меняю стиль отступов. Любая форма является правильной. Выберите тот, с которым вам удобнее. Кроме того, оболочка C разрешает комментарий в качестве единственного оператора внутри условного блока.

Цикл While - loop при значении true

Тест "если" выполняется один раз. Если вы хотите выполнить цикл, пока тест выполняется, используйте команду "while":

```
пока mytest
выполняет
echo, mytest по-прежнему
выполняется true;
```

Команда "while" полезна при чтении входных данных в сочетании с командой "read", которая считывает стандартные входные данные, присваивает своим аргументам увиденные слова и возвращает 0 или false при достижении конца файла. Поэтому следующее повторяет каждую строку:

```
во время чтения
do
echo $ a
готово
```

Это можно использовать для запроса списка аргументов:

```
echo "Пожалуйста, введите аргументы"
echo "Введите Control-D (EOF), когда закончите"
args="":
echo '?'
при чтении
do
args="$ args $ a"
echo '?'
готово
echo "Аргументы - это $ args"
```

Многие люди забывают, что список следует за командой "while". Они предполагают, что одна команда должна следовать за ключевым словом "while". Неправда. Приведенный

выше фрагмент может быть написан:

```
echo "Пожалуйста, введите аргументы"
echo "Введите Control-D (EOF), когда закончите"
args="":
в то время
как echo '?'
прочитайте
do
args="$ args $ a"
готово
echo "Аргументы равны $ args"
```

Это помещает приглашение прямо перед запросом, что является хорошим стилем программирования. Команда "while" может быть объединена с перенаправлением ввода

```
файл cat | при чтении

файла do содержит строку $a done
```

Команда "while" может получать входные данные из подоболочки и помещаться внутри подоболочки. Следующий сценарий

```
#!/bin/sh
(echo a b c;echo 1 2 3) | (при чтении a; выполнить
echo $ a $a
echo $ a $a

) | tr a-z A-Z
```

генерирует следующий вывод:

```
A B C A B C
A B C A B C
1 2 3 1 2 3
1 2 3 1 2 3
```

Также работает следующее:

```
при чтении a
делать
echo a= $a
готово
```

Оболочка C не обеспечивает такой гибкости при перенаправлении стандартного ввода.

Есть несколько тонких моментов, касающихся команд "чтение" и "пока". Ниже приведена синтаксическая ошибка:

```
пока true
do
# комментарий
выполнен
```

В списке между командами ""do" и "done" должна быть команда. Команда null ":" исправит синтаксическую ошибку:

```
пока true
do
:
готово
```

Я ожидал

```
echo 1 2 3 | читать a; echo a - это $ a
```

чтобы работать, но это не так. По крайней мере, в системах, которые я пробовал, это не работает. Однако следующее делает:

```
echo 1 2 3 | ( считывает a; echo a равно $ a)
```

Следующее кажется странным, но работает:

как и:

```
( чтение a; echo a - это $ a
чтение a; echo a - это $ a
чтение a; echo a - это $ a
)
```

Считываются ровно три строки файла. Я должен подчеркнуть, что C shell не может сделать это легко. В нем нет встроенного механизма для чтения точной одной строки. Кроме того, команда оболочки C для чтения одной строки считывается только с управляющего терминала и не может быть перенаправлена для получения входных данных из файла или канала, в то время как оболочка Bourne может.

Что произойдет, если команда "read" имеет более одного аргумента? Команда разбивает строку ввода на слова, используя пробелы в качестве символов между словами. Каждому слову присваиваются разные переменные. Если слов недостаточно, последним переменным присваивается пустое значение. Если их слишком много, последняя переменная получает остатки. Что будет напечатано, если будет выполнено следующее?

```
эхо 1 2 3 4 5 | (
прочитайте a b c
echo first $ a
echo second $ b
echo third $ c
)
```

Первая переменная - "1", вторая - "2", а последняя - "3 4 5".

Вместе эти две команды обеспечивают большую гибкость, большую, чем кажется на первый взгляд.

Если вы хотите зацикливаться вечно, вы можете делать то, что делает большинство людей:

```
пока true

, echo никогда
не останавливается
```

При этом выполняется пустой скрипт "true". Другой способ сделать то же самое без необходимости выполнения другого процесса - использовать команду ":":

```
в то время как:
выполнение
```

```
echo никогда
не прекращается
```

До тех пор, пока цикл не станет истинным

Команда until действует так же, как команда "while", за исключением того, что тест инвертируется.

```
пока testa
do
echo еще не
сделано
```

будет повторяться "еще нет", пока "testa" не станет true (завершается с нулевым статусом).

Далее, команды "for" и "case".

Команды управления потоком Bourne Shell

Ранее я обсуждал команды Bourne Shell "if", "while" и "until". Далее я расскажу о двух других командах, используемых для управления потоком - "for" и "case". Шаблоны для этих команд:

```
слово в названии для готового
списка сделайте имя для... список выполненных
действий (регистровое слово в esac
, регистровое слово в шаблоне) список;; регистровое слово esac
в шаблоне | шаблоне) список;; esac
```

Для - повторения при изменении переменной

Команда "for" выполняет команды в списке, но изменяет значение переменной для каждого цикла. Чтобы проиллюстрировать, команда

```
для i в 1 2 3 4 5 6 7 8 9 10
do
echo $я
сделал
```

выводит десять строк с номером в каждой строке. Переменной "i" присваиваются значения каждого из десяти чисел. Это также может быть записано как:

```
числа="1 2 3 4 5 6 7 8 9 10"
ибо я в $numbers
делаю
echo $i
done
```

