Московский государственный технический университет

им. Н. Э. Баумана

Земляков В. Н.

Основы программирования на языке Shell

Учебное пособие

Москва – 2006г.

Содержание.

Введение 5

Зачем необходимо знание языка Shell? 5
Запуск сценария 7

Служебные символы 7

Переменные и параметры 20

Подстановка переменных 20
Присваивание значений переменным 22
Переменные Bash не имеют типа 23
Специальные типы переменных 24
Кавычки 23
Завершение и код завершения 24

Проверка условий 31

Конструкции проверки условий 31
Else if и elif 33
Операции проверки файлов 35
Операции сравнения 37
Вложенные условные операторы if/then 40

Операции 40

Операторы 40
Числовые константы 43

Переменные 44

Внутренние переменные 44
Позиционные переменные 44
Прочие специальные переменные 44
Работа со строками 49
Извлечение подстроки 49
Удаление части строки 49
Замена подстроки 49
Подстановка параметров 53
$RANDOM: генерация псевдослучайных целых чисел 58
Двойные круглые скобки 59

Циклы и ветвления 60

Циклы 60
Вложенные циклы 63
Управление ходом выполнения цикла 64
Операторы выбора 64

Подстановка команд 66

Арифметические подстановки ... 69
Перенаправление ввода/вывода ... 69
Использование exec ... 72

Внутренние команды ... 73

Файловые системы ... 77
Переменные ... 78
Управление сценарием ... 81
Команды ... 82

Команды управления заданиями ... 83

Внешние команды, программы и утилиты ... 85

Базовые команды ... 85
Более сложные команды ... 87
Команды для работы с датой и временем ... 90
Команды обработки текста ... 91
Команды для работы с файлами и архивами ... 98
Команды для работы с сетью ... 103
Команды управления терминалом ... 105
Команды выполнения математических операций ... 106
Прочие команды ... 108
Команды системного администрирования ... 109

Регулярные выражения ... 118

Краткое введение в регулярные выражения ... 118

Globbing -- Подстановка имен файлов ... 121

Подоболочки, или Subshells ... 121

Функции ... 124

Сложные функции и сложности с функциями ... 124
Локальные переменные ... 128

Списки команд ... 129

Массивы ... 131

/dev и /proc ... 131

Отладка сценариев ... 137

Необязательные параметры (ключи) ... 140

Введение.

Shell -- это командная оболочка. Но это не просто промежуточное звено между пользователем и операционной системой, это еще и мощный язык программирования. Программы на языке shell называют *сценариями*, или *скриптами*. Фактически, из скриптов доступен полный набор команд, утилит и программ UNIX. Если этого недостаточно, то к вашим услугам внутренние команды shell -- условные операторы, операторы циклов и пр., которые увеличивают мощь и гибкость сценариев. Shell-скрипты исключительно хороши при программировании задач администрирования системы и др., которые не требуют для своего создания полновесных языков программирования.

Зачем необходимо знание языка Shell?

Знание языка командной оболочки является залогом успешного решения задач администрирования системы. Даже если вы не предполагаете заниматься написанием своих сценариев. Во время загрузки Linux выполняется целый ряд сценариев из `/etc/rc.d`, которые настраивают конфигурацию операционной системы и запускают различные сервисы, поэтому очень важно четко понимать эти скрипты и иметь достаточно знаний, чтобы вносить в них какие либо изменения.

Язык сценариев легок в изучении, в нем не так много специфических операторов и конструкций. Синтаксис языка достаточно прост и прямолинеен, он очень напоминает команды, которые приходится вводить в командной строке. Короткие скрипты практически не нуждаются в отладке, и даже отладка больших скриптов отнимает весьма незначительное время.

Shell - скрипты очень хорошо подходят для быстрого создания прототипов сложных приложений, даже не смотря на ограниченный набор языковых конструкций и определенную "медлительность". Такая метода позволяет детально проработать структуру будущего приложения, обнаружить возможные "ловушки" и лишь затем приступить к кодированию на C, C++, Java, или Perl.

Скрипты возвращают нас к классической философии UNIX - "разделяй и властвуй" т.е. разделение сложного проекта на ряд простых подзадач. Многие считают такой подход наилучшим или, по меньшей мере, наиболее эстетичным способом решения возникающих проблем, нежели использование нового поколения языков - "все в одном", таких как Perl.

Для каких задач неприменимы скрипты

- для ресурсоемких задач, особенно когда важна скорость исполнения (поиск, сортировка и т.п.)
- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел (для таких задач лучше использовать C++ или FORTRAN)
- для кросс-платформенного программирования (для этого лучше подходит язык C)
- для сложных приложений, когда структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.)
- для целевых задач, от которых может зависеть успех предприятия.
- когда во главу угла поставлена безопасность системы, когда необходимо обеспечить целостность системы и защитить ее от вторжения, взлома и вандализма.
- для проектов, содержащих компоненты, очень тесно взаимодействующие между собой.
- для задач, выполняющих огромный объем работ с файлами
- для задач, работающих с многомерными массивами

- когда необходимо работать со структурами данных, такими как связанные списки или деревья
- когда необходимо предоставить графический интерфейс с пользователем (GUI)
- когда необходим прямой доступ к аппаратуре компьютера
- когда необходимо выполнять обмен через порты ввода-вывода или сокеты
- когда необходимо использовать внешние библиотеки
- для "закрытых" программ (скрипты представляют из себя исходные тексты программ, доступные для всеобщего обозрения)

Если выполняется хотя бы одно из вышеперечисленных условий, то вам лучше обратиться к более мощным скриптовым языкам программирования, например Perl, Tcl, Python, Ruby или к высокоуровневым компилирующим языкам -- C, C++ или Java. Но даже в этом случае, создание прототипа приложения на языке shell может существенно облегчить разработку.

Название BASH - это аббревиатура от "Bourne-Again Shell" и игра слов от, ставшего уже классикой, "Bourne Shell" Стефена Бурна (Stephen Bourne). В последние годы BASH достиг такой популярности, что стал стандартной командной оболочкой *de facto* для многих разновидностей UNIX. Большинство принципов программирования на BASH одинаково хорошо применимы и в других командных оболочках, таких как Korn Shell (ksh), от которой Bash позаимствовал некоторые особенности, и C Shell и его производных.

Далее, на наших занятиях разберем большое количество примеров скриптов, иллюстрирующих возможности shell. Все примеры - работающие. Они были протестированы, причем некоторые из них могут пригодиться в повседневной работе.

В простейшем случае, скрипт - это ни что иное, как простой список команд системы, записанный в файл. Создание скриптов поможет сохранить ваше время и силы, которые тратятся на ввод последовательности команд всякий раз, когда необходимо их выполнить.

Пример cleanup: Сценарий очистки лог-файлов в /var/log.

```
# cleanup
# Для работы сценария требуются права root.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Лог-файлы очищены."
```

Здесь нет ничего необычного, это простая последовательность команд, которая может быть набрана в командной строке с консоли. Преимущество размещения последовательности команд в скрипте состоит в том, что вам не придется всякий раз набирать эту последовательность вручную. Кроме того, скрипты легко могут быть модифицированы или обобщены для разных применений.

Если файл сценария начинается с последовательности **#!**, которая в мире UNIX называется *sha-bang*, то это указывает системе какой интерпретатор следует использовать для исполнения сценария. Это двухбайтовая последовательность, или - специальный маркер, определяющий тип сценария, в данном случае - сценарий командной. Более точно, *sha-bang* определяет интерпретатор, который вызывается для исполнения сценария, это может быть командная оболочка (shell), иной интерпретатор или утилита.

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

```
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Каждая, из приведенных выше сигнатур, приводит к вызову различных интерпретаторов, будь то `/bin/sh` - командный интерпретатор по-умолчанию (**bash** для Linux-систем), либо иной. При переносе сценариев с сигнатурой **`#!/bin/sh`** на другие UNIX системы, где в качестве командного интерпретатора задан другой shell, вы можете лишиться некоторых особенностей, присущих bash. Поэтому такие сценарии должны быть POSIX совместимыми.

Обратите внимание на то, что сигнатура должна указывать правильный путь к интерпретатору, в противном случае вы получите сообщение об ошибке - как правило это "Command not found".

Сигнатура #! может быть опущена, если вы не используете специфичных команд. Еще раз заметим, что сигнатура **`#!/bin/sh`** вызывает командный интерпретатор по умолчанию -- **`/bin/bash`** в Linux-системах.

Запуск сценария.

Запустить сценарий можно командой **`sh scriptname`** или **`bash scriptname`**. (Не рекомендуется запуск сценария командой **`sh <scriptname>`**, поскольку это запрещает использование устройства стандартного ввода `stdin` в скрипте). Более удобный вариант - сделать файл скрипта исполняемым, командой chmod.

Это:

`chmod 555 scriptname` (выдача прав на чтение/исполнение любому пользователю в системе)

или

`chmod +rx scriptname` (выдача прав на чтение/исполнение любому пользователю в системе)

`chmod u+rx scriptname` (выдача прав на чтение/исполнение только "владельцу" скрипта)

После того, как вы сделаете файл сценария исполняемым, вы можете запустить его примерно такой командой **`./scriptname`**. Если, при этом, текст сценария начинается с корректной сигнатуры ("sha-bang"), то для его исполнения будет вызван соответствующий интерпретатор.

И наконец, завершив отладку сценария, вы можете поместить его в каталог **`/usr/local/bin`** (естественно, что для этого вы должны обладать правами root), чтобы сделать его доступным для себя и других пользователей системы. После этого сценарий можно вызвать, просто напечатав название файла в командной строке и нажав клавишу **[ENTER]**.

Служебные символы.

Служебные символы, используемые в текстах сценариев.

**Комментарии.** Строки, начинающиеся с символа # (за исключением комбинации #!) -- являются комментариями.

```
# Эта строка - комментарий.
```

Комментарии могут располагаться и в конце строки с исполняемым кодом.

```
echo "Далее следует комментарий." # Это комментарий.
```

Комментариям могут предшествовать пробелы (пробел, табуляция).

```
        # Перед комментарием стоит символ табуляции.
```

Исполняемые команды не могут следовать за комментарием в той же самой строке. Пока что еще не существует способа отделения комментария от "исполняемого кода", следующего за комментарием в той же строке.

Само собой разумеется, экранированный символ # в операторе **echo** не воспринимается как начало комментария. Более того, он может использоваться в операциях подстановки параметров и в константных числовых выражениях.

```
echo "Символ # не означает начало комментария."
echo 'Символ # не означает начало комментария.'
echo Символ \# не означает начало комментария.
echo А здесь символ # означает начало комментария.
echo ${PATH#*:}       # Подстановка -- не комментарий.
echo $(( 2#101011 ))  # База системы счисления -- не комментарий.
# Спасибо, S.C.
Кавычки " ' и \  экранируют действие символа #.
```

В операциях поиска по шаблону символ # так же не воспринимается как начало комментария.

- **Разделитель команд.** [Точка-с-запятой] Позволяет записывать две и более команд в одной строке.

```
echo hello; echo there
```

Следует отметить, что символ ";" иногда так же как и # необходимо экранировать.

- **Ограничитель в операторе выбора `case` .** [Двойная-точка-с-запятой]

```
case "$variable" in
abc)  echo "$variable = abc" ;;
xyz)  echo "$variable = xyz" ;;
esac
```

- **Команда "точка".** Эквивалент команды source. Это встроенная команда bash.

"точка" может являться частью имени файла . Если имя файла начинается с точки, то это "скрытый" файл, т.е. команда ls при обычных условиях его не отображает.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 student      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 student       877 Dec 17  2000 employment.add
bash$ ls -al
total 14
 drwxrwxr-x    2 student  student      1024 Aug 29 20:54 ./
 drwx------   52 student  student      3072 Aug 29 20:51 ../
 -rw-r--r--    1 student  student      4034 Jul 18 22:04
data1.addressbook
 -rw-r--r--    1 student  student       877 Dec 17  2000 employment.add
 -rw-rw-r--    1 student  student         0 Aug 29 20:54 .hidden-file
```

Если подразумевается имя каталога, то *одна точка* означает текущий каталог и *две точки* -- каталог уровнем выше, или родительский каталог.

```
bash$ pwd
/home/student/projects

bash$ cd .
```

```
bash$ pwd
/home/student/projects

bash$ cd ..
bash$ pwd
/home/student/
```

Символ *точка* довольно часто используется для обозначения каталога назначения в операциях копирования/перемещения файлов.

```
bash$ cp /home/student/current_work/junk/* .
```

Символ "точка" в операциях поиска. При выполнении поиска по шаблону , в регулярных выражениях, символ "точка" обозначает одиночный символ.

\\ Двойные кавычки . В строке *"STRING"*, ограниченной двойными кавычками не выполняется интерпретация большинства служебных символов, которые могут находиться в строке.

\ Одинарные кавычки . [Одинарные кавычки] *'STRING'* экранирует все служебные символы в строке *STRING*. Это более строгая форма экранирования.

, Запятая . Оператор **запятая** используется для вычисления серии арифметических выражений. Вычисляются все выражения, но возвращается результат последнего выражения.

```
let "t2 = ((a = 9, 15 / 3))"  # Присваивает значение переменной "a" и
вычисляет "t2".
```

\ escape. [обратный слэш] Комбинация **\x** "экранирует" символ *X*. Аналогичный эффект имеет комбинация с "одинарными кавычками", т.е. *'X'*. Символ \ может использоваться для экранирования кавычек " и '.

/ Разделитель, используемый в указании пути к каталогам и файлам. [слэш] Отделяет элементы пути к каталогам и файлам (например `/home/student/projects/Makefile`).

В арифметических операциях -- это оператор деления.

**` Подстановка команд.** [обратные кавычки] Обратные кавычки могут использоваться для записи в переменную команды *`command`*.

: Пустая команда. [двоеточие] Это эквивалент операции "NOP" (*no op*, нет операции). Может рассматриваться как синоним встроенной команды **true.** Команда ":" так же является встроенной командой Bash, которая всегда возвращает "true".

Бесконечный цикл:

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done
# То же самое:
    while true
    do
      ...
    done
```

Символ-заполнитель в условном операторе if/then:

```
if condition
```

```
        then : # Никаких действий не
        else
                    take-some-action
fi
```

Как символ-заполнитель в операциях, которые предполагают наличие двух операндов.

```
: ${username=`whoami`}
# ${username=`whoami`}   без символа : выдает сообщение об ошибке,
#                        если "username" не является командой...
```

В операциях замены подстроки с подстановкой значений переменных.

В комбинации с оператором > (оператор перенаправления вывода), усекает длину файла до нуля. Если указан несуществующий файл -- то он создается.

```
: > data.xxx   # Файл "data.xxx" -- пуст

# Тот же эффект имеет команда cat /dev/null >data.xxx
# Однако в данном случае не производится создание нового процесса, поскольку ":" является встроенной командой.
```

В комбинации с оператором >> -- оператор перенаправления с добавлением в конец файла и обновлением времени последнего доступа (**`: >> new_file`**). Если задано имя несуществующего файла, то он создается. Эквивалентно команде **`touch`**.

Символ : может использоваться для создания комментариев, хотя и не рекомендуется. Если строка комментария начинается с символа #, то такая строка не проверяется интерпретатором на наличие ошибок. Однако в случае оператора : это не так.

```
: Это комментарий, который генерирует сообщение об ошибке, ( if [ $x -eq 3] ).
```

Символ ":" может использоваться как разделитель полей в `/etc/passwd` и переменной **$PATH.**

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

! **Инверсия (или логическое отрицание) используемое в условных операторах.** Оператор ! инвертирует код завершения команды, к которой он применен. Так же используется для логического отрицания в операциях сравнения, например, операция сравнения "равно" (=), при использовании оператора отрицания, преобразуется в операцию сравнения -- "не равно" (!=). Символ ! является зарезервированным ключевым словом BASH.

В некоторых случаях символ ! используется для косвенного обращения к переменным.

Кроме того, из *командной строки* оператор ! запускает *механизм историй* Bash. Примечательно, что этот механизм недоступен из сценариев (т.е. исключительно из командной строки).

***** **Символ-шаблон.** [звездочка] Символ * служит "шаблоном" для подстановки в имена файлов. Одиночный символ * означает любое имя файла в заданном каталоге.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

В регулярных выражениях токен * представляет любое количество (в том числе и 0) символов.

Арифметический оператор. В арифметических выражениях символ * обозначает операцию умножения.

Двойная звездочка (два символа звездочки, следующих подряд друг за другом -- **), обозначает операцию возведения в степень.

? **Оператор проверки условия.** В некоторых выражениях символ ? служит для проверки выполнения условия.

В конструкциях с двойными скобками, символ ? подобен трехместному оператору языка C.

В выражениях с подстановкой параметра, символ ? проверяет -- установлена ли переменная.

символ-шаблон. Символ ? обозначает одиночный символ при подстановке в имена файлов. В регулярных выражениях служит для обозначения одиночного символа.

$ **Подстановка переменной.**

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

Символ $, предшествующий имени переменной, указывает на то, что будет получено *значение* переменной.

end-of-line (конец строки). В регулярных выражениях, символ "$" обозначает конец строки.

${} **Подстановка параметра.**

$*, $@ **параметры командной строки.**

$? **код завершения.** Переменная $? хранит код завершения последней выполненной команды, функции или сценария.

$$ **id процесса.** *Переменная $$* хранит *id процесса* сценария.

() **группа команд.**

```
(a=hello; echo $a)
```

Команды, заключенные в *круглые скобки* исполняются в дочернем процессе -- subshell-е.

Переменные, создаваемые в дочернем процессе не видны в "родительском" сценарии. Родительский процесс-сценарий, не может обращаться к переменным, создаваемым в дочернем процессе.

```
a=123
( a=321; )

echo "a = $a"   # a = 123
# переменная "a" в скобках подобна локальной переменной.
```

инициализация массивов.

```
Array=(element1 element2 element3)
```

{ } Фигурные скобки.

```
grep Linux file*.{txt,htm*}
# Поиск всех вхождений слова "Linux"
# в файлах "fileA.txt", "file2.txt", "fileR.html", "file-87.htm", и пр.
```

Команда интерпретируется как список команд, разделенных точкой с запятой, с вариациями, представленными в *фигурных скобках*. При интерпретации имен файлов (подстановка) используются параметры, заключенные в фигурные скобки.

Блок кода. [фигурные скобки] Известен так же как "вложенный блок", эта конструкция, фактически, создает анонимную функцию. Однако, в отличии от обычных функций, переменные, создаваемые во вложенных блоках кода, доступны объемлющему сценарию.