Изменить

```
для i в $numbers
```

Для

```
для i в $ numbers $ numbers
```

и скрипт выведет в два раза больше строк. Переменные в вычисляются так же, как и все переменные. Изменить

```
для i в $numbers
```

Для

```
для i в "$ numbers"
```

и он распечатает одну строку со всеми десятью числами, потому что после команды "in" есть только один аргумент. Существует много способов указать список аргументов в команде "for". Вы можете использовать умную вещь для изменения двух переменных в цикле, объединив команду "for" с командой "set":

```
для аргументов в "a A", "b B", "c C"

установите значение $ args
echo в нижнем регистре - 1 доллар, в верхнем регистре - 2 доллара
.
```

Конечно, вы можете использовать обратные кавычки для использования выходных данных команды в качестве создателя списка:

```
для i в `файле ls *`
```

Информация о строках за раз теряется, поскольку все символы новой строки удаляются. Поэтому вы не можете легко использовать команду "set" для изменения двух или более переменных одновременно. Если вы хотите изменить несколько аргументов из файла, вам нужно либо использовать последовательность "во время чтения", которую я описал ранее, либо прочитать переменную и разделить одну переменную на несколько, используя команду типа "tr" для преобразования некоторых символов в пробелы, а затем с помощью "set" для изменения несколькихпеременные сразу:

```
#!/bin/sh
# прочитайте файл /etc/password
# используйте `cat /etc/passwd`, но пробелы обрабатываются как новые строки
# поэтому замените пробелы на _

для i в `tr ' ' '_'

установите `echo $ i | tr ':' ' "
пользователь echo: $ 1, UID: $ 3, Домашний каталог: $ 6
готово
```

Вы можете использовать стандартный ввод, чтобы получить список значений для переменной:

```
для a в `cat`
```

Наконец, вы можете комбинировать переменные, константы и выполнение программ:

```
для i в BEGIN $ a "$ b $ c" `cat file` END
do
echo i - это $ i
готово
```

Конечно, любая переменная, кроме "i", будет работать. Вам не нужно указывать список значений после команды "in". Если не указано, используются позиционные аргументы. Для иллюстрации, если у вас был скрипт с именем "script1" и выполненный

```
сценарий 1 a b c
```

и хотел посчитать количество слов в каждом файле, тогда вы могли бы использовать

```
#!/bin/sh
в echo $ 1 есть слова `wc -l <$1`
```

, в echo $ 2 есть слова `wc -l <$2`
, в echo $ 3 есть слова `wc -l <$3`

Другой способ сделать то же самое заключается в следующем:

#!/bin/sh
для файла
do
echo $file содержит слова `wc -l <$file`
готово

Проверка нескольких обращений

Оператор case функционирует как сложный оператор "if" с несколькими предложениями. Шаблон:

шаблон *в списке* регистров слов | *шаблонов*;; **esac**

Предположим, вы хотите получить ответ "да" или "нет" на вопрос. Примером является:

```
echo ответьте "да" или "нет
" прочитайте
регистр слов $word в
yes | YES )
 echo вы ответили да
;;
нет | НЕТ)
 echo вы ответили нет
;;
esac
```

Оператор "case" работает с шаблонами, такими как сопоставление имен файлов. Приведенный выше пример не позволяет указать ни одного символа. Вы должны ввести полное слово. Если вы введете "Да", оно не распознает его как "да", потому что первая буква написана в верхнем регистре, а остальные буквы - в нижнем регистре. Есть исправление. Шаблоны - это шаблоны имен файлов. Поэтому вы можете изменить приведенный выше сценарий, чтобы:

```
echo ответьте "да" или "нет
" прочитайте
регистр слов $word в
[Yy]* )
 эхо вы ответили да
 ;;
 [Nn]* )
 эхо вы ответили нет
 ;;
 * )
 echo вы не сказали ни да, ни нет;;
esac
```

Вы увидите последний тест, содержащий звездочку. Это стандартный метод указания условия по умолчанию для инструкции "case". Последний шаблон в этом случае всегда будет совпадать.

Между "case" и "in" может быть только одно слово. Ниже приведена синтаксическая ошибка:

случай a b c в

Чтобы предотвратить эту ошибку, особенно когда проверяемый элемент поступает из другой программы, используйте переменную.

```
#!/bin/sh
arg="a b c"
case $arg в
[aAbBcC] ) echo это никогда не соответствует;;
"a b c") echo это будет соответствовать;;
esac
```

Вы могли бы заключить "$ arg" в кавычки, но в моей системе Sun это не требуется. Возможно, это хорошая идея, на случай, если какая-то более старая версия Bourne shell содержит ошибку. Вы также заметите удвоенную точку с запятой. Это обязательно. Рассмотрим этот фрагмент:

```
case $arg в
a ) echo yes; b ) echo no;;
esac
```

Насколько известно оболочке, буква "b" выглядит как команда, за которой следует незаконный ")". ";;" необходимо, чтобы сообщить оболочке, что следующее найденное слово является шаблоном, а не командой. Кстати, следующее является законным:

```
case $ arg в
esac
```

Он ничего не делает, но оболочка принимает этот синтаксис.