```
bash$ { local a; a=123; }
bash: local: can only be used in a function


a=123
{ a=321; }
echo "a = $a"   # a = 321   (значение, присвоенное во вложенном блоке
кода)

# Спасибо, S.C.
```

Код, заключенный в фигурные скобки, может выполнять перенаправление ввода-вывода.

Пример. Вложенные блоки и перенаправление ввода-вывода.

```
#!/bin/bash
# Чтение строк из файла /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "Первая строка в $File :"
echo "$line1"
echo
echo "Вторая строка в $File :"
echo "$line2"

exit 0
```

Пример. Сохранение результата исполнения вложенного блока в файл.

```
#!/bin/bash
# rpm-check.sh

# Запрашивает описание rpm-архива, список файлов, и проверяется возмож-
ность установки.
# Результат сохраняется в файле.
#
# Этот сценарий иллюстрирует порядок работы со вложенными блоками кода.
```

```
SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` rpm-file"
  exit $E_NOARGS
fi

{
  echo
  echo "Описание архива:"
  rpm -qpi $1       # Запрос описания.
  echo
  echo "Список файлов:"
  rpm -qpl $1       # Запрос списка.
  echo
  rpm -i --test $1  # Проверка возможности установки.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 может быть установлен."
  else
    echo "$1 -- установка невозможна!"
  fi
  echo
} > "$1.test"       # Перенаправление вывода в файл.

echo "Результаты проверки rpm-архива находятся в файле $1.test"

# За дополнительной информацией по ключам команды rpm см. man rpm.

exit 0
```

В отличие от групп команд в (круглых скобках), описанных выше, вложенные блоки кода, заключенные в {фигурные скобки} исполняются в пределах того же процесса, что и сам скрипт (т.е. не вызывают запуск дочернего процесса -- subshell).

{ } \ ; **pathname -- полное имя файла (т.е. путь к файлу и его имя).** Чаще всего используется совместно с командой **find.**

Обратите внимание на то, что символ ";", которым завершается ключ `-exec` команды **find**, экранируется обратным слешем. Это необходимо, чтобы предотвратить его интерпретацию.

[ ] **test.**

Проверка истинности выражения, заключенного в квадратные скобки **[ ]**. Примечательно, что **[** является частью встроенной команды **test** (и ее синонимом), И *не* имеет никакого отношения к "внешней" утилите `/usr/bin/test`.

элемент массива.

При работе с массивами в квадратных скобках указывается порядковый номер того элемента массива, к которому производится обращение.

```
Array[1]=slot_1
echo ${Array[1]}
```

диапазон символов.

В регулярных выражениях, в квадратных скобках задается диапазон искомых символов.

[[ ]] **test.**

Проверка истинности выражения, заключенного между [[ ]] (зарезервированное слово интерпретатора).

( ) **двойные круглые скобки.**

Вычисляется целочисленное выражение, заключенное между двойными круглыми скобками (( )).

> &> >& >> < **Перенаправление**.

Конструкция **scriptname >filename** перенаправляет вывод `scriptname` в файл `filename`. Если файл `filename` уже существовал, то его прежнее содержимое будет утеряно.

Конструкция **command &>filename** перенаправляет вывод команды `command`, как со `stdout`, так и с `stderr`, в файл `filename`.

Конструкция **command >&2** перенаправляет вывод со `stdout` на `stderr`.

Конструкция **scriptname >>filename** добавляет вывод `scriptname` к файлу `filename`. Если задано имя несуществующего файла, то он создается.

подстановка процесса.

(command)>

<(command)

В, символы "<" и ">" обозначают операции.

<< **перенаправление ввода на встроенный документ.**

<, > Посимвольное **ASCII-сравнение.**

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
  echo "Не смотря на то, что в словаре слово $veg1 предшествует слову
$veg2,"
  echo "это никак не отражает мои кулинарные предпочтения."
else
  echo "Интересно. Каким словарем вы пользуетесь?"
fi
```

\<, \> **Границы отдельных слов в регулярных выражениях.**

```
bash$ grep '\<the\>' textfile
```

| **Конвейер.** Передает вывод предыдущей команды на ввод следующей или на вход командного интерпретатора shell. Этот метод часто используется для связывания последовательности команд в единую цепочку.

```
echo ls -l | sh
#  Передает вывод "echo ls -l" команлному интерпретатору shell,
#+ тот же результат дает простая команда "ls -l".


cat *.lst | sort | uniq
# Объединяет все файлы ".lst", сортирует содержимое и удаляет повторяю-
щиеся строки.
```

Конвейеры (еще их называют каналами) -- это классический способ взаимодействия процессов, с помощью которого `stdout` одного процесса перенаправляется на `stdin` другого. Обычно используется совместно с командами вывода, такими как

cat или echo, от которых поток данных поступает в "фильтр" (команда, которая на входе получает данные, преобразует их и обрабатывает).

```
cat $filename | grep $search_word
```

В конвейер могут объединяться и сценарии на языке командной оболочки.

```
#!/bin/bash
# uppercase.sh : Преобразование вводимых символов в верхний регистр.

tr 'a-z' 'A-Z'
#  Диапазоны символов должны быть заключены в кавычки
#+ чтобы предотвратить порождение имен файлов от однобуквенных имен
файлов.

exit 0
```

А теперь попробуем объединить в конвейер команду **ls -l** с этим сценарием.

```
bash$ ls -l | ./uppercase.sh
 -RW-RW-R--    1 STUDENT  STUDENT       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 STUDENT  STUDENT       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 STUDENT  STUDENT       725 APR 20 20:56 DATA-FILE
```

Выход `stdout` каждого процесса в конвейере должен читаться на входе `stdin` последующим, в конвейере, процессом. Если этого не делается, то поток данных *блокируется*, в результате конвейер будет работать не так как ожидается.

```
cat file1 file2 | ls -l | sort
# Вывод команды "cat file1 file2" будет утерян.
```

Конвейер исполняется в дочернем процессе, а посему -- не имеет доступа к переменным сценария.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

Если одна из команд в конвейере завершается аварийно, то это приводит к аварийному завершению работы всего конвейера.

>| **Принудительное перенаправление, даже если установлен ключ noclobber option.**

|| **Логическая операция OR (логическое ИЛИ).** В операциях проверки условий, оператор || возвращает 0 (success), если один из операндов имеет значение true (ИСТИНА).

& **Выполнение задачи в фоне.** Команда, за которой стоит &, будет исполняться в фоновом режиме.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

В сценариях команды, и даже циклы могут запускаться в фоновом режиме.

&& **Логическая операция AND (логическое И).** В операциях проверки условий, оператор && возвращает 0 (success) тогда, и только тогда, когда *оба* операнда имеют значение true (ИСТИНА).

- **Префикс ключа.** С этого символа начинаются опциональные ключи команд.

```
COMMAND -[Option1][Option2][...]

ls -al

sort -dfu $filename
```

```
set -- $variable
if [ $file1 -ot $file2 ]
then
  echo "Файл $file1 был создан раньше чем $file2."
fi
if [ "$a" -eq "$b" ]
then
  echo "$a равно $b."
fi
if [ "$c" -eq 24 -a "$d" -eq 47 ]
then
  echo "$c равно 24, а $d равно 47."
fi
```

перенаправление из/в `stdin` или `stdout`. [дефис]

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Перемещение полного дерева файлов и подкаталогов из одной директории в другую

# 1) cd /source/directory    Переход в исходный каталог,
#                            содержимое которого будет перемещено
# 2) &&                      "И-список": благодаря этому все последующие
#                            команды будут выполнены
#                            только тогда, когда 'cd' завершится успешно
# 3) tar cf - .              ключом 'c' архиватор 'tar' создает новый архив,
#                            ключом 'f' (file) и последующим '-' задается
#                            файл архива -- stdout,
#                            в архив помещается текущий каталог ('.')
#                            с вложенными подкаталогами.
# 4) |                       конвейер с ...
# 5) ( ... )                 subshell-ом (дочерним экземпляром командной
#                            оболочки)
# 6) cd /dest/directory      Переход в каталог назначения.
# 7) &&                      "И-список", см. выше
# 8) tar xpvf -              Разархивирование ('x'), с сохранением атрибутов
#                            "владельца" и прав доступа ('p') к файлам,
#                            с выдачей более подробных сообщений
#                            на stdout ('v'),
#                            файл архива -- stdin ('f' с последующим '-').
#
#                            Примечательно, что 'x' -- это команда, а 'p',
#                             'v' и 'f' -- ключи
# Более элегантный вариант:
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
# cp -a /source/directory /dest     имеет тот же эффект.

bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --разархивирование tar-файла--    | --затем файл передается утилите "tar"--
# Если у вас утилита "tar" не поддерживает работу с "bunzip2",
# тогда придется выполнять работу в два этапа, с использованием конвейера.
# Целью данного примера является разархивирование тарбола (tar.bz2)
# с исходными текстами ядра.
```

Обратите внимание, что в этом контексте "-" - не самостоятельный оператор Bash, а скорее опция, распознаваемая некоторыми утилитами UNIX (такими как **tar**, **cat** и т.п.), которые выводят результаты своей работы в `stdout`.

```
bash$ echo "whatever" | cat -
whatever
```

В случае, когда ожидается имя файла, тогда "-" перенаправляет вывод на `stdout` (вспомните пример с **`tar cf`**) или принимает ввод с `stdin`.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

Сама по себе команда file без параметров завершается с сообщением об ошибке.

Добавим символ "-" и получим более полезный результат. Это заставит командный интерпретатор ожидать ввода от пользователя.

```
bash$ file -
abc
standard input:              ASCII text

bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Теперь команда принимает ввод пользователя со `stdin` и анализирует его.

Используя передачу `stdout` по конвейеру другим командам, можно выполнять довольно эффектные трюки, например вставка строк в начало файла.

С помощью команды diff -- находить различия между одним файлом и *частью* другого:

```
grep Linux file1 | diff file2 -
```

И наконец пример использования служебного символа `"-"` с командой tar.

Пример. Резервное архивирование всех файлов, которые были изменены в течение последних суток.

```
#!/bin/bash

# Резервное архивирование (backup) всех файлов в текущем каталоге,
# которые были изменены в течение последних 24 часов
#+ в тарболл (tarball) (.tar.gz - файл).

BACKUPFILE=backup
archive=${1:-$BACKUPFILE}
#  На случай, если имя архива в командной строке не задано,
#+ т.е. по-умолчанию имя архива -- "backup.tar.gz"

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Каталог $PWD заархивирован в файл \"$archive.tar.gz\"."

#  Stephane Chazelas заметил, что вышеприведенный код будет "падать"
#+ если будет найдено слишком много файлов
#+ или если имена файлов будут содержать символы пробела.

# Им предложен альтернативный код:
# ------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      используется версия GNU утилиты "find".
#
#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#         более универсальный вариант, хотя и более медленный,
#         зато может использоваться в других версиях UNIX.
# ------------------------------------------------------------------

exit 0
```

Могут возникнуть конфликтные ситуации между опреатором перенаправления "-" и именами файлов, начинающимися с символа "-". Поэтому сценарий должен про-

верять имена файлов и предаварять их префиксом пути, например, `./-FILENAME`, `$PWD/-FILENAME` или `$PATHNAME/-FILENAME`.

Если значение переменной начинается с символа "-", то это тоже может быть причиной появления ошибок.

```
var="-n"
echo $var
# В данном случае команда приобретет вид "echo -n" и ничего не выведет.
```

предыдущий рабочий каталог. [дефис] Команда **cd -** выполнит переход в предыдущий рабочий каталог, путь к которому хранится в переменной окружения $OLDPWD .

Минус. Знак минус в арифметических операциях.

= **Символ "равно".** Оператор присваивания

```
a=28
echo $a   # 28
```

В зависимости от контекста применения, символ "=" может выступать в качестве оператора сравнения.

+ **Плюс.** Оператор сложения в арифметических операциях.

В зависимости от контекста применения, символ + может выступать как оператор регулярного выражения.

Ключ (опция). Дополнительный флаг для ключей (опций) команд.

Отдельные внешние и встроенные команды используют символ "+" для разрешения некоторой опции, а символ "-" -- для запрещения.

% **модуль.** Модуль (остаток от деления) -- арифметическая операция.

В зависимости от контекста применения, символ % может выступать в качестве шаблона.

~ **Домашний каталог.** [тильда] Соответствует содержимому внутренней переменной $HOME. *~student* -- домашний каталог пользователя student, а команда **ls ~student** выведет содержимое его домашнего каталога. ~/ -- это домашний каталог текущего пользователя, а команда **ls ~/** выведет содержимое домашнего каталога текущего пользователя.

```
bash$ echo ~student
/home/student

bash$ echo ~
/home/student


bash$ echo ~/
/home/student/

bash$ echo ~:
/home/student:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+ **текущий рабочий каталог.** Соответствует содержимому внутренней переменной $PWD.

~ – **предыдущий рабочий каталог.** Соответствует содержимому внутренней переменной $OLDPWD.

^ **начало-строки.** В регулярных выражениях символ **"^"** задает начало строки текста.

Управляющий символ

изменяет поведение терминала или управляет выводом текста. Управляющий символ набирается с клавиатуры как комбинация **CONTROL** + **<клавиша>**.

- **Ctl-C**

  Завершение выполнения процесса.

- **Ctl-D**

  Выход из командного интерпретатора (log out) (аналог команды exit).

  "EOF" (признак конца файла). Этот символ может выступать в качестве завершающего при вводе с `stdin`.

- **Ctl-G**

  "BEL" (звуковой сигнал -- "звонок").

- **Ctl-H**

  Backspace -- удаление предыдущего символа.

```
#!/bin/bash
# Вставка символа Ctl-H в строку.

a="^H^H"                  # Два символа Ctl-H (backspace).
echo "abcdef"             # abcdef
echo -n "abcdef$a "       # abcd f
# Пробел в конце ^             ^ двойной шаг назад.
echo -n "abcdef$a"        # abcdef
# Пробела в конце нет            backspace не работает (почему?).
# Результаты могут получиться совсем не те, что вы ожидаете.
echo; echo
```

- **Ctl-J**

  Возврат каретки.

- **Ctl-L**

  Перевод формата (очистка экрана (окна) терминала). Аналогична команде clear.

- **Ctl-M**

  Перевод строки.

- **Ctl-U**

  Стирание строки ввода.

- **Ctl-Z**

  Приостановка процесса.

Пробельный символ

используется как разделитель команд или переменных. В качестве пробельного символа могут выступать -- собственно пробел (space), символ табуляции, символ перевода строки, символ возврата каретки или комбинация из вышеперечисленных символов. В некоторых случаях, таких как присваивание значений переменным, использование пробельных символов недопустимо.

Пустые строки никак не обрабатываются командным интерпретатором и могут свободно использоваться для визуального выделения отдельных блоков сценария.

$IFS -- переменная специального назначения. Содержит символы-разделители полей, используемые некоторыми командами. По-умолчанию -- пробельные символы.

Переменные и параметры.

Переменные -- это одна из основ любого языка программирования. Они участвуют в арифметических операциях, в синтаксическом анализе строк и совершенно необходимы для абстрагирования каких либо величин с помощью символических имен. Физически переменные представляют собой ни что иное как участки памяти, в которые записана некоторая информация.

Подстановка переменных.

Когда интерпретатор встречает в тексте сценария *имя* переменной, то он вместо него подставляет *значение* этой переменной. Поэтому ссылки на переменные называются *подстановкой переменных*.

$

Необходимо всегда помнить о различиях между *именем* переменной и ее *значением*. Если **variable1** -- это имя переменной, то **$variable1** -- это ссылка на ее *значение*. "Чистые" имена переменных, без префикса $, могут использоваться только при объявлении переменный, при присваивании переменной некоторого значения, при *удалении (сбросе)*, при экспорте и в особых случаях -- когда переменная представляет собой название сигнала. Присваивание может производится с помощью символа = (например: *var1=27*), инструкцией read и в заголовке цикла (*for var2 in 1 2 3*).

Заключение ссылки на переменную в двойные кавычки (" ") никак не сказывается на работе механизма подстановки. Этот случай называется "частичные кавычки", иногда можно встретить название "нестрогие кавычки". Одиночные кавычки (' ') заставляют интерпретатор воспринимать ссылку на переменную как простой набор символов, потому в одинарных кавычках операции подстановки не производятся. Этот случай называется "полные", или "строгие" кавычки.

Примечательно, что написание **$variable** фактически является упрощенной формой написания **${variable}**. Более строгая форма записи **${variable}** может с успехом использоваться в тех случаях, когда применение упрощенной формы записи порождает сообщения о синтаксических ошибках.

Пример. Присваивание значений переменным и подстановка значений переменных.

```
#!/bin/bash

# Присваивание значений переменным и подстановка значений переменных

a=375
hello=$a

#-----------------------------------------------------------------------
# Использование пробельных символов
# с обеих сторон символа "=" присваивания недопустимо.

#  Если записать "VARIABLE =value",
#+ то интерпретатор попытается выполнить
#  команду "VARIABLE" с параметром "=value".
```

```
#  Если записать "VARIABLE= value",
#+ то интерпретатор попытается установить переменную
#  окружения "VARIABLE" в ""
#+ и выполнить команду "value".
#-----------------------------------------------------------------------


echo hello    # Это не ссылка на переменную, выведет строку "hello".

echo $hello
echo ${hello} # Идентично предыдущей строке.

echo "$hello"
echo "${hello}"

echo

hello="A B  C   D"
echo $hello   # A B C D
echo "$hello" # A B  C   D
# Здесь вы сможете наблюдать различия в выводе echo $hello и echo "$hello".
# Заключение ссылки на переменную в кавычки сохраняет пробельные символы.

echo

echo '$hello'  # $hello
# Внутри одинарных кавычек не производится подстановка значений переменных,
#+ т.е. "$" интерпретируется как простой символ.

# Обратите внимание на различия, существующие между этими типами кавычек.


hello=    # Запись пустого значения в переменную.
echo "\$hello (пустое значение) = $hello"
#  Обратите внимание: запись пустого значения -- это не то же самое,
#+ что сброс переменной, хотя конечный результат -- тот же (см. ниже).

# ---------------------------------------------------------------

#  Допускается присваивание нескольких переменных в одной строке,
#+ если они отделены пробельными символами.
#  Внимание! Это может снизить читабельность сценария и
#  оказаться непереносимым.

var1=variable1  var2=variable2  var3=variable3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# Могут возникнуть проблемы с устаревшими версиями "sh".