Прервать и продолжить

Команды "for" и "while" выполняют каждую часть кода более одного раза. Если вы хотите проверить условие и выйти из этого цикла, вы можете. Вы можете управлять этим с помощью команд "прервать" или "продолжить". "Break" приводит к выходу элемента управления из оператора "for" или "while". Оператор "continue" заставляет цикл немедленно запускать следующий цикл. Ранее я использовал оператор "case", чтобы проверить, является ли ввод да или нет. Если ни то, ни другое, возникает ошибка. Если вы хотите подождать, пока у вас не будет правильного ответа, можно использовать оператор "break":

```
в то время как:

повторите "Введите да или нет"
, прочитайте ответ
"$answer" в
[yYnN]*) break;;
ecas
сделано
```

Обратите внимание, что я использовал нулевую команду ":" вместо "/bin/true"." Оба имеют одну и ту же функцию, но команда null встроена в оболочку.

Команда "продолжить" приводит к немедленному переходу команды "for" или "while" к следующему циклу. Следующий цикл печатает только нечетные числа:

```
для числа в 1 2 3 4 5 6 7 8
выполните
регистр "$ number" в
2/4/6/8) продолжить;;
esac
echo $number
готово
```

Команды "for" и "while" могут быть вложенными. В каких циклах работают команды "break" и "continue"? Они работают с самым внутренним циклом, но вы можете

переопределить это. Если вы поместите число после команды, число определяет глубину цикла. Возьмите следующее:

```
для числа в 1 2 3 4 5 6
сделайте
для буквы в a b c d e f g
сделайте
case $ number в
3) разорвать
esac
echo $ number $ letter

готово сделано
```

При этом будут напечатаны все комбинации букв и цифр, за исключением того, что не будет напечатано ни одной строки с цифрой "3". Однако измените разрыв на "разрыв 2", и скрипт будет печатать только те комбинации, которые содержат 1 или 2.

Expr - вычислитель выражений Bourne Shell

Этот раздел посвящен команде *expr*, используемой для выполнения вычисления выражения.

Оболочка Bourne изначально не имела никакого механизма для выполнения простых арифметических действий. В более старых версиях UNIX использовались внешние программы, либо *awk*, либо более простая программа *expr*. В UNIX System V и, следовательно, в Solaris в Bourne shell был добавлен *expr*, который увеличивает скорость любого скрипта Bourne shell, который использует встроенную версию. Если используется полное имя пути, выполняется внешняя версия, что приводит к небольшому снижению скорости. Существует четыре типа операций, выполняемых *expr*:

```
Арифметическая
логическая
реляционная
строка
```

Большинство операторов очевидны:

Оператор	Тип	Meaining
+	Арифметика	Сложение
-	Арифметика	Вычитание
*	Арифметика	Умножение
/	Арифметика	Разделение
%	Арифметика	Остаток
=	Реляционный	Равно
>	Реляционный	Больше, чем
>=	Реляционный	Больше или равно
<	Реляционный	Менее
<=	Реляционный	Меньше или равно
!=	Реляционный	Не равно
|	Логическое	Или
&	Логическое	И
:	Строка	Сопоставьте или замените

Не так очевидно, как использовать эти операторы. Общая форма

оператор выражения выражение

Прежде всего, между операторами и выражениями должны быть пробелы. Приведенный ниже пример неверен:

выражение 2 +2

Правильная форма

выражение 2 + 2

который выводит "4" на стандартный вывод. Вторая проблема - это оболочка, которая обрабатывает некоторые символы как мета-символы. Поэтому, если вы используете любой из этих символов "*&|><()", вы должны поместить перед ними обратную косую черту или заключить их в кавычки. Примеры:

```
# 2 раза по 10
выражение 2 "*" 10
# или
выражение 2 \* 10
# сложное выражение
# true, если оба $ a и $ b больше нуля
expr $ a \'>' 0 \'&' $ b \'>' 0
# Но это выдает ошибку, если $ a или $ b не определены
# итак, вы должны использовать
выражение "$ a" '>' 0 '&' "$ b" '>' 0
```

Третья потенциальная проблема заключается в цитировании слишком большого количества символов. Выражение

выражение '2 + 2'

это не "4", а "2 + 2", длина которого составляет пять символов. Это позволяет сравнивать строки с пробелами:

выражение $ a = '2 + 2'

Если задано только одно выражение, то *expr* просто повторяет его.

expr нет

выводит слово "нет".

Другой потенциальной проблемой являются переменные, содержащие операторы. Следующий пример, который выглядит так, как будто он может проверять, равна ли переменная "=", генерирует ошибку:

```
A='='
# является ли $A '='?
expr $ A = '='
# НЕВЕРНО - синтаксическая ошибка - такая же, как 'expr = = ='
```

Исправление аналогично тому, которое использовалось для исправления *mecma*, т. Е. Добавьте дополнительный символ перед переменной:

выражение X $ A = X=

Арифметические операторы

B Expr используются 32-разрядные целые числа со знаком. Поэтому допустимы отрицательные числа. Обычно используются циклы:

```
# считайте от 1 до 10
A = 1
, пока [ $ A -le 10]
выполняет
echo $ A
A= `expr $ A + 1`
готово
```

Реляционные операторы

Реляционные операторы проверяют, являются ли оба выражения целыми числами. Если это так, сравнение является числовым. Если одно или несколько выражений являются нечисловыми, сравнение является лексическим. Поэтому вы можете сравнивать целые числа или строки. Запутанная часть реляционных операторов решает, как использовать информацию. Видите ли, операторы выводят на стандартный вывод, а также возвращают статус выхода. Помните, что значение true обычно ненулевое, но истинное состояние выхода равно нулю? Что ж, *expr* продолжает эту традицию. Рассмотрим следующий тест:

```
выражение 1 = 1
```

Это выводит "1" или "true" на стандартный вывод, но возвращает нулевое состояние выхода. Чтобы быть точным, если выражение имеет значение, которое не равно нулю и не равно нулевой строке, то статус выхода равен нулю. Статус выхода единицы является противоположным, т.е. выводом является нулевая строка или ноль. *Expr* вернет статус выхода, равный 2, если произойдет ошибка. Статус выхода можно использовать в простых тестах:

```
выражение $A = 1 > /dev/null && echo A = 1
```

Обратите внимание, что стандартный вывод должен был быть отброшен, иначе скрипт напечатал бы "1". Это выражение также может быть использовано в цикле:

```
A=1;
в то время как expr $A '<=' 10 >/dev/null
делает
A=`expr $A + 1`
echo $A
сделано
```

Реляционные операторы всегда выводят ноль или единицу.

Логические операторы

Два логических оператора работают со строками и целыми числами. Нулевая строка и ноль являются ложными. Все остальное верно. Я уже упоминал о статусе ранее. Это "побочный эффект", потому что программа также отправляет строку в стандартный вывод. Другими словами, разработчики программы *expr* попытались сделать ее как можно более универсальной. Рассмотрим следующее:

```
выражение $ a \ & $ b
```

Если любая из переменных является пустой строкой или нулем, программа выводит ноль. (Статус равен единице.) В противном случае программа завершает работу со статусом ноль, но не выводит единицу. вместо этого он выводит значение переменной "a". Представьте, что Гарри Оуэнс или Дон Пардо говорят

> *expr* не только возвращает статус, но и печатает! И это не просто выводит ноль или единицу. Он выводит либо ноль, либо фактическое значение выражения, экономя ваше время **и** деньги.

С другой стороны, они, возможно, просто чувствовали, что печать one приведет к потере информации без причины. Оператор "|" также возвращает ноль, если оба выражения равны нулю или представляют собой нулевую строку. В противном случае он возвращает либо первое выражение, либо второе. Точнее, *expr* просматривает первое выражение, и если оно не равно нулю или нулю, оно возвращает значение, в противном случае оно возвращает второе выражение, даже не проверяя его значение. Это небольшая путаница. Таблица может помочь. Первый столбец - это выражение. Второе - это распечатка. Третий столбец - это статус выхода:

Выражение	Вывод	Статус выхода
выражение " | "	0	1
выражение " | 0	0	1
выражение " | 1	1	0
выражение " | A	A	0
выражение 0 | "	0	1
выражение 0/1	1	0
выражение 0 | A	A	0
выражение 1 | что угодно	1	0
выражение A | anything	A	0
выражение " и что угодно	0	1
выражение 0 и что угодно	0	1
выражение 1 & "	0	1
выражение 1 и 0	0	1
выражение 1 и A	1	0
выражение A & "	0	1
выражение A & 0	0	1
выражение A & 1	A	0
выражение A & B	A	0

Строковый оператор

Последний оператор - это строковая операция. Это также самое сложное. Синтаксис

строка: *регулярное выражение*

Предполагается, что перед регулярным выражением стоит символ "^", поэтому оно всегда выравнивается по первому символу строки. Выходные данные - это количество символов, совпадающих в регулярном выражении. Это,

выражение abc: abc

выводит "3", в то время как

выражение abc: abd

выводит "0". Как вы можете видеть, если оно не совпадает, возвращается ноль, что эквивалентно "false". Если происходит совпадение, возвращается истинное значение (отличное от нуля), и вы знаете, сколько символов совпадает. Это можно использовать для подсчета символов, букв или цифр:

```
# выводит числовые символы в переменной a
выражением "$a": '.*'
# выводит количество строчных букв
выражения "$a": '[a-z] *'
# выводит количество строчных букв
выражения "$ a": '[0-9] *'
```

Помните, что регулярное выражение начинается с первого символа, поэтому в последнем примере будут учитываться только числа в начале переменной. Если переменная "a" имеет значение "123abc", выражение вернет значение, равное трем. *Expr* возвращает количество совпадающих символов. Поэтому *expr* можно использовать для определения местоположения первой буквы в строке:

```
выражение "$ x": '[^a-zA-Z]*[a-zA-Z]'
```

Скобки можно использовать в регулярном выражении, как команду substitute в *sed* . Если найдено совпадение, возвращается подстрока в круглых скобках. Следовательно, если переменная "a" имеет значение "123abc", две приведенные ниже команды выводят одну и ту же строку, которая является "abc".

```
# если $ a = 123abc, то выведите 'abc'
выражение "$a" : '[0-9]*\([ a-z]*\)'
# то же, что и выше
, echo $ a | sed 's / ^[0-9] *\([a-z] * \)/\1/'
```

Эта функция часто используется для извлечения части параметра командной строки. Вы должны знать, что перед круглыми скобками должна быть косая черта, и *expr* должен их видеть. То есть обратная косая черта не является сигналом для оболочки о том, что следующий символ не является мета-символом. Причина, по которой он должен быть исключен, заключается в том, чтобы соответствовать тому же синтаксису, что и *sed*, и т. Д. Предположим, у вас есть сценарий оболочки со следующей опцией

```
myscript -x123
```

Теперь предположим, что вы хотите извлечь число из параметра. Вы могли бы использовать сценарий *sed*, подобный приведенному выше. Вы также можете использовать команду *expr*:

```
x= `выражение "$1": '-x \(.*\)"
```