# ---------------------------------------------------------------

echo; echo

numbers="один два три"
other_numbers="1 2 3"
# Если в значениях переменных встречаются пробелы,
# то использование кавычек обязательно.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
echo

echo "uninitialized_variable = $uninitialized_variable"
# Неинициализированная переменная содержит "пустое" значение.
```

```
uninitialized_variable=   #  Объявление неинициализированной переменной
                          #+ (то же, что и присваивание пустого значения,
                          #   см. выше).
echo "uninitialized_variable = $uninitialized_variable"
                          # Переменная содержит "пустое" значение.

uninitialized_variable=23       # Присваивание.
unset uninitialized_variable    # Сброс.
echo "uninitialized_variable = $uninitialized_variable"
                                # Переменная содержит "пустое" значение.

echo

exit 0
```

Неинициализированная переменная хранит "пустое" значение - не ноль!. Использование неинициализированных переменных может приводить к ошибкам разного рода в процессе исполнения.

Не смотря на это в арифметических операциях допускается использовать неинициализированные переменные.

```
echo "$uninitialized"                                   # (пустая строка)
let "uninitialized += 5"                                # Прибавить 5.
echo "$uninitialized"                                   # 5

#  Заключение:
#  Неинициализированные переменные не имеют значения, однако
#+ в арифметических операциях за значение
#  таких переменных принимается число 0.
#  Это недокументированная (и возможно непереносимая) возможность.
```

Присваивание значений переменным.

=

оператор присваивания (*пробельные символы до и после оператора -- недопустимы*)

Не путайте с операторами сравнения = и -eq!

Обратите внимание: символ = может использоваться как в качестве оператора присваивания, так и в качестве оператора сравнения, конкретная интерпретация зависит от контекста применения.

Пример. Простое присваивание.

```
#!/bin/bash
# Явные переменные
# Когда перед именем переменной не употребляется символ '$'?
# В операциях присваивания.

# Присваивание
a=879
echo "Значение переменной \"a\" -- $a."

# Присваивание с помощью ключевого слова 'let'
let a=16+5
echo "Значение переменной \"a\" теперь стало равным: $a."

# В заголовке цикла 'for' (своего рода неявное присваивание)
echo -n "Значения переменной \"a\" в цикле: "
```

```
for a in 7 8 9 11
do
  echo -n "$a "
done

# При использовании инструкции 'read'
# (тоже одна из разновидностей присваивания)
echo -n "Введите значение переменной \"a\" "
read a
echo "Значение переменной \"a\" теперь стало равным: $a."

exit 0
```

Пример. Присваивание значений переменным простое и замаскированное.

```
#!/bin/bash

a=23              # Простейший случай
echo $a
b=$a
echo $b

# Теперь немного более сложный вариант (подстановка команд).

a=`echo Hello!`   # В переменную 'a' попадает результат
                  # работы команды 'echo'
echo $a
#  Обратите внимание на восклицательный знак (!) в подстанавливаемой команде
#+ этот вариант не будет работать при наборе в командной строке,
#+ поскольку здесь используется механизм "истории команд" BASH
#  Однако, в сценариях, механизм истории команд запрещен.

a=`ls -l`         # В переменную 'a' записывается результат
                  # работы команды 'ls -l'
echo $a           # Кавычки отсутствуют, удаляются лишние пробелы
                  # и пустые строки.
echo
echo "$a"         # Переменная в кавычках, все пробелы и пустые
                  # строки сохраняются.
                  # (См. главу "Кавычки.")
exit 0
```

Присваивание переменных с использованием $(...) (более современный метод, по сравнению с обратными кавычками)

```
# Взято из /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

Переменные Bash не имеют типа.

В отличие от большинства других языков программирования, Bash не производит разделения переменных по "типам". По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными. Определяющим фактором здесь служит содержимое переменных.

Пример. Целое число или строка?

```
#!/bin/bash
# int-or-string.sh: Целое число или строка?

a=2334                   # Целое число.
```

```
let "a += 1"
echo "a = $a "            # a = 2335
echo                      # Все еще целое число.

b=${a/23/BB}              # замена "23" на "BB".
                          # Происходит трансформация числа в строку.
echo "b = $b"             # b = BB35
declare -i b              # Явное указание типа здесь не поможет.
echo "b = $b"             # b = BB35

let "b += 1"              # BB35 + 1 =
echo "b = $b"             # b = 1
echo

c=BB34
echo "c = $c"             # c = BB34
d=${c/BB/23}              # замена "BB" на "23".
                          # Переменная $d становится целочисленной.
echo "d = $d"             # d = 2334
let "d += 1"              # 2334 + 1 =
echo "d = $d"             # d = 2335
echo

# А что происходит с "пустыми" переменными?
e=""
echo "e = $e"             # e =
let "e += 1"              # Арифметические операции допускают
                          # использование "пустых" переменных?
echo "e = $e"             # e = 1
echo                      # "Пустая" переменная становится целочисленной.

# А что происходит с необъявленными переменными?
echo "f = $f"             # f =
let "f += 1"              # Арифметические операции допустимы?
echo "f = $f"             # f = 1
echo                      # Необъявленная переменная
                          # трансформируется в целочисленную.
# Переменные Bash не имеют типов.

exit 0
```

Отсутствие типов - это и благословение и проклятие. С одной стороны - отсутствие типов делает сценарии более гибкими и облегчает чтение кода. С другой - является источником потенциальных ошибок и поощряет привычку к "неряшливому" программированию.

Бремя отслеживания типа той или иной переменной полностью лежит на плечах программиста. Bash не будет делать это за вас!

Специальные типы переменных.

локальные переменные

переменные, область видимости которых ограничена блоком кода или телом функции

переменные окружения

переменные, которые затрагивают командную оболочку и порядок взаимодействия с пользователем

В более общем контексте, каждый процесс имеет некоторое "окружение" (среду исполнения), т.е. набор переменных, к которым процесс может обращаться за получением

определенной информации. В этом смысле командная оболочка подобна любому другому процессу.

Каждый раз, когда запускается командный интерпретатор, для него создаются переменные, соответствующие переменным окружения. Изменение переменных или добавление новых переменных окружения заставляет оболочку обновить свои переменные, и все дочерние процессы (и команды, исполняемые ею) наследуют это окружение.

Если сценарий изменяет переменные окружения, то они должны "экспортироваться", т.е передаваться окружению, локальному по отношению к сценарию. Эта функция возложена на команду **export**.

Сценарий может **экспортировать** переменные только дочернему процессу, т.е. командам и процессам запускаемым из данного сценария. Сценарий, запускаемый из командной строки *не может* экспортировать переменные "на верх" командной оболочке. Дочерний процесс не может экспортировать переменные родительскому процессу.

позиционные параметры

аргументы, передаваемые скрипту из командной строки -- $0, $1, $2, $3..., где $0 -- это название файла сценария, $1 -- это первый аргумент, $2 -- второй, $3 -- третий и так далее. Аргументы, следующие за $9, должны заключаться в фигурные скобки, например: ${10}, ${11}, ${12}.

Специальные переменные $* и $@ содержат *все* позиционные параметры (аргументы командной строки).

Пример. Позиционные параметры.

```
#!/bin/bash

# Команда вызова сценария должна содержать по меньшей мере 10 параметров, например
# ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo "Имя файла сценария: \"$0\"."
# Для текущего каталога добавит ./
echo "Имя файла сценария: \"`basename $0`\"."
# Добавит путь к имени файла (см. 'basename')

echo

if [ -n "$1" ]              # Проверяемая переменная заключена в кавычки.
then
 echo "Параметр #1: $1"     # необходимы кавычки для экранирования символа #
fi

if [ -n "$2" ]
then
 echo "Параметр #2: $2"
fi

if [ -n "$3" ]
then
 echo "Параметр #3: $3"
fi

# ...

if [ -n "${10}" ]  # Параметры, следующие за $9 должны заключаться в фигурные скобки
then
 echo "Параметр #10: ${10}"
fi

echo "-----------------------------------"
```

```
echo "Все аргументы командной строки: "$*""

if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "Количество аргументов командной строки должно быть не менее
$MINPARAMS !"
fi

echo
exit 0
```

Скобочная нотация позиционных параметров дает довольно простой способ обращения к *последнему* аргументу, переданному в сценарий из командной строки. Такой способ подразумевает использование косвенной адресации.

```
args=$#           # Количество переданных аргументов.
lastarg=${!args}  # Обратите внимание: lastarg=${!$#} неприменимо.
```

В сценарии можно предусмотреть различные варианты развития событий, в зависимости от имени сценария. Для этого сценарий должен проанализировать аргумент `$0` -- имя файла сценария. Это могут быть и имена символических ссылок на файл сценария.

Если сценарий ожидает передачи аргументов в командной строке, то при их отсутствии он получит "пустые" переменные, что может вызвать нежелательный побочный эффект. Один из способов борьбы с подобными ошибками -- добавить дополнительный символ в обеих частях операции присваивания, где используются аргументы командной строки.

```
variable1_=$1_
# Это предотвратит появление ошибок, даже при отсутствии входного аргумента.

critical_argument01=$variable1_

# Дополнительные символы всегда можно "убрать" позднее.
# Это может быть сделано примерно так:
variable1=${variable1_/_/}   # Побочный эффект возникает только если имя пе-
ременной
                             # $variable1_ будет начинаться с символа "_".
# Здесь используется один из вариантов подстановки параметров, обсуждаемых в
Главе 9.
# Отсутствие шаблона замены приводит к удалению.

# Более простой способ заключается
#+ в обычной проверке наличия позиционного параметра.
if [ -z $1 ]
then
  exit $POS_PARAMS_MISSING
fi
```

Пример wh, whois выяснение имени домена.

```
#!/bin/bash

# Команда 'whois domain-name' выясняет имя домена на одном из 3 серверов:
#                     ripe.net, cw.net, radb.net

# Разместите этот скрипт под именем 'wh' в каталоге /usr/local/bin

# Требуемые символические ссылки:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
```

```
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` [domain-name]"
  exit 65
fi

case `basename $0` in
# Проверка имени скрипта и, соответственно, имени сервера
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "Порядок использования: `basename $0` [domain-name]";;
esac

exit 0
```

Команда **shift** "сдвигает" позиционные параметры, в результате чего парметры "сдвигаются" на одну позицию влево.

`$1` <--- `$2`, `$2` <--- `$3`, `$3` <--- `$4`, и т.д.

Прежний аргумент `$1` теряется, но аргумент *`$0` (имя файла сценария) остается без изменений*. Если вашему сценарию передается большое количество входных аргументов, то команда **shift** позволит вам получить доступ к аргументам, с порядковым номером больше `9`, без использования {фигурных скобок}.

Пример. Использование команды shift.

```
#!/bin/bash
# Использование команды 'shift' с целью перебора всех аргументов командной
строки.

#  Назовите файл с этим сценарием, например "shft",
#+ и вызовите его с набором аргументов, например:
#          ./shft a b c def 23 skidoo

until [ -z "$1" ]  # До тех пор пока не будут разобраны все входные аргумен-
ты...
do
  echo -n "$1 "
  shift
done
echo               # Дополнительная пустая строка.
exit 0
```

Команда **shift** может применяться и к входным аргументам функций.

Кавычки.

Кавычки, ограничивающие строки с обеих сторон, служат для предотвращения интерпретации специальных символов, которые могут находиться в строке. (Символ называется "специальным", если он несет дополнительную смысловую нагрузку, например символ шаблона -- *.)

```
bash$ ls -l [Vv]*
 -rw-rw-r--    1 student  student       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 student  student       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 student  student       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

Вообще, желательно использовать двойные кавычки (" ") при обращении к переменным. Это предотвратит интерпретацию специальных символов, которые могут содержаться в именах переменных, за исключением $, ` (обратная кавычка) и \ (escape -- обратный слэш). То, что символ $ попал в разряд исключений, позволяет выполнять обращение к переменным внутри строк, ограниченных двойными кавычками (*"$variable"*), т.е. выполнять подстановку значений переменных.

Двойные кавычки могут быть использованы для предотвращения разбиения строки на слова. Заключение строки в кавычки приводит к тому, что она передается как один аргумент, даже если она содержит пробельные символы - разделители.

Одиночные кавычки (' ') схожи по своему действию с двойными кавычками, только не допускают обращение к переменным, поскольку специальный символ "$" внутри одинарных кавычек воспринимается как обычный символ. Внутри одиночных кавычек, *любой* специальный символ, за исключением ', интерпретируется как простой символ. Одиночные кавычки ("строгие, или полные кавычки") следует рассматривать как более строгий вариант чем двойные кавычки ("нестрогие, или неполные кавычки").

Поскольку внутри одиночных кавычек даже экранирующий (\) символ воспринимается как обычный символ, попытка вывести одиночную кавычку внутри строки, ограниченной одинарными кавычками, не даст желаемого результата.

```
echo "Why can't I write 's between single quotes"

echo

# Обходной метод.
echo 'Why can'\''t I write '"'"'s between single quotes'
#    |-------|  |----------|   |----------------------|
# Три строки, ограниченных одинарными кавычками,
# и экранированные одиночные кавычки между ними.
```

Экранирование -- это способ заключения в кавычки одиночного символа. Экранирующий (escape) символ (\) сообщает интерпретатору, что следующий за ним символ должен восприниматься как обычный символ.

\"

кавычки

```
echo "Привет"                    # Привет
echo "Он сказал: \"Привет"."    # Он сказал: "Привет".
```

\$

символ доллара (если за комбинацией символов \$ следует имя переменной, то она не будет разименована)

```
echo "\$variable01"  # выведет $variable01
```

****

обратный слэш

```
echo "\\"  # выведет \
```

Экранирование пробелов предотвращает разбиение списка аргументов командной строки на отдельные аргументы.

```
file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# Список файлов как аргумент(ы) командной строки.

# Добавить два файла в список и вывести список.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list
```

```
echo "------------------------------------------------------------------------"

# Что произойдет, если экранировать пробелы в списке?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# Ошибка: первые три файла будут "слиты" воедино
# и переданы команде 'ls -l' как один аргумент
# потому что два пробела, разделяющие аргументы (слова) -- экранированы.
```

Кроме того, escape-символ позволяет писать многострочные команды. Обычно, каждая команда занимает одну строку, но escape-символ позволяет *экранировать символ перевода строки*, в результате чего одна команда может занимать несколько строк.

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Команда копирования дерева каталогов.
# Разбита на две строки для большей удобочитаемости.

# Альтернативный вариант:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
```

Если строка сценария заканчивается символом создания конвейера |, то необходимость в применении символа \, для экранирования перевода строки, отпадает. Тем не менее, считается хорошим тоном, всегда использовать символ "\" в конце промежуточных строк многострочных команд.

```
echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar'    # Никаких различий.
#foo
#bar

echo

echo foo\
bar     # Перевод строки экранирован.
#foobar

echo

echo "foo\
bar"     # Внутри "нестрогих" кавычек символ "\"
         # интерпретируется как экранирующий.
#foobar

echo

echo 'foo\
bar'     # В "строгих" кавычках обратный слеш воспринимается как
         # обычный символ.
#foo\
#bar
```

Завершение и код завершения.

Команда **exit** может использоваться для завершения работы сценария, точно так же как и в программах на языке C. Кроме того, она может возвращать некоторое значение, которое может быть проанализировано вызывающим процессом.

Каждая команда возвращает *код завершения* (иногда код завершения называют *возвращаемым значением*). В случае успеха команда должна возвращать 0, а в случае ошибки -- ненулевое значение, которое, как правило, интерпретируется как код ошибки. Прак-

тически все команды и утилиты UNIX возвращают 0 в случае успешного завершения, но имеются и исключения из правил.

Аналогичным образом ведут себя функции, расположенные внутри сценария, и сам сценарий, возвращая код завершения. Код, возвращаемый функцией или сценарием, определяется кодом возврата последней команды. Команде **exit** можно явно указать код возврата, в виде: **exit** *nnn*, где *nnn* -- это код возврата (число в диапазоне 0 - 255).

Когда работа сценария завершается командой **exit** без параметров, то код возврата сценария определяется кодом возврата последней исполненной командой.

Код возврата последней команды хранится в специальной переменной `$?`. После исполнения кода функции, переменная `$?` хранит код завершения последней команды, исполненной в функции. Таким способом в Bash передается "значение, возвращаемое" функцией. После завершения работы сценария, код возврата можно получить, обратившись из командной строки к переменной `$?`, т.е. это будет код возврата последней команды, исполненной в сценарии.

Пример. завершение / код завершения.

```
#!/bin/bash

echo hello
echo $?    # код возврата = 0, поскольку команда выполнилась успешно.

lskdf      # Несуществующая команда.
echo $?    # Ненулевой код возврата, поскольку команду выполнить не удалось.
echo
exit 113   # Явное указание кода возврата 113.
           # Проверить можно, если набрать в командной строке "echo $?"
           # после выполнения этого примера.

#  В соответствии с соглашениями, 'exit 0' указывает на успешное завершение,
#+ в то время как ненулевое значение означает ошибку.
```

Переменная $? особенно полезна, когда необходимо проверить результат исполнения команды.

Символ !, может выступать как логическое "НЕ" для инверсии кода возврата.

Пример. Использование символа ! для логической инверсии кода возврата.

```
true  # встроенная команда "true".
echo "код возврата команды \"true\" = $?"     # 0

! true
echo "код возврата команды \"! true\" = $?"   # 1
# Обратите внимание: символ "!" от команды необходимо отделять пробелом.
#    !true   вызовет сообщение об ошибке "command not found"
```

Проверка условий.

Практически любой язык программирования включает в себя условные операторы, предназначенные для проверки условий, чтобы выбрать тот или иной путь развития событий в зависимости от этих условий. В Bash, для проверки условий, имеется команда **test**, различного вида скобочные операторы и условный оператор **if/then**.

Конструкции проверки условий.

- Оператор **if/then** проверяет - является ли код завершения списка команд 0 (поскольку 0 означает "успех"), и если это так, то выполняет одну, или более, команд, следующие за словом then.
- Существует специальная команда -- **[** (левая квадратная скобка). Она является синонимом команды **test**, и является встроенной командой (т.е. более эффективной, в смысле производительности). Эта команда воспринимает свои аргументы как выражение сравнения или как файловую проверку и возвращает код завершения в соответствии с результатами проверки (0 -- истина, 1 -- ложь).
- Начиная с версии 2.02, Bash предоставляет в распоряжение программиста конструкцию [[...]] *расширенный вариант команды test*, которая выполняет сравнение способом более знакомым программистам, пишущим на других языках программирования. Обратите внимание: **[[** -- это зарезервированное слово, а не команда.

  Bash исполняет **`[[ $a -lt $b ]]`** как один элемент, который имеет код возврата.