Команда *expr* также может использоваться для проверки того, является ли переменная числом:

```
echo "Введите число"
, прочитайте ans
number=`expr "$ans" : "\([0-9]*\)"`
if [ "$number" != "$ans" ]; затем
echo "Не число"
elif [ "$number" -eq 0 ]; затем
echo "Ничего не было введено"
, иначе
echo "$ number - это точное число"
fi
```

Вы можете комбинировать тесты. Например, следующее приведет к усечению строки до четырех символов:

```
выражение "$ x" : '\(....\)' \| "$ x"
```

Результаты каждого выражения - это другое выражение, поэтому вы можете комбинировать выражения. Следующее печатает 32:

```
выражение 2 + 3 \* 10
```

Команда *expr* может даже использоваться как простая версия basename . Следующее выведет последнюю часть имени файла:

```
выражение "$ x" : '.*/\(.*\)' '|' "$ x"
```

Однако в этом коде есть ошибка. Если переменная "x" равна "/", то это значение равно

```
expr / : '.*/\(.*\)' '|' /
```

Если вы помните, я указывал на эту проблему ранее. Оболочка предполагает, что косая черта является оператором, а не выражением. Он жалуется на синтаксическую ошибку. Решение состоит в том, чтобы поместить косые черты перед переменной:

```
выражение //$x : '.*/\(.*\)'
```

Приоритет операторов

Вы можете заключать выражения в круглые скобки для построения более сложных выражений. Убедитесь, что вы заключили их в кавычки, так как оболочка обрабатывает круглые скобки как специальные символы..

```
expr \( 2 + 3 \) \* 10
```

Скобка переопределяет приоритет по умолчанию, и результат составляет от 50 до 32. Эти круглые скобки отличаются от скобок, используемых в регулярных выражениях для выполнения операции замены. *Expr* должен видеть скобки без обратной косой черты. Обратная косая черта предназначена для оболочки. Вы могли бы использовать кавычки вместо обратной косой черты. Естественный приоритет сгруппирован по шести различным уровням, которые:

Наивысший приоритет
выражение: выражение
выражение * выражение
expr / expr
expr % expr
выражение + выражение
expr - выражение
expr = expr
expr > expr
expr >= expr
expr < expr
expr <= expr
выражение != выражение
выражение и выражение
expr | expr
Самый низкий приоритет

Расширения Berkeley

Версия *expr* для Беркли имеет три специальные функции:

сопоставьте

длину substr

Оператор *сопоставления* действует как двоеточие. Оператор подстроки действует подобно функции *awk*. Оператор length возвращает длину строки. Я предлагаю вам не использовать их по соображениям переносимости. Вы получите эту функциональность, если поместите каталог "/ usr / ucb" перед "/ usr / bin" в Solaris или "/ usr / 5bin" в SunOS. Не имеет значения, используете ли вы встроенную версию или внешнюю версию. Обе версии поддерживают расширения Berkeley, если сначала в пути указан каталог "ucb". "Ucb", кстати, означает Калифорнийский университет в Беркли.

Надеюсь, я дал вам некоторое представление о том, как использовать команду *expr*. Многие люди знают только о функциях числовых вычислений. Он не такой мощный, как *sed* или *awk*, но если он справится с этой задачей, вы можете получить прирост производительности, потому что вы используете встроенную функцию вместо внешней программы.

Bourne Shell -- проверка функций и аргументов

В оригинальной версии Bourne shell не было функций. Если вы хотели выполнить операцию более одного раза, вам нужно было либо дублировать код, либо создать новый сценарий оболочки. Существует небольшое наказание за каждый вызванный скрипт, поскольку должен быть создан другой процесс. Вы также должны знать, где находится скрипт, если его нет в пути.

Оболочка C имеет псевдонимы, но они ограничены одной строкой, а синтаксический анализ аргументов чрезвычайно запутан. Оболочка Bourne решила эту проблему с помощью концепции функций. Вот пример, который насчитывает от 1 до 10, увеличивая переменную A в функции:

```
#!/bin/sh
inc_A() {
# Увеличьте A на 1
 A= `выражение $ A + 1`
}
A= 1
пока [ $ A -le 10 ]
выполняет
echo $A
inc_A
готово
```

При желании вы можете определить ту же функцию в одной строке:

```
inc_A() { A=`выражение $A + 1`; }
```

Обратите внимание, что вам нужно добавить точку с запятой перед фигурной скобкой Керли, поскольку оболочка ожидает список между фигурными скобками Керли. Еще один способ запомнить это: "}" должна быть первой командой в строке.

Передача аргумента функции проста: она идентична механизму, используемому для передачи аргумента сценарию оболочки. Все, что вам нужно сделать, это помнить, что позиционные аргументы относятся к функции, а не к сценарию. То есть, если у вас есть следующий скрипт:

```
#!/bin/sh
функция() {echo $1;}
функция $2
```

и если вы вызываете его с двумя аргументами, скрипт печатает второй аргумент, потому что это первый аргумент в функции.

Передача значений по имени

Вы можете передавать имена переменных функциям. Однако это добавляет много сложностей. Вы должны обойти обычную оценку переменных оболочки. Кроме того, строки, подобные "$$", имеют особые значения. Вот функция, которая увеличивает указанную переменную

```
#!/bin/sh
inc() { eval $1=`выражение $ $1 + 1`; }
```

```
A = 10
вкл. A
# переменная A теперь имеет значение 11
```

Команда *eval* работает со строкой

```
a= `выражение $ a + 1`
```

и были добавлены обратные косые черты, чтобы оболочка не интерпретировала обратные кавычки и знак доллара слишком рано.