  Круглые скобки ((...)) и предложение let ... так же возвращают код 0, если результатом арифметического выражения является ненулевое значение. Таким образом, арифметические выражения могут учавствовать в операциях сравнения.

```
Предложение let "1<2" возвращает 0 (так как результат сравнения "1<2" --
"1", или "истина")
(( 0 && 1 )) возвращает 1 (так как результат операции "0 && 1" -- "0",
или "ложь")
```

- Условный оператор **if** проверяет код завершения любой команды, а не только результат выражения, заключенного в квадратные скобки.

```
if cmp a b &> /dev/null  # Подавление вывода.
   then echo "Файлы a и b идентичны."
   else echo "Файлы a и b имеют различия."
fi

if grep -q Bash file
   then echo "Файл содержит, как минимум, одно слово Bash."
fi

if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
   then echo "Команда выполнена успешно."
   else echo "Обнаружена ошибка при выполнении команды."
fi
```

Оператор **if/then** допускает наличие вложенных проверок.

```
if echo "Следующий *if* находится внутри первого *if*."

  if [[ $comparison = "integer" ]]
    then (( a < b ))
  else
    [[ $a < $b ]]
  fi

then
```

```
    echo '$a меньше $b'
  fi
```

Пример. Что есть "истина"?

```
echo "Проверяется \"0\""
if [ 0 ]      # ноль
then
  echo "0 -- это истина."
else
  echo "0 -- это ложь."
fi            # 0 -- это истина.

echo "Проверяется \"1\""
if [ 1 ]      # единица
then
  echo "1 -- это истина."
else
  echo "1 -- это ложь."
fi            # 1 -- это ложь.

echo "Testing \"-1\""
if [ -1 ]     # минус один
then
  echo "-1 -- это истина."
else
  echo "-1 -- это ложь."
fi            # -1 -- это истина.

echo "Проверяется \"NULL\""
if [ ]        # NULL (пустое условие)
then
  echo "NULL -- это истина."
else
  echo "NULL -- это ложь."
fi            # NULL -- это ложь.

echo "Проверяется \"xyz\""
if [ xyz ]    # строка
then
  echo "Случайная строка -- это истина."
else
  echo "Случайная строка -- это ложь."
fi            # Случайная строка -- это истина.

echo "Проверяется \"\$xyz\""
if [ $xyz ]   # Проверка, если $xyz это null, но...
              # только для неинициализированных переменных.
then
  echo "Неинициализированная переменная -- это истина."
else
  echo "Неинициализированная переменная -- это ложь."
fi            # Неинициализированная переменная -- это ложь.

echo "Проверяется \"-n \$xyz\""
if [ -n "$xyz" ]            # Более корректный вариант.
then
  echo "Неинициализированная переменная -- это истина."
else
  echo "Неинициализированная переменная -- это ложь."
fi            # Неинициализированная переменная -- это ложь.

xyz=          # Инициализирована пустым значением.
```

```
echo "Проверяется \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "Пустая переменная -- это истина."
else
  echo "Пустая переменная -- это ложь."
fi            # Пустая переменная -- это ложь.

# Когда "ложь" истинна?

echo "Проверяется \"false\""
if [ "false" ]              #  это обычная строка "false".
then
  echo "\"false\" -- это истина." #+ и она истинна.
else
  echo "\"false\" -- это ложь."
fi            # "false" -- это истина.

echo "Проверяется \"\$false\""  # Опять неинициализированная переменная.
if [ "$false" ]
then
  echo "\"\$false\" -- это истина."
else
  echo "\"\$false\" -- это ложь."
fi            # "$false" -- это ложь.
              # Теперь мв получили ожидаемый результат.
```

Else if и elif.

elif

elif -- это краткая форма записи конструкции else if. Применяется для построения многоярусных инструкций if/then.

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [ condition2 ]
# То же самое, что и else if
then
   command4
   command5
else
   default-command
fi
```

Конструкция **if test condition-true** является точным эквивалентом конструкции **if [condition-true]**, где левая квадратная скобка **[** выполняет те же действия, что и команда **test**. Закрывающая правая квадратная скобка **]** не является абсолютно необходимой, однако, более новые версии Bash требуют ее наличие.

Команда **test** -- это встроенная команда Bash, которая выполняет проверки файлов и производит сравнение строк. Таким образом, в Bash-скриптах, команда **test** *не* вызывает внешнюю (`/usr/bin/test`) утилиту, которая является частью пакета *sh-utils*. Аналогично, [не производит вызов утилиты `/usr/bin/[`, которая является символической ссылкой на `/usr/bin/test`.

Пример. Эквиваленты команды test -- `/usr/bin/test`, [ ], и `/usr/bin/[`.

```
if test -z "$1"
```

```
then
  echo "Аргументы командной строки отсутствуют."
else
  echo "Первый аргумент командной строки: $1."
fi

if /usr/bin/test -z "$1"      # Дает тот же рузультат, что и встроенная ко-
манда "test".
then
  echo "Аргументы командной строки отсутствуют."
else
  echo "Первый аргумент командной строки: $1."
fi

if [ -z "$1" ]                # Функционально идентично вышеприведенному бло-
ку кода.
#   if [ -z "$1"                эта конструкция должна работать, но...
#+  Bash выдает сообщение об отсутствующей закрывающей скобке.
then
  echo "Аргументы командной строки отсутствуют."
else
  echo "Первый аргумент командной строки: $1."
fi

if /usr/bin/[ -z "$1"         # Функционально идентично вышеприведенному бло-
ку кода.
# if /usr/bin/[ -z "$1" ]     # Работает, но выдает сообщение об ошибке.
then
  echo "Аргументы командной строки отсутствуют."
else
  echo "Первый аргумент командной строки: $1."
fi
```

Конструкция [[ ]] более универсальна, по сравнению с [ ]. Этот *расширенный вариант команды test* перекочевал в Bash из *ksh88*.

Внутри этой конструкции не производится никакой дополнительной интерпретации имен файлов и не производится разбиение аргументов на отдельные слова, но допускается подстановка параметров и команд.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Файл паролей найден."
Fi
```

Конструкция **[[...]]** более предпочтительна, нежели **[...]**, поскольку поможет избежать некоторых логических ошибок. Например, операторы &&, ||, < и > внутри [[ ]] вполне допустимы, в то время как внутри [ ] порождают сообщения об ошибках.

Строго говоря, после оператора **if**, ни команда **test**, ни квадратные скобки ([ ] или [[ ]]) не являются обязательными.

```
dir=/home/bozo

if cd "$dir" 2>/dev/null; then   # "2>/dev/null" подавление вывода сообщений
об ошибках.
  echo "Переход в каталог $dir выполнен."
else
  echo "Невозможно перейти в каталог $dir."
```

```
fi
```
Инструкция "if COMMAND" возвращает код возврата команды COMMAND.

Точно так же, условие, находящееся внутри квадратных скобок может быть проверено без использования оператора **if**.

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 не равно $var2"

home=/home/bozo
[ -d "$home" ] || echo "каталог $home не найден."
```

Внутри (( )) производится вычисление арифметического выражения. Если результатом вычислений является ноль, то возвращается 1, или "ложь". Ненулевой результат дает код возврата 0, или "истина". То есть полная противоположность инструкциям **test** и [ ], обсуждавшимся выше.

Пример. Арифметические выражения внутри (( )).

```
#!/bin/bash
# Проверка арифметических выражений.

# Инструкция (( ... )) вычисляет арифметические выражения.
# Код возврата противоположен коду возврата инструкции [ ... ] !

(( 0 ))
echo "Код возврата \"(( 0 ))\":  $?."         # 1

(( 1 ))
echo "Код возврата \"(( 1 ))\":  $?."         # 0

(( 5 > 4 ))                                   # true
echo "Код возврата \"(( 5 > 4 ))\":  $?."     # 0

(( 5 > 9 ))                                   # false
echo "Код возврата \"(( 5 > 9 ))\":  $?."     # 1

(( 5 - 5 ))                                   # 0
echo "Код возврата \"(( 5 - 5 ))\":  $?."     # 1

(( 5 / 4 ))                                   # Деление, все в порядке
echo "Код возврата \"(( 5 / 4 ))\":  $?."     # 0

(( 1 / 2 ))                                   # Результат деления < 1.
echo "Код возврата \"(( 1 / 2 ))\":  $?."     # Округляется до 0.
                                              # 1

(( 1 / 0 )) 2>/dev/null                       # Деление на 0.
echo "Код возврата \"(( 1 / 0 ))\":  $?."     # 1
```

Операции проверки файлов.

Возвращает true если...

-e файл существует
-f *обычный* файл (не каталог и не файл устройства)
-s ненулевой размер файла
-d файл является каталогом
-b файл является блочным устройством (floppy, cdrom и т.п.)
-c файл является символьным устройством (клавиатура, модем, звуковая карта и т.п.)
-p файл является каналом
-h файл является символической ссылкой
-L файл является символической ссылкой

-S файл является сокетом
-t файл (дескриптор) связан с терминальным устройством

Этот ключ может использоваться для проверки -- является ли файл стандартным устройством ввода `stdin` (`[ -t 0 ]`) или стандартным устройством вывода `stdout` (`[ -t 1 ]`).

-r файл доступен для чтения (*пользователю, запустившему сценарий*)
-w файл доступен для записи (пользователю, запустившему сценарий)
-x файл доступен для исполнения (пользователю, запустившему сценарий)
-g set-group-id (sgid) флаг для файла или каталога установлен

Если для каталога установлен флаг *sgid*, то файлы, создаваемые в таком каталоге, наследуют идентификатор группы каталога, который может не совпадать с идентификатором группы, к которой принадлежит пользователь, создавший файл. Это может быть полезно для каталогов, в которых хранятся файлы, общедоступные для группы пользователей.

-u set-user-id (suid) флаг для файла установлен

Установленный флаг suid приводит к изменению привилегий запущенного процесса на привилегии владельца исполняемого файла. Исполняемые файлы, владельцем которых является *root*, с установленным флагом *set-user-id* запускаются с привилегиями *root*, даже если их запускает обычный пользователь. Это может оказаться полезным для некоторых программ (таких как **pppd** и **cdrecord**), которые осуществляют доступ к аппаратной части компьютера. В случае отсутствия флага *suid*, программы не смогут быть запущены рядовым пользователем, не обладающим привилегиями root.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

Файл с установленным флагом *suid* отображается с включенным флагом *s* в поле прав доступа.

-k флаг *sticky bit* (бит фиксации) установлен

Общеизвестно, что флаг "sticky bit" -- это специальный тип прав доступа к файлам. Программы с установленным флагом "sticky bit" остаются в системном кэше после своего завершения, обеспечивая тем самым более быстрый запуск программы. [17] Если флаг установлен для каталога, то это приводит к ограничению прав на запись. Установленный флаг "sticky bit" отображается в виде символа *t* в поле прав доступа.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

Если пользователь не является владельцем каталога, с установленным "sticky bit", но имеет право на запись в каталог, то он может удалять только те файлы в каталоге, владельцем которых он является. Это предотвращает удаление и перезапись "чужих" файлов в общедоступных каталогах, таких как `/tmp`.

-O вы являетесь владельцем файла
-G вы принадлежите к той же группе, что и файл
-N файл был модифицирован с момента последнего чтения

f1 -nt f2 файл *f1* более новый, чем *f2*
f1 -ot f2 файл *f1* более старый, чем *f2*
f1 -ef f2 файлы *f1* и *f2* являются "жесткими" ссылками на один и тот же файл

! "НЕ" -- логическое отрицание (инверсия) результатов всех вышеприведенных проверок (возвращается true если условие отсутствует).

Операции сравнения.

сравнение целых чисел.

-eq равно

```
if [ "$a" -eq "$b" ]
```

-ne не равно

```
if [ "$a" -ne "$b" ]
```

-gt больше

```
if [ "$a" -gt "$b" ]
```

-ge больше или равно

```
if [ "$a" -ge "$b" ]
```

-lt меньше

```
if [ "$a" -lt "$b" ]
```

-le меньше или равно

```
if [ "$a" -le "$b" ]
```

< меньше (внутри двойных круглых скобок)

```
(("$a" < "$b"))
```

<= меньше или равно (внутри двойных круглых скобок)

```
(("$a" <= "$b"))
```

> больше (внутри двойных круглых скобок)

```
(("$a" > "$b"))
```

>= больше или равно (внутри двойных круглых скобок)

```
(("$a" >= "$b"))
```

сравнение строк.

= равно

```
if [ "$a" = "$b" ]
```

== равно

```
if [ "$a" == "$b" ]
```

Синоним оператора =.

```
[[ $a == z* ]]    # истина, если $a начинается с символа "z" (сравнение
по шаблону)
[[ $a == "z*" ]]  # истина, если $a равна z*

[ $a == z* ]      # имеют место подстановка имен файлов и разбиение на
слова
[ "$a" == "z*" ]  # истина, если $a равна z*
```

!= не равно

```
if [ "$a" != "$b" ]
```

Этот оператор используется при поиске по шаблону внутри [[...]].

< меньше, в смысле величины ASCII-кодов

if [["$a" < "$b"]]

if ["$a" \< "$b"]

Обратите внимание! Символ "<" необходимо экранировать внутри **[ ]**.

> больше, в смысле величины ASCII-кодов

if [["$a" > "$b"]]

if ["$a" \> "$b"]

Обратите внимание! Символ ">" необходимо экранировать внутри **[ ]**.

-z строка "пустая", т.е. имеет нулевую длину

-n строка не "пустая".

Оператор **-n** требует, чтобы строка была заключена в кавычки внутри квадратных скобок. Как правило, проверка строк, не заключенных в кавычки, оператором **! -z**, или просто указание строки без кавычек внутри квадратных скобок, проходит нормально, однако это небезопасная, с точки зрения отказоустойчивости, практика. *Всегда* заключайте проверяемую строку в кавычки.

Пример. Операции сравнения.

```
a=4
b=5

#  Здесь переменные "a" и "b" могут быть как целыми числами, так и строками.
#  Здесь наблюдается некоторое размывание границ
#+ между целочисленными и строковыми переменными,
#+ поскольку переменные в Bash не имеют типов.

#  Bash выполняет целочисленные операции над теми переменными,
#+ которые содержат только цифры
#  Будьте внимательны!

if [ "$a" -ne "$b" ]
then
  echo "$a не равно $b"
  echo "(целочисленное сравнение)"
fi

if [ "$a" != "$b" ]
then
  echo "$a не равно $b."
  echo "(сравнение строк)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# Оба варианта, "-ne" и "!=", работают правильно.
```

Пример. Проверка - является ли строка *пустой.*

```
# Проверка пустых строк и строк, не заключенных в кавычки,
# Используется конструкция   if [ ... ]
# Если строка не инициализирована,
# то она не имеет никакого определенного значения.
# Такое состояние называется "null" (пустая) (это не то же самое, что ноль).

if [ -n $string1 ]    # $string1 не была объявлена или инициализирована.
then
  echo "Строка \"string1\" не пустая."
```

```
else
  echo "Строка \"string1\" пустая."
fi
# Неверный результат.
# Выводится сообщение о том, что $string1 не пустая,
#+не смотря на то, что она не была инициализирована.
# Попробуем еще раз.

if [ -n "$string1" ]  # На этот раз, переменная $string1 заключена в кавычки.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi      # Внутри квадратных скобок заключайте строки в кавычки!

if [ $string1 ]       # Опустим оператор -n.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# Все работает прекрасно.
# Квадратные скобки -- [ ], без посторонней помощи определяют,
# что строка пустая.
# Тем не менее, хорошим тоном считается заключать строки в кавычки
("$string1").
#
#    if [ $string 1 ]   один аргумент "]"
#    if [ "$string 1" ]  два аргумента, пустая "$string1" и "]"

string1=initialized

if [ $string1 ]       # Опять, попробуем строку без ничего.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# И снова получим верный результат.
# И опять-таки, лучше поместить строку в кавычки ("$string1"), поскольку...

string1="a = b"

if [ $string1 ]       # И снова, попробуем строку без ничего..
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# Строка без кавычек дает неверный результат!
```

построение сложных условий проверки.

-a логическое И (and)

exp1 -a exp2 возвращает true, если *оба* выражения, и exp1, и exp2 истинны.

-o логическое ИЛИ (or)

exp1 -o exp2 возвращает true, если хотябы одно из выражений, exp1 *или* exp2 истинно.

Они похожи на операторы Bash **&&** и ||, употребляемые в двойных квадратных скобках.

```
[[ condition1 && condition2 ]]
```
Операторы **-o** и **-a** употребляются совместно с командой **test** или внутри одинарных квадратных скобок.

```
if [ "$exp1" -a "$exp2" ]
```

Вложенные условные операторы if/then.

Операторы проверки условий **if/then** могут быть вложенными друг в друга. Конечный результат будет таким же как если бы результаты всех проверок были объединены оператором **&&**.

```
if [ condition1 ]
then
  if [ condition2 ]
  then
    do-something  # Только если оба условия "condition1" и "condition2" ис-
тинны.
  fi
fi
```

Операции.

Операторы.

Присваивание.

variable assignment

Инициализация переменной или изменение ее значения

= Универсальный оператор присваивания, пригоден как для сравнения целых чисел, так и для сравнения строк.

```
var=27
category=minerals  # Пробелы до и после оператора "=" -- недопустимы.
```

Пусть вас не смущает, что оператор присваивания ("="), по своему внешнему виду, совпадает с оператором сравнения (=).

```
#    Здесь знак "="  выступает в качестве оператора сравнения
if [ "$string1" = "$string2" ]
# if [ "X$string1" = "X$string2" ] более отказоустойчивый вариант,
# предохраняет от "сваливания" по ошибке в случае,
# когда одна из переменных пуста.
# (добавленные символы "X" компенсируют друг друга.)
then
   command
fi
```

арифметические операторы.

\+ сложение
\- вычитание
* умножение
/ деление
** возведение в степень

```
let "z=5**3"
echo "z = $z"   # z = 125
```

% модуль (деление по модулю), возвращает остаток от деления
bash$ **echo `expr 5 % 3`**

+= "плюс-равно" (увеличивает значение переменной на заданное число)

let "var += 5" значение переменной var будет увеличено на 5.

-= "минус-равно" (уменьшение значения переменной на заданное число)
*= "умножить-равно" (умножить значение переменной на заданное число, результат записать в переменную)

let "var *= 4" значение переменной var будет увеличено в 4 раза.

/= "слэш-равно" (уменьшение значения переменной в заданное число раз)
%= "процент-равно" (найти остаток от деления значения переменной на заданное число, результат записать в переменную)

Арифметические операторы очень часто используются совместно с командами expr и let.

Пример. Арифметические операции.

```
# От 1 до 6 пятью различными способами.
n=1; echo -n "$n "

let "n = $n + 1"   # let "n = n + 1"   тоже допустимо
echo -n "$n "

: $((n = $n + 1))
# оператор ":" обязателен, поскольку в противном случае, Bash будет
#+ интерпретировать выражение "$((n = $n + 1))" как команду.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
# оператор ":" обязателен, поскольку в противном случае, Bash будет
#+ интерпретировать выражение "$[ n = $n + 1 ]" как команду.
# Не вызывает ошибки даже если "n" содержит строку.
echo -n "$n "
```

Целые числа в Bash фактически являются знаковыми *длинными* целыми (32-бит), с диапазоном изменений от -2147483648 до 2147483647. Если в результате какой либо операции эти пределы будут превышены, то результат получится ошибочным.

```
a=2147483646
echo "a = $a"      # a = 2147483646
let "a+=1"         # Увеличить "a" на 1.
echo "a = $a"      # a = 2147483647
let "a+=1"         # увеличить "a" еще раз, с выходом за границы диапазона.
echo "a = $a"      # a = -2147483648
                   #      ОШИБКА! (выход за границы диапазона)
```

битовые операции. Битовые операции очень редко используются в сценариях командного интерпретатора. Их главное назначение, на мой взгляд, установка и проверка некоторых значений, читаемых из портов ввода-вывода и сокетов. "Битовые операции" гораздо более уместны в компилирующих языках программирования, таких как C и C++.

битовые операции.

<< сдвигает на 1 бит влево (умножение на 2)

<<= "сдвиг-влево-равно"

let "var <<= 2" значение переменной `var` сдвигается влево на 2 бита (умножается на `4`)

>> сдвиг вправо на 1 бит (деление на `2`)

>>= "сдвиг-вправо-равно" (имеет смысл обратный <<=)

& по-битовое И (AND)

&= "по-битовое И-равно"

| по-битовое ИЛИ (OR)

|= "по-битовое ИЛИ-равно"

~ по-битовая инверсия

! По-битовое отрицание

^ по-битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)

^= "по-битовое ИСКЛЮЧАЮЩЕЕ-ИЛИ-равно"

логические операции.

&& логическое И (and)

```
if [ $condition1 ] && [ $condition2 ]
# То же самое, что:  if [ $condition1 -a $condition2 ]
# Возвращает true если оба операнда condition1 и condition2 истинны...

if [[ $condition1 && $condition2 ]]    # То же верно
# Обратите внимание: оператор && не должен использоваться внутри [ ...
].
```

оператор &&, в зависимости от контекста, может так же использоваться в И-списках для построения составных команд.

|| логическое ИЛИ (or)

```
if [ $condition1 ] || [ $condition2 ]
# То же самое, что:  if [ $condition1 -o $condition2 ]
# Возвращает true если хотя бы один из операндов истинен...

if [[ $condition1 || $condition2 ]]    # Also works.
# Обратите внимание: оператор || не должен использоваться внутри [ ...
].
```

Bash производит проверку кода возврата КАЖДОГО из операндов в логических выражениях.

Пример. Построение сложных условий, использующих && и ||.

```
a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Первая проверка прошла успешно."
else
  echo "Первая проверка не прошла."
fi

if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Вторая проверка прошла успешно."
else
  echo "Вторая проверка не прошла."
fi
```

```
#  Опции -a и -o предоставляют
#+ альтернативный механизм проверки условий.

if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "Третья проверка прошла успешно."
else
  echo "Третья проверка не прошла."
fi


if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Четвертая проверка прошла успешно."
else
  echo "Четвертая проверка не прошла."
fi

a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "Пятая проверка прошла успешно."
else
  echo "Пятая проверка не прошла."
fi
```

Операторы && и || могут использоваться и в арифметических вычислениях.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

прочие операции.

, запятая

С помощью оператора **запятая** можно связать несколько арифметических в одну последовательность. При разборе таких последовательностей, командный интерпретатор вычисляет все выражения (которые могут иметь побочные эффекты) в последовательности и возвращает результат последнего.

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"               # t1 = 11

let "t2 = ((a = 9, 15 / 3))"  #  Выполняется присваивание "a" = 9,
                              #+ а затем вычисляется "t2".
echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

Оператор запятая чаще всего находит применение в циклах for.

Числовые константы.

Интерпретатор командной оболочки воспринимает числа как десятичные, в противном случае числу должен предшествовать специальный префикс, либо число должно быть записано в особой нотации. Числа, начинающиеся с символа *0*, считаются *восьмеричными*. если числу предшествует префикс *0x*, то число считается *шестнадцатеричным*. Число, в записи которого присутствует символ *#*, расценивается как запись числа с указанием основы счисления в виде *ОСНОВА#ЧИСЛО*.

Пример. Различные представления числовых констант.

```
# numbers.sh: Различные представления числовых констант.

# Десятичное: по-умолчанию
let "dec = 32"
echo "десятичное число = $dec"              # 32
# Вобщем-то ничего необычного.

# Восьмеричное: числа начинаются с '0' (нуля)
let "oct = 032"
echo "восьмеричное число = $oct"            # 26
# Результат печатается в десятичном виде.
# --------- ------ -- -------

# Шестнадцатиричное: числа начинаются с '0x' или '0X'
let "hex = 0x32"
echo "шестнадцатиричное число = $hex"       # 50
# Результат печатается в десятичном виде.

# Другие основы счисления: ОСНОВА#ЧИСЛО
# ОСНОВА должна быть между 2 и 64.
# для записи ЧИСЛА должен использоваться соответствующий ОСНОВЕ диапазон сим-
волов,

let "bin = 2#111100111001101"
echo "двоичное число = $bin"                # 31181

let "b32 = 32#77"
echo "32-ричное число = $b32"               # 231

let "b64 = 64#@_"
echo "64-ричное число = $b64"               # 4094
#
#  Нотация ОСНОВА#ЧИСЛО может использоваться на ограниченном
#+ диапазоне основ счисления (от 2 до 64)
#  10 цифр + 26 символов в нижнем регистре + 26 символов в верхнем регистре +
@ + _

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                        # 1295 170 44822 3375

#  Важное замечание:
#  --------------
#  Использование символов, для записи числа, выходящих за диапазон,
#+ соответствующий ОСНОВЕ счисления
#+ будет приводить к появлению сообщений об ошибках.

let "bad_oct = 081"
# numbers.sh: let: oct = 081: value too great for base (error token is "081")
#                  Для записи восьмеричных чисел допускается использовать
#+                 только цифры в диапазоне 0 - 7.
```

Переменные.

Правильное использование переменных может придать сценариям дополнительную мощь и гибкость, а для этого необходимо изучить все тонкости и нюансы.

Внутренние переменные.

Встроенные *переменные*
$BASH

путь к исполняемому файлу *Bash*

```
bash$ echo $BASH
/bin/bash
```

$DIRSTACK

содержимое вершины стека каталогов (который управляется командами pushd и popd)

Эта переменная соответствует команде dirs, за исключением того, что **dirs** показывает полное содержимое всего стека каталогов.

$EDITOR

заданный по-умолчанию редактор, вызываемый скриптом, обычно **vi** или **emacs**.

$EUID

"эффективный" идентификационный номер пользователя (Effective User ID)

Идентификационный номер пользователя, права которого были получены, возможно с помощью команды su.

$GLOBIGNORE

Перечень шаблонных символов, которые будут проигнорированы при выполнении подстановки имен файлов (globbing) .

$GROUPS

группы, к которым принадлежит текущий пользователь

Это список групп (массив) идентификационных номеров групп для текущего пользователя, как эо записано в `/etc/passwd`.

```
root# echo $GROUPS
0

root# echo ${GROUPS[1]}
1

root# echo ${GROUPS[5]}
6
```

$HOME

домашний каталог пользователя, как правило это `/home/username`

$HOSTNAME

Сетевое имя хоста устанавливается командой hostname во время исполнения инициализирующих сценариев на загрузке системы. Внутренняя переменная `$HOSTNAME` Bash получает свое значение посредством вызова функции `gethostname()`.

$HOSTTYPE

тип машины

Подобно $MACHTYPE, идентифицирует аппаратную архитектуру.

```
bash$ echo $HOSTTYPE
i686
```

$IFS

разделитель полей во вводимой строке (IFS -- Input Field Separator)

По-умолчанию -- пробельный символ (пробел, табуляция и перевод строки), но может быть изменен, например, для разбора строк, в которых отдельные поля разделены запятыми. Обратите внимание: при составлении содержимого переменной $*, Bash использует первый символ из `$IFS` для разделения аргументов.

```
bash$ echo $IFS | cat -vte
$

bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```

$LC_COLLATE

Чаще всего устанавливается в `.bashrc` или `/etc/profile`, эта переменная задает порядок сортировки символов, в операциях подстановки имен файлов и в поиске по шаблону. При неверной настройке переменной `LC_COLLATE` можно получить весьма неожиданные результаты.

$LC_CTYPE

Эта внутренняя переменная определяет кодировку символов. Используется в операциях подстановки и поиске по шаблону.

$LINENO

Номер строки исполняемого сценария. Эта переменная имеет смысл только внутри исполняемого сценария и чаще всего применяется в отладочных целях.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_  # Запомнить.

echo "Строка $LINENO: переменная \"v1\" = $v1"
echo "Последний аргумент командной строки = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

$MACHTYPE

аппаратная архитектура

Идентификатор аппаратной архитектуры.

```
bash$ echo $MACHTYPE
i686
```

$OLDPWD

прежний рабочий каталог ("OLD-Print-Working-Directory")

$OSTYPE

тип операционной системы

```
bash$ echo $OSTYPE
linux
```

$PATH

путь поиска, как правило включает в себя каталоги `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, и т.д.

Когда командный интерпретатор получает команду, то он автоматически пытается отыскать соответствующий исполняемый файл в указанном списке каталогов (в переменной $PATH). Каталоги, в указанном списке, должны отделяться друг от друга двоеточиями. Обычно, переменная `$PATH` инициализируется в `/etc/profile` и/или в `~/.bashrc`.

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

Инструкция **PATH=${PATH}:/opt/bin** добавляет каталог /opt/bin в конец текущего пути поиска. Иногда может оказаться целесообразным, внутри сценария, временно добавить какой-либо каталог к пути поиска. По завершении работы скрипта, эти изменения будут утеряны (вспомните о том, что невозможно изменить переменные окружения вызывающего процесса).

Текущий "рабочий каталог", ./, обычно не включается в $PATH из соображений безопасности.

$PPID

Переменная $PPID хранит PID (идентификатор) родительского процесса.

Сравните с командой pidof.

$PWD

рабочий (текущий) каталог

Аналог встроенной команды pwd.

$SECONDS

Время работы сценария в секундах.

```
TIME_LIMIT=10
INTERVAL=1

echo "Для прерывания работы сценария, ранее чем через $TIME_LIMIT се-
кунд, нажмите Control-C."
echo

while [ "$SECONDS" -le "$TIME_LIMIT" ]
do
  if [ "$SECONDS" -eq 1 ]
  then
   units=second
  else
    units=seconds
  fi
echo "Сценарий отработал $SECONDS $units."
  #  В случае перегруженности системы, скрипт может
  #  перескакивать через отдельные
  #+  значения счетчика
  sleep $INTERVAL
done

echo -e "\a"  # Сигнал!
```

$SHELLOPTS

список допустимых опций интерпретатора shell. Переменная доступна только для чтения.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-
comments:emacs
```

Позиционные параметры (аргументы).

$0, $1, $2 и т.д.

аргументы передаются... из командной строки в сценарий, функциям или команде set

$#

количество аргументов командной строки, или позиционных.

$*

Все аргументы в виде одной строки (слова)

$@

То же самое, что и $*, но при этом каждый параметр представлен как отдельная строка (слово), т.е. параметры не подвергаются какой либо интерпретации.

Пример. arglist: Вывод списка аргументов с помощью переменных $* и $@

```
#!/bin/bash
# Вызовите сценарий с несколькими аргументами,
# например: "один два три".

E_BADARGS=65

if [ ! -n "$1" ]
then
  echo "Порядок использования: `basename $0` argument1 argument2 и
т.д."
  exit $E_BADARGS
fi

echo

index=1

echo "Список аргументов в переменной \"\$*\":"
for arg in "$*"  # Работает некорректно, если "$*" не ограничена кавыч-
ками.
do
  echo "Аргумент #$index = $arg"
  let "index+=1"
done             # $* воспринимает все аргументы как одну строку.
echo "Полный список аргументов выглядит как одна строка."

index=1

echo "Список аргументов в переменной \"\$@\":"
for arg in "$@"
do
  echo "Аргумент #$index = $arg"
  let "index+=1"
done             # $@ воспринимает аргументы как отдельные строки (сло-
ва).
echo "Список аргументов выглядит как набор различных строк (слов)."
```

После команды **shift** (сдвиг), первый аргумент, в переменной `$@`, теряется, а остальные сдвигаются на одну позицию "вниз" (или "влево", если хотите).

```
#!/bin/bash
# Вызовите сценарий в таком виде: ./scriptname 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
```

```
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Каждая из команд "shift" приводит к потере аргумента $1,
# но остальные аргументы остаются в "$@".
```

Специальная переменная `$@` может быть использована для выбора типа ввода в сценария. Команда **cat "$@"** позволяет выполнять ввод как со стандартного устройства ввода `stdin`, так и из файла, имя которого передается сценарию из командной строки.

Переменные `$*` и `$@`, в отдельных случаях, могут содержать противоречивую информацию! Это зависит от содержимого переменной $IFS.

Прочие специальные переменные.

$!

PID последнего, запущенного в фоне, процесса

```
LOG=$0.log
COMMAND1="sleep 100"
echo "Запись в лог всех PID фоновых процессов, запущенных из сценария:
$0" >> "$LOG"
# Таким образом возможен мониторинг и удаление процессов
# по мере необходимости.
echo >> "$LOG"
# Команды записи в лог.
echo -n "PID of \"$COMMAND1\":  " >> "$LOG"
${COMMAND1} &
echo $! >> "$LOG"
# PID процесса "sleep 100":  1506
```

$_

Специальная переменная, содержит последний аргумент предыдущей команды.

Пример. Переменная "подчеркивание".

```
#!/bin/bash
echo $_              # /bin/bash
                     # Для запуска сценария был вызван /bin/bash.
du >/dev/null        # Подавление вывода.
echo $_              # du
ls -al >/dev/null    # Подавление вывода.
echo $_              # -al  (последний аргумент)
:
echo $_              # :
```

$?

Код возврата команды, функции или скрипта

$$

PID самого процесса-сценария. Переменная `$$` часто используется при генерации "уникальных" имен для временных файлов. Обычно это проще чем вызов mktemp.

Работа со строками.

Bash поддерживает на удивление большое количество операций над строками. К сожалению, этот раздел Bash испытывает недостаток унификации. Одни операции являются подмножеством операций подстановки параметров, а другие - совпадают с функциональностью команды UNIX -- expr. Это приводит к противоречиям в синтаксисе команд и

перекрытию функциональных возможностей, не говоря уже о возникающей путанице.

Длина строки.

${#string}
expr length $string
expr "$string" : '.*'

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

Пример. Вставка пустых строк между параграфами в текстовом файле.

```
#!/bin/bash
# paragraph-space.sh

# Вставка пустых строк между параграфами в текстовом файле.
# Порядок использования: $0 <FILENAME

MINLEN=45        # Возможно потребуется изменить это значение.
#  Строки, содержащие количество символов меньшее, чем $MINLEN
#+ принимаются за последнюю строку параграфа.

while read line  # Построчное чтение файла от начала до конца...
do
  echo "$line"   # Вывод строки.

  len=${#line}
  if [ "$len" -lt "$MINLEN" ]
    then echo    # Добавление пустой строки после последней строки параграфа.
  fi
done

exit 0
```

Длина подстроки в строке (подсчет совпадающих символов ведется с начала строки).

expr match "$string" '$substring'

где *$substring* -- регулярное выражение.

expr "$string" : '$substring'

где *$substring* -- регулярное выражение.

```
stringZ=abcABC123ABCabc
#       |------|

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

Index.

expr index $string $substring

Номер позиции первого совпадения в $string с первым символом в $substring.

```
stringZ=abcABC123ABCabc
echo `expr index "$stringZ" C12`             # 6
                                             # позиция символа C.
```

```
echo `expr index "$stringZ" 1c`              # 3
# символ 'c' (в #3 позиции) совпал раньше, чем '1'.
```

Эта функция довольно близка к функции *strchr()* в языке C.

Извлечение подстроки.

${string:position}

Извлекает подстроку из *$string*, начиная с позиции *$position*.

Если строка `$string` -- "*" или "@", то извлекается позиционный параметр (аргумент), с номером `$position`.

${string:position:length}

Извлекает *$length* символов из *$string*, начиная с позиции *$position*.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       Индексация начинается с 0.

echo ${stringZ:0}                            # abcABC123ABCabc
echo ${stringZ:1}                            # bcABC123ABCabc
echo ${stringZ:7}                            # 23ABCabc

echo ${stringZ:7:3}                          # 23A
                                             # Извлекает 3 символа.

# Возможна ли индексация с "правой" стороны строки?

echo ${stringZ:-4}                           # abcABC123ABCabc
# По-умолчанию выводится полная строка.
# Однако . . .

echo ${stringZ:(-4)}                         # Cabc
echo ${stringZ: -4}                          # Cabc
# Теперь выводится правильно.
# Круглые скобки или дополнительный пробел "экранируют" параметр позиции.
```

Если `$string` -- "*" или "@", то извлекается до `$length` позиционных параметров (аргументов), начиная с `$position`.

```
echo ${*:2}          # Вывод 2-го и последующих аргументов.
echo ${@:2}          # То же самое.

echo ${*:2:3}        # Вывод 3-х аргументов, начиная со 2-го.
```

expr substr $string $position $length

Извлекает *$length* символов из *$string*, начиная с позиции *$position*.

```
stringZ=abcABC123ABCabc
#       123456789......
#       Индексация начинается с 1.

echo `expr substr $stringZ 1 2`              # ab
echo `expr substr $stringZ 4 3`              # ABC
```

expr match "$string" '\($substring\)'

Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* -- это регулярное выражение.

expr "$string" : '\($substring\)'

Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* -- это регулярное выражение.

```
stringZ=abcABC123ABCabc
#       =======

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
# Все вышеприведенные операции дают один и тот же результат.
```

expr match "$string" '.*\($substring\)'

Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* -- это регулярное выражение. Поиск начинается с конца *$string*.

expr "$string" : '.*\($substring\)'

Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* -- это регулярное выражение. Поиск начинается с конца *$string*.