Выход из функции

Что произойдет, если вы выполните команду *exit* внутри функции? То же самое, как если бы вы выполнили его из любого другого скрипта. Он прерывает сценарий и передает значение вызывающему сценарию.

Предположим, вы хотите вернуть значение из функции? Если это переменная, просто поместите значение в переменную. Но это не то, о чем я говорю. Я показал вам, как использовать статус выхода в таких командах, как *if* и *while*. Что произойдет, если вы создадите функцию и используете ее в операторе *if*? У вас есть два варианта. Обычно функция возвращается со статусом завершения последней команды. Если вы хотите явно управлять значением, в Bourne shell есть специальная команда с именем *return*, которая устанавливает значение статуса в указанное значение. Если значение не указано, используется статус последней команды.

Вы можете написать простой скрипт для бесконечного цикла:

```
always_true() { возвращает 0;}

в то время
как
always_true повторяет: "Я просто не могу себя контролировать".
 повторите: "Кто-нибудь, остановите меня, пожалуйста!"
Выполнено
```

Помните, что состояние выхода, равное нулю, является истинным условием в программировании оболочки. Теперь, когда я описал функции, я покажу вам, как использовать их для анализа аргументов в сценарии оболочки.

Проверка количества аргументов

Допустим, у вас есть сценарий оболочки с тремя аргументами. Есть много способов решить эту проблему. Я постараюсь предоставить образец механизмов, чтобы вы могли использовать тот, который подходит для вашего приложения. Один из способов убедиться, что ваш скрипт будет работать без нужного количества аргументов, - использовать значения по умолчанию для переменных. В этом примере файл перемещается из одного

каталога в другой. Если вы не укажете никаких аргументов, он переместит файл "a.out" из текущего каталога в ваш домашний каталог:

```
#!/bin/sh -x
arg1=${1:-a.out}
arg2=${2:-`pwd`}
arg3=${3:- $HOME}
mv $ arg2/$arg1 $arg3
```

Короткое и простое. Не многие люди используют эту форму, и она может показаться сложной для начинающего программиста оболочки. Иногда это удобно. Если вы хотите потребовать, чтобы был указан первый аргумент, используйте форму, которая сообщает об ошибке, если аргумент отсутствует:

```
#!/bin/sh
file_to_be_moved="$1"
arg1=${file_to_be_moved:?"имя файла отсутствует"}
arg2=${2:-`pwd`}
arg3=${3:-$HOME}
mv $ arg2/$arg1 $arg3
```

Обратите внимание, как я добавил еще одну переменную с длинным именем. Без этого я бы получил следующую ошибку:

```
1: отсутствует имя файла
```

С дополнительной переменной я теперь получаю:

```
file_to_be_moved: имя файла отсутствует
```

Это лучше, но, возможно, немного непонятно. Возможно, лучший механизм:

```
#!/bin/sh
arg1="a.out"
arg2=`pwd`
arg3=$HOME
если [ $# -gt 3 ]
тогда
 :
fi

если [ $# -eq 0 ]
, то
echo должен указать хотя бы один аргумент
exit 1
, иначе, если [$ # -eq 1]
, то
arg1="$ 1";
иначе, если [ $ # -уравнение 2]
, то
arg1="$ 1";
 arg2="$ 2";
иначе, если [ $ # -уравнение 3]
, тогда
arg1="$ 1";
 arg2="$ 2";
 arg3="$ 3";
else
echo слишком много аргументов
exit 1
fi
```

```
mv $ arg2/$ arg1 $ arg3
```

Нажмите здесь, чтобы получить файл: ShCmdChk1.sh
или, возможно:

```
#!/bin/sh
arg1="a.out"
arg2= `pwd`
arg3= $HOME
если [$# -gt 3 ]
тогда
эхо слишком много аргументов
, выход 1
fi

если [ $# -eq 0 ]
, то
echo должен указать хотя бы один аргумент
exit 1
fi
[ $ # -ge 1 ] && arg1= $ 1 # сделайте это, если 1, 2 или 3 аргумента
[ $# -ge 2 ] && arg2= $ 2 # сделайте это, если 2 или 3 аргумента
[ $# -ge 3 ] && arg3= $ 3 # сделайте это, если 3 аргумента

mv $ arg2/$ arg1 $ arg3
```

Нажмите здесь, чтобы получить файл: ShCmdChk2.sh
Другой способ решить проблему - использовать команду shift . На этот раз я добавлю функцию:

```
#!/bin/sh

использование() {
echo `базовое имя $0`: ОШИБКА: $* 1> &2
 использование echo: `базовое имя $ 0` 'имя файла [fromdir] [todir]' 1> &2
 выход 1
}
arg1="a.out"
arg2= `pwd`
arg3= $HOME

[ $# -gt 3 -o $ # -lt 1 ] && использование "Неправильного количества аргументов"

arg1= $ 1; сдвиг
[ $# -gt 0 ] && { arg2=$1;сдвиг;}
[ $# -gt 0 ] && { arg3=$1;сдвиг;}

mv $ arg2/$ arg1 $ arg3
```