```
stringZ=abcABC123ABCabc

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

Удаление части строки.

${string#substring}

Удаление самой короткой, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с начала строки

${string##substring}

Удаление самой длинной, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с начала строки

```
stringZ=abcABC123ABCabc
#       |----|
#       |----------|

echo ${stringZ#a*C}      # 123ABCabc
# Удаление самой короткой подстроки.

echo ${stringZ##a*C}     # abc
# Удаление самой длинной подстроки.
```

${string%substring}

Удаление самой короткой, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с конца строки

${string%%substring}

Удаление самой длинной, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с конца строки

```
stringZ=abcABC123ABCabc
#                    ||
#        |------------|

echo ${stringZ%b*c}      # abcABC123ABCa
# Удаляется самое короткое совпадение. Поиск ведется с конца $stringZ.

echo ${stringZ%%b*c}     # a
# Удаляется самое длинное совпадение. Поиск ведется с конца $stringZ.
```

Замена подстроки.

${string/substring/replacement}

Замещает первое вхождение *$substring* строкой *$replacement*.

${string//substring/replacement}

Замещает все вхождения *$substring* строкой *$replacement*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}           # xyzABC123ABCabc
                                  # Замена первой подстроки 'abc' стро-
кой 'xyz'.

echo ${stringZ//abc/xyz}          # xyzABC123ABCxyz
                                  # Замена всех подстрок 'abc' строкой
'xyz'.
```

${string/#substring/replacement}

Подстановка строки *$replacement* вместо *$substring*. Поиск ведется с начала строки *$string*.

${string/%substring/replacement}

Подстановка строки *$replacement* вместо *$substring*. Поиск ведется с конца строки *$string*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
                                  # Поиск ведется с начала строки
echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  # Поиск ведется с конца строки
```

Подстановка параметров.

Работа с переменными и/или подстановка их значений.

${parameter}

То же самое, что и *$parameter*, т.е. значение переменной *parameter*. В отдельных случаях, при возникновении неоднозначности интерпретации, корректно будет работать только такая форма записи: *${parameter}*.

Может использоваться для конкатенации (слияния) строковых переменных.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Старый \$PATH = $PATH"
PATH=${PATH}:/opt/bin  #Добавление /opt/bin в $PATH.
```

```
echo "Новый \$PATH = $PATH"
```

${parameter-default}**, **${parameter:-default}

Если параметр отсутствует, то используется значение по-умолчанию.

```
echo ${username-`whoami`}
# Вывод результата работы команды `whoami`, если переменная $username
не установлена.

username0=
# переменная username0 объявлена, но инициализирована
# "пустым" значением.
echo "username0 = ${username0-`whoami`}"
# Вывод после символа "=" отсутствует.

echo "username1 = ${username1-`whoami`}"
# Переменная username1 не была объявлена.
# Выводится имя пользователя, выданное командой `whoami`.

username2=
# переменная username2 объявлена, но инициализирована "пустым" значени-
ем.
echo "username2 = ${username2:-`whoami`}"
# Выводится имя пользователя, выданное командой `whoami`, поскольку
#+здесь употребляется конструкция ":-" , а не  "-".
```

Параметры по-умолчанию очень часто находят применение в случаях, когда сценарию необходимы какие либо входные аргументы, передаваемые из командной строки, но такие аргументы не были переданы.

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
#  Если имя файла не задано явно, то последующие операторы будут рабо-
тать
#+ с файлом "generic.data".
#
```

${parameter=default}**, **${parameter:=default}

Если значения параметров не задананы явно, то они принимают значения по-умолчанию.

Оба метода задания значений по-умолчанию до определенной степени идентичны. Символ : имеет значение только когда *$parameter* был инициализирован "пустым" (null) значением, как показано выше.

```
echo ${username=`whoami`}
# Переменная "username" принимает значение, возвращаемое командой
`whoami`.
```

${parameter+alt_value}**, **${parameter:+alt_value}

Если параметр имеет какое либо значение, то используется **alt_value**, иначе -- null ("пустая" строка).

Оба варианта до определенной степени идентичны. Символ : имеет значение только если *parameter* объявлен и "пустой", см. ниже.

```
      echo "###### \${parameter+alt_value} ########"
      echo

      a=${param1+xyz}
      echo "a = $a"      # a =

      param2=
      a=${param2+xyz}
      echo "a = $a"      # a = xyz

      param3=123
      a=${param3+xyz}
      echo "a = $a"      # a = xyz

      echo
      echo "###### \${parameter:+alt_value} ########"
      echo

      a=${param4:+xyz}
      echo "a = $a"      # a =

      param5=
      a=${param5:+xyz}
      echo "a = $a"      # a =
      # Вывод отличается от a=${param5+xyz}

      param6=123
      a=${param6+xyz}
      echo "a = $a"      # a = xyz
```

${parameter?err_msg}**, **${parameter:?err_msg}

Если parameter инициализирован, то используется его значение, в противном случае -- выводится err_msg.

Обе формы записи можно, до определенной степени, считать идентичными. Символ : имеет значение только когда *parameter* инициализирован "пустым" значением, см. ниже.

Пример . Подстановка параметров и сообщения об ошибках.

```
#  Проверка отдельных переменных окружения.
#  Если переменная, к примеру $USER, не установлена,
#+ то выводится сообщение об ошибке.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Имя машины: $HOSTNAME."
  echo "Ваше имя: $USER."
  echo "Ваш домашний каталог: $HOME."
  echo "Ваш почтовый ящик: $MAIL."
  echo
  echo "Если перед Вами появилось это сообщение,"
  echo "то это значит, что все критические переменные окружения установлены."
  echo
  echo

# ------------------------------------------------------

#  Конструкция ${variablename?} так же выполняет проверку
#+ наличия переменной в сценарии.

ThisVariable=Value-of-ThisVariable
```

```
#  Обратите внимание, в строковые переменные могут быть записаны
#+ символы, которые запрещено использовать в именах переменных.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".

: ${ZZXy23AB?"Переменная ZZXy23AB не инициализирована."}
#  Если ZZXy23AB не инициализирована,
#+ то сценарий завершается с сообщением об ошибке.

# Текст сообщения об ошибке можно задать свой.
# : ${ZZXy23AB?"Переменная ZZXy23AB не инициализирована."}


# То же самое:  dummy_variable=${ZZXy23AB?}
#               dummy_variable=${ZZXy23AB?"Переменная ZXy23AB не инициализи-
рована."}
#
#               echo ${ZZXy23AB?} >/dev/null

echo "Это сообщение не будет напечатано, поскольку сценарий завершится рань-
ше."

HERE=0
exit $HERE   # Сценарий завершит работу не здесь.
```

Длина переменной / Удаление подстроки.

`${#var}`

> **`String length`** (число символов в переменной `$var`). В случае массивов, команда **${#array}** возвращает длину первого элемента массива.

Исключения:

- **${#*}** и **${#@}** возвращает *количество аргументов (позиционных параметров)*.
- Для массивов, **${#array[*]}** и **${#array[@]}** возвращает количество элементов в массиве.

Пример. Длина переменной.

```
#!/bin/bash
# length.sh
E_NO_ARGS=65
if [ $# -eq 0 ]  # Для работы скрипта необходим хотя бы один входной
параметр.
then
  echo "Вызовите сценарий с одним или более параметром командной стро-
ки."
  exit $E_NO_ARGS
fi
var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Length of var01 = ${#var01}"

echo "Количество входных параметров = ${#@}"
echo "Количество входных параметров = ${#*}"

exit 0
```

Пример. Поиск по шаблону в подстановке параметров.

```
#!/bin/bash
# Поиск по шаблону в операциях подстановки параметров # ## % %%.
```

```
var1=abcd12345abc6789
pattern1=a*c  # * (символ шаблона), означает любые символы между a и c.

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789   (альтернативный вариант)
echo "Число символов в ${var1} = ${#var1}"
echo "pattern1 = $pattern1"   # a*c  (между 'a' и 'c' могут быть любые симво-
лы)
echo

echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12345abc6789
# Наименьшая подстрока, удаляются первые 3 символа  abcd12345abc6789
                                ^^^^^^              |-|
echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                  6789
# Наибольшая подстрока, удаляются первые 12 символов abcd12345abc6789
#                               ^^^^^^              |----------|

echo; echo

pattern2=b*9            # все, что между 'b' и '9'
echo "var1 = $var1"     # abcd12345abc6789
echo "pattern2 = $pattern2"
echo

echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd12345a
# Наименьшая подстрока, удаляются последние 6 символов  abcd12345abc6789
#                               ^^^^^^^^^                          |----|
echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
# Наибольшая подстрока, удаляются последние 12 символов  abcd12345abc6789
#                               ^^^^^^^^^               |-------------|

# Запомните, # и ## используются для поиска с начала строки,
#            % и %% используются для поиска с конца строки.
```

Пример. Изменение расширений в именах файлов:

```
# Изменение расширений в именах файлов.
#
#         rfe old_extension new_extension
#
# Пример:
# Изменить все расширения *.gif в именах файлов на *.jpg, в текущем каталоге
#          rfe gif jpg

ARGS=2
E_BADARGS=65
if [ $# -ne "$ARGS" ]
then
  echo "Порядок использования: `basename $0` old_file_suffix new_file_suffix"
  exit $E_BADARGS
fi

for filename in *.$1
# Цикл прохода по списку имен файлов, имеющих расширение равное первому аргу-
менту.
do
  mv $filename ${filename%$1}$2
  #  Удалить первое расширение и добавить второе,
done
```

Подстановка значений переменных / Замена подстроки.

Эти конструкции перекочевали в Bash из *ksh*.

${var:pos}

Подстанавливается значение переменной *var*, начиная с позиции *pos*.

${var:pos:len}

Подстанавливается значение переменной *var*, начиная с позиции *pos*, не более *len* символов.

${var/Pattern/Replacement}

Первое совпадение с шаблоном *Pattern*, в переменной *var* замещается подстрокой *Replacement*.

Если подстрока *Replacement* отсутствует, то найденное совпадение будет удалено.

${var//Pattern/Replacement}

Глобальная замена. Все найденные совпадения с шаблоном *Pattern*, в переменной *var*, будут замещены подстрокой *Replacement*.

Как и в первом случае, если подстрока *Replacement* отсутствует, то все найденные совпадения будут удалены.

${var/#Pattern/Replacement}

Если в переменной *var* найдено совпадение с *Pattern*, причем совпадающая подстрока расположена в начале строки (префикс), то оно заменяется на *Replacement*. Поиск ведется с начала строки

${var/%Pattern/Replacement}

Если в переменной *var* найдено совпадение с *Pattern*, причем совпадающая подстрока расположена в конце строки (суффикс), то оно заменяется на *Replacement*. Поиск ведется с конца строки

${!varprefix*}**, **${!varprefix@}

Поиск по шаблону всех, ранее объявленных переменных, имена которых начинаются с *varprefix*.

```
xyz23=whatever
xyz24=

a=${!xyz*}      # Подстановка имен объявленных переменных, которые на-
чинаются с "xyz".
echo "a = $a"   # a = xyz23 xyz24
a=${!xyz@}      # То же самое.
echo "a = $a"   # a = xyz23 xyz24

# Эта возможность была добавлена в Bash, в версии 2.04.
```

$RANDOM: генерация псевдослучайных целых чисел.

$RANDOM -- внутренняя функция Bash (не константа), которая возвращает *псевдослучайные* целые числа в диапазоне 0 - 32767. Функция $RANDOM *не* должна использоваться для генераци ключей шифрования.

Пример. Генерация случайных чисел.

```
#!/bin/bash

# $RANDOM возвращает различные случайные числа при каждом обращении к ней.
# Диапазон изменения: 0 - 32767 (16-битовое целое со знаком).
```

```
MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT случайных чисел:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]      # Генерация 10 ($MAXCOUNT) случайных
чисел.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # Нарастить счетчик.
done
echo "-----------------"

# Если вам нужны случайные числа не превышающие определенного числа,
# воспользуйтесь оператором деления по модулю (остаток от деления).

RANGE=500
FLOOR=200

number=0   #initialize
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"  # Ограничение "сверху" числом $RANGE.
done
echo "Случайное число в диапазоне от $FLOOR до $RANGE ---  $number"
echo


# Генерация случайных "true" и "false" значений.
BINARY=2
number=$RANDOM
T=1

let "number %= $BINARY"
# let "number >>= 14"    дает более равномерное распределение
# (сдвиг вправо смещает старший бит на нулевую позицию, остальные биты обну-
ляются).
if [ "$number" -eq $T ]
then
  echo "TRUE"
else
  echo "FALSE"
fi
```

Системный генератор `/dev/urandom` дает последовательность псевдослучайных чисел с более равномерным распределением, чем `$RANDOM`. Команда **`dd if=/dev/urandom of=targetfile bs=1 count=XX`** создает файл, содержащий последовательность псевдослучайных чисел. Однако, эти числа требуют дополнительной обработки, например с помощью команды od (этот прием используется в примере выше) или dd.

Двойные круглые скобки.

Эта конструкция во многом похожа на инструкцию let, внутри **((...))** вычисляются арифметические выражения и возвращается их результат. В простейшем случае, конструкция **`a=$(( 5 + 3 ))`** присвоит переменной "a" значение выражения "5 + 3", или 8. Но, кроме того, двойные круглые скобки позволяют работать с переменными в стиле языка C.

Пример. Работа с переменными в стиле языка C.

```
#!/bin/bash
# Работа с переменными в стиле языка C.

(( a = 23 ))  # Присвоение переменной в стиле C, с обоих строн от "=" стоят
пробелы.
echo "a (начальное значение) = $a"

(( a++ ))     # Пост-инкремент 'a', в стиле C.
echo "a (после a++) = $a"

(( a-- ))     # Пост-декремент 'a', в стиле C.
echo "a (после a--) = $a"

 (( ++a ))     # Пред-инкремент 'a', в стиле C.
echo "a (после ++a) = $a"

(( --a ))     # Пред-декремент 'a', в стиле C.
echo "a (после --a) = $a"

 (( t = a<45?7:11 ))   # Трехместный оператор в стиле языка C.
echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "        # Да!
```

Циклы и ветвления.

Управление ходом исполнения - один из ключевых моментов структурной организации сценариев на языке командной оболочки. Циклы и переходы являются теми инструментальными средствами, которые обеспечивают управление порядком исполнения команд.

Циклы.

Цикл - это блок команд, который исполняется многократно до тех пор, пока не будет выполнено условие выхода из цикла.

циклы for

for (in)

Это одна из основных разновидностей циклов. И она значительно отличается от аналога в языке C.

for *arg* in [*list*]
do
 команда(ы)...
done

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# На первом проходе, $arg = $var1
# На втором проходе, $arg = $var2
# На третьем проходе, $arg = $var3
# На N-ном проходе, $arg = $varN

# Элементы списка заключены в кавычки для того, чтобы предотвратить
возможное разбиение их на отдельные аргументы (слова).
```

Элементы списка могут включать в себя шаблонные символы.

Есл ключевое слово **do** находится в одной строке со словом **for**, то после списка аргументов (перед do) необходимо ставить точку с запятой.

for *arg* in [*list*] ; do

Пример. Простой цикл for.

```
# Список планет.
for planet in Меркурий Венера Земля Марс Юпитер Сатурн Уран Нептун Плу-
тон
do
  echo $planet
done

echo

# Если 'список аргументов' заключить в кавычки,
# то он будет восприниматься как единственный аргумент .
for planet in "Меркурий Венера Земля Марс Юпитер Сатурн Уран Нептун
Плутон"
do
  echo $planet
done
```

Каждый из элементов **[списка]** может содержать несколько аргументов. Это бывает полезным при обработке групп параметров. В этом случае, для принудительного разбора каждого из аргументов в списке, необходимо использовать инструкцию **set**

Пример. Цикл for с двумя параметрами в каждом из элементов списка.

```
# Список планет.
# Имя кажой планеты ассоциировано с расстоянием от планеты до Солнца
(млн. миль).
for planet in "Меркурий 36" "Венера 67" "Земля 93"  "Марс 142" "Юпитер
483"
do
  set -- $planet  # Разбиение переменной "planet" на
                  # множество аргументов (позиционных параметров).
  # Конструкция "--" предохраняет от неожиданностей, если $planet "пус-
та" или начинается с символа "-".
  # Если каждый из аргументов потребуется сохранить, поскольку на сле-
дующем проходе они будут "забиты" новыми значениями,
  # То можно поместить их в массив,
  #        original_params=("$@")
  echo "$1      в $2,000,000 миль от Солнца"
  #----две табуляции---к параметру $2 добавлены нули
done
```

Если **[список]** в цикле **for** не задан, то в качестве оного используется переменная $@ -- список аргументов командной строки.

Пример. Цикл for без списка аргументов.

```
for a
do
 echo -n "$a "
done
#  Список аргументов не задан, поэтому цикл работает с переменной '$@'
#+ (список аргументов командной строки, включая пробельные символы).
```

while

Оператор while проверяет условие перед началом каждой итерации и если условие истинно (если код возврата равен 0), то управление передается в тело цикла. В отличие от циклов for, циклы *while* используются в тех случаях, когда количество итераций заранее не известно.

while [*condition*]

do
command...
done

Как и в случае с циклами for/in, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do**.

while [*condition*] ; do

Пример. Простой цикл while.

```
var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n подавляет перевод строки.
  var0=`expr $var0 + 1`   # допускается var0=$(($var0+1)).
done
```

Пример. Другой пример цикла while.

```
while [ "$var1" != "end" ]     # возможна замена на while test "$var1"
!= "end"
do
  echo "Введите значение переменной #1 (end - выход) "
  read var1      # Конструкция 'read $var1' недопустима (почему?).
  echo "переменная #1 = $var1"
  # кавычки обязательны, потому что имеется символ "#".
  # Если введено слово 'end', то оно тоже выводится на экран.
  # потому, что проверка переменной выполняется
  # в начале итерации (перед вводом).
  echo
done
```

Оператор **while** может иметь несколько условий. Но только последнее из них определяет возможность продолжения цикла. В этом случае синтаксис оператора цикла должен быть несколько иным.

Пример. Цикл while с несколькими условиями.

```
var1=unset
previous=$var1

while echo "предыдущее значение = $previous"
      echo
      previous=$var1     # запомнить предыдущее значение
      [ "$var1" != end ]
      # В операторе "while" присутствуют 4 условия,
      # но только последнее управляет циклом.
      # *последнее* условие - единственное,
      # которое вычисляется.
do
echo "Введите значение переменной #1 (end - выход) "
  read var1
  echo "текущее значение = $var1"
done
```

Как и в случае с **for**, цикл **while** может быть записан в С-подобной нотации, с использованием двойных круглых скобок

Пример. С-подобный синтаксис оформления цикла while.

```
# wh-loopc.sh: Цикл перебора от 1 до 10.
LIMIT=10
a=1
while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done            # Пока ничего особенного.

# А теперь оформим в стиле языка C.
((a = 1))       # a=1
# Двойные скобки допускают наличие лишних пробелов в выражениях.
while (( a <= LIMIT ))    # В двойных скобках символ "$"
                          # Перед переменными опускается.
do
  echo -n "$a "
  ((a += 1))    # let "a+=1"
  # Двойные скобки позволяют наращивание переменной в стиле языка C.
Done
```

Стандартное устройство ввода stdin, для цикла **while**, можно перенаправить на файл с помощью команды перенаправления < в конце цикла.

until

Оператор цикла **until** проверяет условие в начале каждой итерации, но в отличие от **while** итерация возможна только в том случае, если условие ложно.

until [*condition-is-true*]
do
 command...
done

Обратите внимание: оператор **until** проверяет условие завершения цикла ПЕРЕД очередной итерацией, а не после, как это принято в некоторых языках программирования.

Как и в случае с циклами for/in, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do**.

until [*condition-is-true*] ; do

Пример. Цикл until.

```
until [ "$var1" = end ] # Проверка условия производится
                        # в начале итерации.
do
  echo "Введите значение переменной #1 "
  echo "(end - выход)"
  read var1
  echo "значение переменной #1 = $var1"
done
```

Вложенные циклы.

Цикл называется вложенным, если он размещается внутри другого цикла. На первом проходе, внешний цикл вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл. Само собой, как внешний, так и внутренний циклы могут быть прерваны командой **break**.

Пример. Вложенный цикл.

```
# Вложенные циклы "for".
outer=1             # Счетчик внешнего цикла.
# Начало внешнего цикла.
for a in 1 2 3 4 5
do
  echo "Итерация #$outer внешнего цикла."
  echo "---------------------"
  inner=1           # Сброс счетчика вложенного цикла.
  # Начало вложенного цикла.
  for b in 1 2 3 4 5
  do
    echo "Итерация #$inner вложенного цикла."
    let "inner+=1"  # Увеличить счетчик итераций вложенного цикла.
  done
  # Конец вложенного цикла.
  let "outer+=1"    # Увеличить счетчик итераций внешнего цикла.
  echo              # Пустая строка для отделения итераций внешнего цикла.
Done
```

Управление ходом выполнения цикла.

break, **continue**

Для управления ходом выполнения цикла служат команды **break** и **continue** и точно соответствуют своим аналогам в других языках программирования. Команда **break** прерывает исполнение цикла, в то время как **continue** передает управление в начало цикло, минуя все последующие команды в теле цикла.

Пример. Команды break и continue в цикле.

```
LIMIT=19  # Верхний предел
echo "Печать чисел от 1 до 20 (исключая 3 и 11)."
a=0
while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Исключить 3 и 11
 then
   continue  # Переход в начало цикла.
 fi
 echo -n "$a "
done
```

Команде **break** может быть передан необязательный параметр. Команда **break** без параметра прерывает тот цикл, в который она вставлена, а **break N** прерывает цикл, стоящий на N уровней выше (причем 1-й уровень - это уровень текущего цикла.).

Команда **continue**, как и команда **break**, может иметь необязательный параметр. В простейшем случае, команда **continue** передает управление в начало текущего цикла, а команда **continue N** прерывает исполнение текущего цикла и передает управление в начало внешнего цикла, отстоящего от текущего на N уровней.

Операторы выбора.

Инструкции **case** и **select** технически не являются циклами, поскольку не преду-

сматривают многократное исполнение блока кода. Однако, они, как и циклы, управляют ходом исполнения программы, в зависимости от начальных или конечных условий.

case (in) / esac

Конструкция **case** эквивалентна конструкции **switch** в языке C/C++. Она позволяет выполнять тот или иной участок кода, в зависимости от результатов проверки условий. Она является, своего рода, краткой формой записи большого количества операторов if/then/else и может быть неплохим инструментом при создании разного рода меню.

case "$*variable*" in

"$*condition1*")
command...
;;

"$*condition2*")
command...
;;

esac

- Заключать переменные в кавычки необязательно, поскольку здесь не производится разбиения на отдельные слова.
- Каждая строка с условием должна завершаться правой (закрывающей) круглой скобкой).
- Каждый блок команд, отрабатывающих по заданному условию, должен завершаться *двумя* символами точка-с-запятой ;;.
- Блок **case** должен завершаться ключевым словом **esac** (*case* записанное в обратном порядке).

Пример. Использование case.

```
echo; echo "Нажмите клавишу и затем клавишу Return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "буква в нижнем регистре";;
  [A-Z]   ) echo "Буква в верхнем регистре";;
  [0-9]   ) echo "Цифра";;
  *       ) echo "Знак пунктуации, пробел или что-то другое";;
esac  # Допускается указывать диапазоны
      # символов в [квадратных скобках].
```

Очень хороший пример использования **case** для анализа аргументов, переданных из командной строки.

```
case "$1" in
"") echo "Порядок использования: ${0##*/} <filename>"; exit 65;;
# Параметры командной строки отсутствуют,
# или первый параметр -- "пустой".
# Обратите внимание на ${0##*/} это подстановка параметра
${var##pattern}. В результате получается $0.
-*) FILENAME=./$1;;   # Если имя файла (аргумент $1) начинается с "-",
                      # то заменить его на ./$1
                      # тогда параметр не будет восприниматься
                      # как ключ команды.
```

```
      * ) FILENAME=$1;;      # В противном случае -- $1.
      esac
```

select

Оператор **select** был заимствован из Korn Shell, и является еще одним инструментом, используемым при создании меню.

select *variable* [in *list*]
do
command...
break
done

Этот оператор предлагает пользователю выбрать один из представленных вариантов. Примечательно, что **select** по-умолчанию использует в качестве приглашения к вводу (prompt) -- `PS3` (`#?`), который легко изменить.

Пример. Создание меню с помощью select.

```
PS3='Выберите ваш любимый овощ: ' # строка приглашения к вводу (prompt)
select vegetable in "бобы" "морковь" "картофель" "лук" "брюква"
do
  echo
  echo "Вы предпочитаете $vegetable."
  echo ";-))"
  echo
  break  # если 'break' убрать, то получится бесконечный цикл.
done
```

Если в операторе **select** список **in *list*** не задан, то в качестве списка будет использоваться список аргументов (`$@`), передаваемый сценарию или функции.

Пример. Создание меню с помощью select в функции

```
PS3='Выберите ваш любимый овощ: '
choice_of()
{
select vegetable
# список выбора [in list] отсутствует, поэтому 'select'
# использует входные аргументы функции.
do
  echo
  echo "Вы предпочитаете $vegetable."
  echo ";-))"
  echo
  break
done
}

choice_of бобы рис морковь редис томат шпинат
#         $1   $2  $3      $4    $5    $6
#         передача списка выбора в функцию choice_of()
```

Подстановка команд.

Подстановка команд -- это подстановка результатов выполнения команды или даже серии команд; буквально, эта операция позволяет вызвать команду в другом окружении.

Классический пример подстановки команд -- использование обратных одиночных кавычек (`...`). Команды внутри этих кавычек представляют собой текст командной строки.

```
script_name=`basename $0`
```

```
echo "Имя этого файла-сценария: $script_name."
```

Вывод от команд может использоваться: как аргумент другой команды, для установки значения переменной и даже для генерации списка аргументов цикла for**.**

```
rm `cat filename`   # здесь "filename" содержит список удаляемых файлов.
#
# Такой вариант будет лучше:   xargs rm -- < filename
# ( -- подходит для случая, когда "filename" начинается с символа "-" )

textfile_listing=`ls *.txt`
# Переменная содержит имена всех файлов *.txt в текущем каталоге.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # Альтернативный вариант.
echo $textfile_listing2
# Результат будет тем же самым.

# Проблема записи списка файлов в строковую переменную состоит в том,
# что символы перевода строки заменяются на пробел.
#
# Как вариант решения проблемы -- записывать список файлов в массив.
#      shopt -s nullglob    # При несоответствии, имя файла игнорируется.
#      textfile_listing=( *.txt )
```

Подстанавливаемая команда может получиться разбитой на отдельные слова.

```
COMMAND `echo a b`     # 2 аргумента: a и b

COMMAND "`echo a b`"   # 1 аргумент: "a b"

COMMAND `echo`         # без аргументов

COMMAND "`echo`"       # один пустой аргумент
```

Даже когда не происходит разбиения на слова, операция подстановки команд может удалять завершающие символы перевода строки.

```
# cd "`pwd`"  # Должна выполняться всегда.
# Однако...

mkdir 'dir with trailing newline'

cd 'dir with trailing newline'

cd "`pwd`"  # Ошибка:
# bash: cd: /tmp/dir with trailing newline: No such file or directory

cd "$PWD"   # Выполняется без ошибки.
old_tty_setting=$(stty -g)   # Сохранить настройки терминала.
echo "Нажмите клавишу "
stty -icanon -echo           # Запретить "канонический" режим терминала.
                             # Также запрещает эхо-вывод.
key=$(dd bs=1 count=1 2> /dev/null)   # Поймать нажатие на клавишу.
stty "$old_tty_setting"      # Восстановить настройки терминала.
echo "Количество нажатых клавиш = ${#key}."  # ${#variable} = количество сим-
волов в переменной $variable
#
# Нажмите любую клавишу, кроме RETURN, на экране появится "Количество нажатых
```

```
клавиш = 1."
# Нажмите RETURN, и получите: "Количество нажатых клавиш = 0."
# Символ перевода строки будет "съеден" операцией подстановки команды.
```

Подстановка команд позволяет даже записывать в переменные содержимое целых файлов, с помощью перенаправления или команды cat.

```
variable1=`<file1`      # Записать в переменную  "variable1" содержимое файла
"file1".
variable2=`cat file2`   # Записать в переменную "variable2" содержимое файла
"file2".

#  Замечание 1:
#  Удаляются символы перевода строки.
#
#  Замечание 2:
#  В переменные можно записать даже управляющие символы.

#  Выдержки из системного файла /etc/rc.d/rc.sysinit
#+ (Red Hat Linux)

if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=01"`
...
fi
```

Подстановка команд, позволяет записать в переменную результаты выполнения цикла. Ключевым моментом здесь является команда echo, в теле цикла.

Пример. Запись результатов выполнения цикла в переменную.

```
# csubloop.sh: Запись результатов выполнения цикла в переменную

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                 #  Здесь 'echo' -- это ключевой момент
done`

echo "variable1 = $variable1"  # variable1 = 12345

i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                 # Опять же, команда 'echo' просто необходима.
  let "i += 1"                 # Увеличение на 1.
```

```
done`

echo "variable2 = $variable2"  # variable2 = 0123456789
```

Подстановка команд позволяет существенно расширить набор инструментальных средств, которыми располагает Bash. Суть состоит в том, чтобы написать программу или сценарий, которая выводит результаты своей работы на `stdout` (как это делает подавляющее большинство утилит в UNIX) и записать вывод от программы в переменную.

Арифметические подстановки.

Арифметические подстановки - это мощный инструмент, предназначенный для выполнения арифметических операций в сценариях. Перевод строки в числовое выражение производится с помощью обратных одиночных кавычек, двойных круглых скобок или предложения let.

Вариации.

Арифметические подстановки в обратных одиночных кавычках (часто используются совместно с командой expr)

```
z=`expr $z + 3`            # Команда 'expr' вычисляет значение выражения.
```

Арифметические подстановки в двойных круглых скобках, и предложение **let**

В арифметических подстановках, обратные одиночные кавычки могут быть заменены на двойные круглые скобки **`$((...))`** или очень удобной конструкцией, с применением предложения **let**.

```
z=$(($z+3))
# $((EXPRESSION)) -- это подстановка арифметического выражения.
#  Не путайте с подстановкой команд.

let z=z+3
let "z += 3"  # Кавычки позволяют вставлять пробелы и специальные операторы.
#  Оператор 'let' вычисляет арифметическое выражение,
#  это не подстановка арифметического выражения.
```

Все вышеприведенные примеры эквивалентны. Вы можете использовать любую из этих форм записи "по своему вкусу".

Перенаправление ввода/вывода.

В системе по-умолчанию всегда открыты три "файла" -- `stdin` (клавиатура), `stdout` (экран) и `stderr` (вывод сообщений об ошибках на экран). Эти, и любые другие открытые файлы, могут быть перенаправлены. В данном случае, термин "перенаправление" означает получить вывод из файла, команды, программы, сценария или даже отдельного блока в сценарии и передать его на вход в другой файл, команду, программу или сценарий.

С каждым открытым файлом связан дескриптор файла. Дескрипторы файлов `stdin`, `stdout` и `stderr` - 0, 1 и 2, соответственно. При открытии дополнительных файлов, дескрипторы с 3 по 9 остаются незанятыми. Иногда дополнительные дескрипторы могут сослужить неплохую службу, временно сохраняя в себе ссылку на `stdin`, `stdout` или `stderr`. Это упрощает возврат дескрипторов в нормальное состояние после сложных ма-

нипуляций с перенаправлением и перестановками.

```
COMMAND_OUTPUT >
      # Перенаправление stdout (вывода) в файл.
      # Если файл отсутствовал, то он создается,
      # иначе -- перезаписывается.

ls -lR > dir-tree.list
      # Создает файл, содержащий список дерева каталогов.

: > filename
      # Операция > усекает файл "filename" до нулевой длины.
      # Если до выполнения операции файла не существовало,
      # то создается новый файл с нулевой длиной
      # (тот же эффект дает команда 'touch').
      # Символ : выступает здесь в роли местозаполнителя,
      # не выводя ничего.

> filename
      # Операция > усекает файл "filename" до нулевой длины.
      # Если до выполнения операции файла не существовало,
      # то создается новый файл с нулевой длиной
      # (тот же эффект дает команда 'touch').
      # (тот же результат, что и выше -- ": >",
      # но этот вариант неработоспособен
      # в некоторых командных оболочках.)

COMMAND_OUTPUT >>
      # Перенаправление stdout (вывода) в файл.
      # Создает новый файл, если он отсутствовал,
      # иначе -- дописывает в конец файла.

      # Однострочные команды перенаправления
      # (затрагивают только ту строку, в которой они встречаются):
      # ------------------------------------------------------

1>filename
      # Перенаправление вывода (stdout) в файл "filename".
1>>filename
      # Перенаправление вывода (stdout) в файл "filename",
      # файл открывается в режиме добавления.
2>filename
      # Перенаправление stderr в файл "filename".
2>>filename
      # Перенаправление stderr в файл "filename",
      # файл открывается в режиме добавления.
&>filename
      # Перенаправление stdout и stderr в файл "filename".


#==============================================================
      # Перенаправление stdout, только для одной строки.
      LOGFILE=script.log

      echo "Эта строка будет записана в файл \"$LOGFILE\"." 1>$LOGFILE
      echo "Эта строка будет добавлена в конец файла \"$LOGFILE\"."
1>>$LOGFILE
      echo "Эта строка тоже будет добавлена в конец файла
\"$LOGFILE\"." 1>>$LOGFILE
      echo "Эта строка будет выведена на экран и не попадет в файл
\"$LOGFILE\"."
      # После каждой строки, сделанное перенаправление автоматически
"сбрасывается".
```

```
      # Перенаправление stderr, только для одной строки.
ERRORFILE=script.errors

bad_command1 2>$ERRORFILE
      #  Сообщение об ошибке запишется в $ERRORFILE.
bad_command2 2>>$ERRORFILE
      #  Сообщение об ошибке добавится в конец $ERRORFILE.
#============================================================================
   2>&1
      # Перенаправляется stderr на stdout.
      # Сообщения об ошибках передаются туда же, куда и стандартный вывод.

   i>&j
      # Перенаправляется файл с дескриптором i в j.
      # Вывод в файл с дескриптором i передается в файл с дескриптором j.

   >&j
      # Перенаправляется  файл с дескриптором 1 (stdout) в файл с дескриптором j.
      # Вывод на stdout передается в файл с дескриптором j.

   0< FILENAME
    < FILENAME
      # Ввод из файла.
      # Парная команде ">", часто встречается в комбинации с ней.
      #
      # grep search-word <filename


   [j]<>filename
      # Файл "filename" открывается на чтение и запись, и связывается с дескриптором "j".
      # Если "filename" отсутствует, то он создается.
      # Если дескриптор "j" не указан, то, по-умолчанию, бередся дескриптор 0, stdin.
      #
      # Как одно из применений этого - запись в конкретную позицию в файле.
      echo 1234567890 > File    # Записать строку в файл "File".
      exec 3<> File             # Открыть "File" и связать
                                # с дескриптором 3.
      read -n 4 <&3             # Прочитать 4 символа.
      echo -n . >&3             # Записать символ точки.
      exec 3>&-                 # Закрыть дескриптор 3.
      cat File                  # ==> 1234.67890
      # Произвольный доступ, да и только!
   |
      # Конвейер (канал).
      # Универсальное средство для объединения команд в одну цепочку.
      # Похоже на ">", но на самом деле -- более обширная.
      # Используется для объединения команд, сценариев,
      # файлов и программ в одну цепочку (конвейер).
cat *.txt | sort | uniq > result-file
      # Содержимое всех файлов .txt сортируется,
      # удаляются повторяющиеся строки,
      # результат сохраняется в файле "result-file".
```

Операции перенаправления и/или конвейеры могут комбинироваться в одной командной строке.

```
command < input-file > output-file

command1 | command2 | command3 > output-file
```

Допускается перенаправление нескольких потоков в один файл.

```
ls -yz >> command.log 2>&1
# Сообщение о неверной опции "yz" в команде "ls"
# будет записано в файл "command.log".
# Поскольку stderr перенаправлен в файл.
```

Закрытие дескрипторов файлов.

n<&-

Закрыть дескриптор входного файла *n*.

0<&-, <&-

Закрыть `stdin`.

n>&-

Закрыть дескриптор выходного файла *n*.

1>&-, >&-

Закрыть `stdout`.

Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры. Чтобы предотвратить наследование дескрипторов -- закройте их перед запуском дочернего процесса.