Нажмите здесь, чтобы получить файл: ShCmdChk3.sh
Вот еще один вариант использования команды *case*:

```
#!/bin/sh

использование() {
echo `базовое имя $0`: ОШИБКА: $* 1> &2
```

```
 использование echo: `базовое имя $ 0` 'имя файла [fromdir] [todir]' 1> &2
 выход 1
}
arg1="a.out"
arg2= `pwd`
arg3= $HOME

case $ # в
 0) использование "должно содержать хотя бы один аргумент";;
 1) arg1 = $ 1;;
 2) arg1 = 1 доллар; arg2 = 2 доллара;;
 3) arg1 = 1 доллар; arg2 = 2 доллара; arg3 = 3 доллара;;
 *) использование "слишком много аргументов";;

esac
mv $ arg2 / $ arg1 $ arg3
```

Нажмите здесь, чтобы получить файл: ShCmdChk4.sh
Как вы можете видеть, существует множество вариантов. Обратите внимание на использование "basename $ 0" в функции *использования*. Очень важно указывать имя скрипта в сообщении об ошибке. Может быть очень неприятно получать отчет об ошибке, но понятия не иметь, где ошибка. Когда скрипты вызывают другие скрипты хаотично, поиск виновника может занять много времени. Мне также нравится помещать функцию *использования* в начало скрипта, чтобы помочь кому-то быстро научиться использовать скрипт, изучив код. Также обратите внимание на "1> & 2" при сообщении об ошибке. Это приводит к тому, что вывод переходит к стандартной ошибке.

Соглашения UNIX для аргументов командной строки

В мире UNIX существуют стандарты для аргументов командной строки. Некоторые команды, такие как *tar*, *find* и *dd*, не соответствуют этим соглашениям. Я мог бы объяснить, почему они не следуют соглашению, но это древняя история. Нужно иметь дело с жизненными разочарованиями, кузнечик.

Если возможно, вы должны согласиться с соглашениями. Большинство сценариев оболочки не соответствуют всем соглашениям. Возможно, курс повышения квалификации хорошая идея. Все еще здесь? Я буду краток. Необязательные аргументы всегда начинаются с дефиса и состоят из одной буквы или цифры. Если после параметра следует значение, пробел является необязательным. Например, если переменная "o" принимает значение, следующие два примера должны делать то же самое:

```
program -ofile
program -o файл
```

Параметры, которые не принимают аргументы, можно комбинировать с дефисом. Порядок не имеет значения. Поэтому следующие примеры должны быть эквивалентными:

```
программа -a -b -c
программа -c -b -a
программа -ab -c
программа -cb -a
программа -cba
программа -abc
```

Если указана опция, которая не соответствует списку предложенных опций, должна возникнуть ошибка. Другим популярным соглашением является соглашение о дефисе-дефисе. Если вы хотите передать имя файла скрипту, начинающемуся с дефиса, используйте дефис-дефис перед ним:

программа - -a

Проверка наличия необязательных аргументов

Давайте предположим, что нам нужен скрипт, который использует буквы от a до c в качестве однобуквенных опций и "-o" в качестве опции, требующей значения. Давайте также предположим, что после параметров есть любое количество аргументов. Простым решением может быть:

```
#!/bin/sh

использование() {
echo `базовое имя $0`: ОШИБКА: $* 1> &2
 использование эхо-сигнала: `базовое имя $ 0` '[-a] [-b] [-c] [-o файл]
 [файл ...]' 1>&2
 выход 1
}

a = b = c = o =

в то время как:
выполните
регистр "$ 1" в
 -a) a = 1;;
 -b) b = 1;;
 -c) c = 1;;
 -o) сдвиг; o = "$ 1";;
 --) сдвиг; перерыв;;
 -*) использование "неверного аргумента $ 1";;
 *) перерыв;;

сдвиг esac
выполнен
# остальная часть скрипта...
```

Нажмите здесь, чтобы получить файл: ShCmdArgs3.sh
Этот скрипт не позволяет комбинировать несколько однобуквенных вариантов с одним дефисом. Кроме того, после параметра "-o" обязательно ставится пробел. Эти параметры могут быть исправлены, но за определенную плату. Код становится намного сложнее. Однако это работает:

```
#!/bin/sh

использование() {
echo `базовое имя $0`: ОШИБКА: $* 1> &2
 использование эха: `базовое имя $ 0` '[-[abc]] [-o file]' '[file ...]' 1> &2
 выход 1
}


внутри () {
# эта функция возвращает значение TRUE, если $ 2 находится внутри $ 1
# Я буду использовать оператор case, потому что это встроенный в оболочку,
# и быстрее.
# Я мог бы использовать grep:
# echo $1 | grep -s "$2" >/dev/null
# или это
# echo $ 1 | grep -qs "$ 2"
# или выражение:
# expr "$1" : ".*$2" >/ dev/null && возвращает 0 # true
# но для этого случая не требуется другой процесс
```

```
оболочки "$ 1" в
 * $ 2 *) возвращает 0;;
esac
возвращает 1;
}


выполненные опции=
дополнительные опции() {
 # верните true(0), если остались варианты для анализа
 # в противном случае верните false

 # проверьте проверку флага 'short-circuit'
$done_options && возвращает 1 # true

 # сколько аргументов осталось?
 [ $# -eq 0 ] && возвращает 0

 # начинается ли следующий аргумент с дефиса
внутри "$ 1" '-' && возвращает 0;

 # в противном случае верните false
верните 1 # false
}
a = b = c = o =

в то время как more_options "$ 1"
выполняют
регистр "$ 1" в
 --) выполненные варианты =1;;
 -[abc]*)
 внутри "$ 1" a && a = 1;
 внутри "$ 1" b & & b = 1;
 внутри "$ 1" c && c = 1;
 внутри "$ 1" "[d-zA-Z]" &&
использование "неизвестная опция в $ 1";
 ;;
 -o?*)
 # необходимо извлечь строку из аргумента
o=`expr "$1": '-o(.*)"
 ;;
 -o)
 [ $# -gt 1 ] || использование "-o требует значения";
 сдвиг
o = "$ 1";;
 -*) использование "неизвестный вариант $ 1";;
 esac
shift # каждый раз отключайте опцию
готово
# продолжить работу со сценарием
# ...
```

Нажмите здесь, чтобы получить файл: ShCmdArgs4.sh
В одной из моих ранних версий этого скрипта вместо оператора *case* использовался *expr*. Это позволило создать сценарий меньшего размера и короче, но команда *expr* является внешней командой, и выполнение сценария заняло больше времени. Эта версия более эффективна, но более сложна. Вопрос, который вы должны задать, стоит ли оно того?
Есть и другой вариант. Многие системы UNIX имеют программу *getopt*. Это позволяет обрабатывать параметры командной строки в соответствии со стандартными соглашениями. Вот как его использовать:

```
#!/bin/sh

использование() {
echo `базовое имя $0`: ОШИБКА: $* 1> &2
 использование эха: `базовое имя $ 0` '[-[abc]] [-o file]' '[file ...]' 1> &2
 выход 1
}

set -- `getopt "abco:" "$@"` || использование

a = b = c = o=
в то время как:
выполните
регистр "$ 1" в
 -a) a = 1;;
 -b) b = 1;;
 -c) c = 1;;
 -o) сдвиг; o = "$ 1";;
 --) перерыв;;
 esac
сдвиг
выполнен
сдвиг # избавиться от --
# остальная часть скрипта...
```

Нажмите здесь, чтобы получить файл: ShCmdArgs.sh
команда getopt выполняет большую часть работы. Он автоматически сообщает о недопустимых аргументах. Первый аргумент задает юридические аргументы, и любая буква с двоеточием после нее требует значения. Вы должны отметить, что он всегда ставит двойной дефис после параметров и перед аргументами. Я использовал команду *shift*, чтобы избавиться от нее. У команды *getopt* есть один плохой побочный эффект. Если у вас есть какие-либо нулевые аргументы или аргументы с пробелами, команда *getopt* потеряет эту информацию. Другие варианты этого не делают.
Я дал вам очень полную коллекцию скриптов, которые вы можете изменять в соответствии с вашими потребностями. Существует много других способов решения проблем, представленных здесь. Не существует единственного наилучшего метода для использования. Надеюсь, теперь вы знаете несколько вариантов и можете выбрать наилучший для вас метод.

Управление заданиями

Я добавил этот раздел позже, потому что он настолько мощный, но сбивает с толку некоторых людей.
Bourne shell имеет очень мощный способ управления фоновыми процессами в скрипте. Вот несколько примеров.
Этот скрипт выполняет три задания. Если родительское задание получает сигнал HUP или TERM, оно отправляет его дочерним процессам. Поэтому, если вы прервете родительский процесс, он передаст этот сигнал дочерним процессам.

```
PIDS =
program1 & PIDS="$PIDS $!"
program2 & PIDS="$PIDS $!"
program3 & PIDS="$PIDS $!"
ловушка "убить -1 15 $ PIDS" 1 15
```

Вот еще один пример, который позволяет запускать три процесса, но завершать их по истечении 30 секунд.

```
 MYID=$$
 PIDS =
 (sleep 30; убить -1 $ MYID) &
(sleep 5; echo A) & PIDS="$ PIDS $!"
 (sleep 10;echo B) & PIDS="$PIDS $!"
 (sleep 50; echo C) & PIDS="$ PIDS $!"
 ловушка "тайм-АУТ эха; убить $ PIDS" 1
эхо, ожидающее $ PIDS
ожидание $ PIDS
эхо, все в порядке
```

Вот еще один пример, в котором вы запускаете "prog1" и "prog2" в фоновом режиме и запускаете "prog3" несколько раз, пока "prog1" или "prog2" не завершатся.

```
 #!/bin/sh
 MYPID=$$
 сделано = 0
ловушка "сделано = 1" USR1
 (prog1;echo prog1 выполнено; kill -USR1 $$) & pid1 = $!
 ловушка "сделано = 1" USR2
 (prog2;echo prog2 выполнено; kill -USR2 $$) & pid2 = $!
 ловушка "убить -1 $ pid1 $ pid2" 1 15

 в то время как [ "$ done" -eq 0 ]


готово ли prog3
```

Я мог бы упростить приведенный выше пример, объединив USR1 и USR2, но таким образом вы могли бы изменить его, чтобы проверить выполнение обоих заданий, а не только одного.

Также обратите внимание на ловушку сигнала 1, которая позволяет вам завершить родительский элемент и заставить его завершить дочерние элементы. Вы не хотите, чтобы долго выполняемые задания зависали, особенно если вы отлаживаете сценарий.

Вы также можете использовать его для передачи других сигналов дочерним процессам и заставить их делать что-то особенное. Например, они могут ждать, пока не получат сигнал. Таким образом, вы можете запускать все процессы одновременно (почти).
Этот документ был переведен troff2html v0.21 22 сентября 2001 года.