```
# В конвейер передается только stderr.

exec 3>&1                                 # Сохранить текущее "состояние"
                                          # stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-       # Закрыть дескр. 3 для 'grep'
                                          # (но не для 'ls').
#              ^^^^   ^^^^
exec 3>&-                                 # Теперь закрыть его для оставшейся
                                          # части сценария.
```

Использование exec.

Команда **exec <filename** перенаправляет ввод со `stdin` на файл. С этого момента весь ввод, вместо `stdin` (обычно это клавиатура), будет производиться из этого файла. Это дает возможность читать содержимое файла, строку за строкой, и анализировать каждую введенную строку с помощью sed и/или awk.

Пример. Перенаправление `stdin` с помощью exec

```
#!/bin/bash
# Перенаправление stdin с помощью 'exec'.
exec 6<&0          # Связать дескр. #6 со стандартным вводом (stdin).
                   # Сохраняя stdin.

exec < data-file   # stdin заменяется файлом "data-file"

read a1            # Читается первая строка из "data-file".
read a2            # Читается вторая строка из "data-file."

echo
echo "Следующие строки были прочитаны из файла."
```

```
echo "----------------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Восстанавливается stdin из дескр. #6, где он был предварительно сохранен,
#+ и дескр. #6 закрывается ( 6<&- ) освобождая его для других процессов.
#
# <&6 6<&-    дает тот же результат.

echo -n "Введите строку  "
read b1  # Теперь функция "read", как и следовало ожидать, принимает данные с
обычного stdin.
echo "Строка, принятая со stdin."
echo "--------------------------"
echo "b1 = $b1"
```

Аналогично, конструкция **exec >filename** перенаправляет вывод на `stdout` в заданный файл. После этого, весь вывод от команд, который обычно направляется на `stdout`, теперь выводится в этот файл.

Пример. Перенаправление `stdout` с помощью exec

```
LOGFILE=logfile.txt

exec 6>&1           # Связать дескр. #6 со stdout.
                    # Сохраняя stdout.

exec > $LOGFILE     # stdout замещается файлом "logfile.txt".

# ------------------------------------------------------------ #
# Весь вывод от команд, в данном блоке, записывается в файл $LOGFILE.

echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Вывод команды \"ls -al\""
echo
ls -al
echo; echo
echo "Вывод команды \"df\""
echo
df

# ------------------------------------------------------------ #

exec 1>&6 6>&-      # Восстановить stdout и закрыть дескр. #6.

echo
echo "== stdout восстановлено в значение по-умолчанию == "
echo
ls -al
```

Внутренние команды.

Внутренняя **команда** - это **команда**, которая встроена непосредственно в Bash. Команды делаются встроенными либо из соображений производительности - встроенные команды исполняются быстрее, чем внешние, которые, как правило, запускаются в дочер-

нем процессе, либо из-за необходимости прямого доступа к внутренним структурам командного интерпретатора.

Действие, когда какая либо команда или сама командная оболочка инициирует (*порождает*) новый подпроцесс, что бы выполнить какую либо работу, называется *ветвлением* (*forking*) процесса. Новый процесс называется "дочерним" (или "потомком"), а породивший его процесс - "родительским" (или "предком"). В результате и *потомок* и *предок* продолжают исполняться одновременно -- параллельно друг другу.

В общем случае, *встроенные команды* Bash, при исполнении внутри сценария, не порождают новый подпроцесс, в то время как вызов внешних команд, как правило, приводит к созданию нового подпроцесса.

Внутренние команды могут иметь внешние аналоги. Например, внутренняя команда Bash -- **echo** имеет внешний аналог `/bin/echo` и их поведение практически идентично.

```
echo "Эта строка выводится внутренней командой \"echo\"."
/bin/echo "А эта строка выводится внешней командой the /bin/echo."
```

Ключевое слово (keyword) -- это *зарезервированное* слово, синтаксический элемент (token) или оператор. Ключевые слова имеют специальное назначение для командного интерпретатора, и фактически являются элементами синтаксиса языка командной оболочки. В качестве примера можно привести "for", "while", "do", "!", которые являются ключевыми (или зарезервированными) словами. Подобно *встроенным командам*, ключевые слова жестко зашиты в Bash, но в отличие от встроенных команд, ключевые слова не являются командами как таковыми, хотя при этом могут являться их составной частью.

Ввод/вывод.

echo

выводит (на `stdout`) выражение или содержимое переменной.

```
echo Hello
echo $a
```

Для вывода экранированных символов, **echo** требует наличие ключа `-e`.

Обычно, командв **echo** выводит в конце символ перевода строки. Подавить вывод это символа можно ключом `-n`.

Команда **echo** может использоваться для передачи информации по конвейеру другим командам.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR содержит подстроку \"txt\""
fi
```

Кроме того, команда **echo**, в комбинации с подстановкой команд может учавствовать в операции присвоения значения переменной.

```
a=`echo "HELLO" | tr A-Z a-z`
```

Следует запомнить, что команда **echo `command`** удалит все символы перевода строки, которые будут выведены командой *`command`*.

Переменная $IFS обычно содержит символ перевода строки \n, как один из вариантов пробельного символа. Bash разобьет вывод команды *`command`*, по пробельным символам, на аргументы и передаст их команде **echo**, которая выведет эти аргументы, разделенные пробелами.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
-rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au

bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1
root root 362 Nov 7 2000 seconds.au
```

Это встроенная команда Bash и имеет внешний аналог `/bin/echo`.

```
bash$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

printf

printf -- команда форматированного вывода, расширенный вариант команды **echo** и ограниченный вариант библиотечной функции `printf()` в языке C, к тому же синтаксис их несколько отдичается друг от друга.

printf *format-string... parameter...*

Это встроенная команда Bash. Имеет внешний аналог `/bin/printf` или `/usr/bin/printf`. За более подробной информацией обращайтесь к страницам справочного руководства **man 1 printf** по системным командам.

Пример. printf в действии

```
PI=3,14159265358979
DecimalConstant=31373
Message1="Поздравляю,"
Message2="Землянин."

printf "пи с точностью до 2 знака после запятой = %1.2f" $PI
echo
printf "Число пи с точностью до 9 знака после запятой = %1.9f" $PI
# округляет правильно.

printf "\n"                                      # Перевод строки,
printf "Константа = \t%d\n" $DecimalConstant     # Вставлен символ
                                                 # табуляции (\t)
printf "%s %s \n" $Message1 $Message2

# Эмуляция функции 'sprintf' в языке C.
# Запись форматированной строки в переменную.
Pi12=$(printf "%1.12f" $PI)
echo "Число пи с точностью до 12 знака после запятой = $Pi12"

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg
```

Одно из полезных применений команды **printf** -- форматированный вывод сообщений об ошибках

```
E_BADDIR=65
var=nonexistent_directory
error()
{
  printf "$@" >&2
```

```
  # Форматированный вывод аргументов на stderr.
  echo
  exit $E_BADDIR
}

cd $var || error $"Невозможно перейти в каталог %s." "$var"
```

read

"Читает" значение переменной с устройства стандартного ввода -- `stdin`, в интерактивном режиме это означает клавиатуру. Ключ `-a` позволяет записывать значения в массивы.

Пример. Ввод значений переменных с помощью read

```
echo -n "дите значение переменной 'var1': "
# Ключ -n подавляет вывод символа перевода строки.

read var1
# Обратите внимание -- перед именем переменной отсутствует символ '$'.

echo "var1 = $var1"
echo

# Одной командой 'read' можно вводить несколько переменных.
echo -n "дите значения для переменных 'var2' и 'var3' (через пробел или
табуляцию): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
# Если было введено значение только одной переменной, то вторая оста-
нется "пустой".
```

Если команде **read** не была передано ни одной переменной, то ввод будет осуществлен в переменную $REPLY.

Пример. Пример использования команды read без указания переменной для ввода

Обычно, при вводе в окне терминала с помощью команды "read", символ **** служит для экранирования символа перевода строки. Ключ `-r` заставляет интерпретировать символ **** как обычный символ.

Команда **read** имеет ряд очень любопытных опций, которые позволяют выводить подсказку - приглашение ко вводу (prompt), и даже читать данные не дожидаясь нажатия на клавишу **ENTER**.

```
# Чтение данных, не дожидаясь нажатия на клавишу ENTER.

read -s -n1 -p "Нажмите клавишу " keypress
echo; echo "Была нажата клавиша "\"$keypress\""."

# -s   -- подавляет эхо-вывод, т.е. ввод с клавиатуры
#         не отображается на экране.
# -n N -- ввод завершается автоматически,
#         сразу же после ввода N-го символа.
# -p   -- задает вид строки подсказки - приглашения к вводу (prompt).
# Использование этих ключей немного осложняется тем, что они должны
следовать в определенном порядке.
```

Ключ `-n`, кроме всего прочего, позволяет команде **read** обнаруживать нажатие *курсорных* и некоторых других служебных клавиш.

Ключ `-t` позволяет ограничивать время ожидания ввода командой **read**.

Команда **read** может считывать значения для переменных из файла, перенаправленного на `stdin`. Если файл содержит не одну строку, то переменной будет присвоена только первая строка. Если команде **read** будет передано несколько переменных, то первая строка файла будет разбита, по пробелам, на несколько подстрок, каждая из которых будет записана в свою переменную. Будьте осторожны!

Файловая система.

cd

Уже знакомая нам команда **cd**, изменяющая текущий каталог, может быть использована в случаях, когда некоторую команду необходимо запустить только находясь в определенном каталоге.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

Команда **cd** с ключом `-P` (physical) игнорирует символические ссылки.

Команда "**cd -**" выполняет переход в каталог $OLDPWD -- предыдущий рабочий каталог.

pwd

Выводит название текущего рабочего каталога (Print Working Directory). Кроме того, имя текущего каталога хранится во внутренней переменной $PWD.

pushd, **popd**, **dirs**

Этот набор команд является составной частью механизма "закладок" на каталоги и позволяет перемещаться по каталогам вперед и назад в заданном порядке. Для хранения имен каталогов используется стек (LIFO -- "последний вошел, первый вышел").

pushd dir-name -- помещает имя текущего каталога в стек и осуществляет переход в каталог *dir-name*.

popd -- выталкивает, находящееся на вершине стека, имя каталога и одновременно осуществляет переход в каталог, оказавшийся на врешине стека.

dirs -- выводит содержимое стека каталогов (сравните с переменной $DIRSTACK). В случае успеха, обе команды -- **pushd** и **popd** автоматически вызывают **dirs**.

Эти команды могут оказаться весьма полезными, когда в сценарии нужно производить частую смену каталогов, но при этом не хочется жестко "зашивать" имена каталогов. Обратите внимание: содержимое стека каталогов постоянно хранится в переменной-массиве -- `$DIRSTACK`.

Пример. Смена текущего каталога

```
dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Команда 'dirs' будет вызвана автоматически
# (на stdout будет выведено содержимое стека).
echo "Выполнен переход в каталог `pwd`." # Обратные одиночные кавычки.

# Теперь можно выполнить какие либо действия
# в каталоге 'dir1'.
pushd $dir2
echo "Выполнен переход в каталог `pwd`."
```

```
# Теперь можно выполнить какие либо действия в каталоге 'dir2'.
echo "На вершине стека находится: $DIRSTACK."
popd
echo "Возврат в каталог `pwd`."

# Теперь можно выполнить какие либо действия в каталоге 'dir1'.
popd
echo "Возврат в первоначальный рабочий каталог `pwd`."
```

Переменные.

let

Команда **let** производит арифметические операции над переменными. В большинстве случаев, ее можно считать упрощенным вариантом команды expr.

Пример. Команда let, арифметические операции.

```
let a=11          # То же, что и 'a=11'
let a=a+5         # Эквивалентно "a = a + 5"
                  # (Двойные кавычки и дополнительные
                  # пробелы делают код более удобочитаемым)
echo "11 + 5 = $a"

let "a <<= 3"     # Эквивалентно  let "a = a << 3"
echo "\"\$a\" (=16) после сдвига влево на 3 разряда = $a"

let "a /= 4"      # Эквивалентно let "a = a / 4"
echo "128 / 4 = $a"

let "a -= 5"      # Эквивалентно let "a = a - 5"
echo "32 - 5 = $a"

let "a = a * 10"  # Эквивалентно let "a = a * 10"
echo "27 * 10 = $a"

let "a %= 8"      # Эквивалентно let "a = a % 8"
echo "270 mod 8 = $a  (270 / 8 = 33, остаток = $a)"
```

eval

```
eval arg1 [arg2] ... [argN]
```

Транслирует список аргументов, из списка, в команды.

Пример. Демонстрация команды eval

```
y=`eval ls -l`  # Подобно y=`ls -l`
echo $y         # но символы перевода строки не выводятся,
                # поскольку имя переменной не в кавычках.
echo "$y"       # Если имя переменной записать в кавычках -
                # символы перевода строки сохраняются.
y=`eval df`     # Аналогично y=`df`
echo $y         # но без символов перевода строки.
```

Пример. Принудительное завершение сеанса

```
y=`eval ps ax | sed -n '/ppp/p' | awk '{ print $1 }'`
# Выяснить PID процесса 'ppp'.
kill -9 $y   # "Прихлопнуть" его
# Предыдущие строки можно заменить одной строкой
#  kill -9 `ps ax | awk '/ppp/ { print $1 }'
```

```
chmod 666 /dev/ttyS3
# Завершенный, по сигналу SIGKILL, ppp изменяет права доступа
# к последовательному порту. Вернуть их в первоначальное состояние.

rm /var/lock/LCK..ttyS3   # Удалить lock-файл последовательного порта.
```

Команда **eval** может быть небезопасна. Если существует приемлемая альтернатива, то желательно воздерживаться от использования **eval**. Так, **eval $COMMANDS** исполняет код, который записан в переменную *COMMANDS*, которая, в свою очередь, может содержать весьма неприятные сюрпризы, например **rm -rf ***. Использование команды **eval**, для исполнения кода неизвестного происхождения, крайне опасно.

set

Команда **set** изменяет значения внутренних переменных сценария. Она может использоваться для переключения опций (ключей, флагов), определяющих поведение скрипта. Еще одно применение - сброс/установка позиционных параметров (аргументов), значения которых будут восприняты как результат работы команды (**set `command`**).

Пример. Установка значений аргументов с помощью команды set.

```
# script "set-test"
# Вызовите сценарий с тремя аргументами командной строки,
# например: "./set-test one two three".

echo
echo "Аргументы перед вызовом set \`uname -a\` :"
echo "Аргумент #1 = $1"
echo "Аргумент #2 = $2"
echo "Аргумент #3 = $3"

set `uname -a` # Изменение аргументов
               # значения которых берутся из результата
               # работы `uname -a`
echo $_
echo "Аргументы после вызова set \`uname -a\` :"
#  $1, $2, $3 и т.д. будут переустановлены в соответствии с выводом
#  команды `uname -a`
echo "Поле #1 'uname -a' = $1"
echo "Поле #2 'uname -a' = $2"
echo "Поле #3 'uname -a' = $3"
echo ---
echo $_    # ---
echo
```

Вызов **set** без параметров просто выводит список инициализированных переменных окружения.

```
bash$ set
AUTHORCOPY=/home/student/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/student/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
```

Если команда **set** используется с ключом "--", после которого следует переменная, то значение переменной переносится в позиционные параметры (аргументы). Если имя переменной отсутствует, то эта команда приводит к сбросу позиционных параметров.

unset

Команда **unset** удаляет переменную, фактически - устанавливает ее значение в *null*. Обратите внимание: эта команда не может сбрасывать позиционные параметры (аргументы).

```
bash$ unset PATH
bash$ echo $PATH
```

export

Команда **export** экспортирует переменную, делая ее доступной дочерним процессам. К сожалению, невозможно экспортировать переменную родительскому процессу. В качестве примера использования команды **export** можно привести сценарии инициализации системы, вызываемые в процессе загрузки, которые инициализируют и экспортируют переменные окружения, делая их доступными для пользовательских процессов.

Допускается объединение инициализации и экспорта переменной в одну инструкцию: **export var1=xxx**.

Однако, как заметил Greg Keraunen, в некоторых ситуациях такая комбинация может давать иной результат, нежели раздельная инициализация и экспорт.

```
bash$ export var=(a b); echo ${var[0]}
(a b)
bash$ var=(a b); export var; echo ${var[0]}
a
```

declare, **typeset**

Команды declare и typeset задают и/или накладывают ограничения на переменные.

readonly

То же самое, что и declare -r, делает переменную доступной только для чтения, т.е. переменная становится подобна константе. При попытке изменить значение такой переменной выводится сообщение об ошибке. Эта команда может расцениваться как квалификатор типа **const** в языке C.

getopts

Мощный инструмент, используемый для разбора аргументов, передаваемых сценарию из командной строки. Это встроенная команда Bash, но имеется и ее "внешний" аналог /usr/bin/getopt, а так же программистам, пишущим на C, хорошо знакома похожая библиотечная функция **getopt**. Она позволяет обрабатывать серии опций, объединенных в один аргумент и дополнительные аргументы, передаваемые сценарию (например, **`scriptname -abc -e /usr/local`**).

С командой **getopts** очень тесно взаимосвязаны скрытые переменные. `$OPTIND` -- указатель на аргумент (*OPTion INDex*) и `$OPTARG` (*OPTion ARGument*) -- дополнительный аргумент опции. Символ двоеточия, следующий за именем опции, указывает на то, что она имеет дополнительный аргумент.

Обычно **getopts** упаковывается в цикл while, в каждом проходе цикла извлекается очередная опция и ее аргумент (если он имеется), обрабатывается, затем уменьша-

ется на 1 скрытая переменная `$OPTIND` и выполняется переход к началу новой итерации.

1. Опциям (ключам), передаваемым в сценарий из командной строки, должен предшествовать символ "минус" (-) или "плюс" (+). Этот префикс (- или +) позволяет **getopts** отличать опции (ключи) от прочих аргументов. Фактически, **getopts** не будет обрабатывать аргументы, если им не предшествует символ - или +, выделение опций будет прекращено как только встретится первый аргумент.
2. Типичная конструкция цикла **while** с **getopts** несколько отличается от стандартной из-за отсутствия квадратных скобок, проверяющих условие продолжения цикла.
3. Пример **getopts**, заменившей устаревшую, и не такую мощную, внешнюю команду getopt.

```
while getopts ":abcde:fg" Option
# Начальное объявление цикла анализа опций.
# a, b, c, d, e, f, g -- это возможные опции (ключи).
# Символ : после опции 'e' указывает на то, что с данной опцией может
идти
# дополнительный аргумент.
do
  case $Option in
    a ) # Действия, предусмотренные опцией 'a'.
    b ) # Действия, предусмотренные опцией 'b'.
    ...
    e)  # Действия, предусмотренные опцией 'e', а так же необходимо об-
работать $OPTARG,
        # в которой находится дополнительный аргумент этой опции.
    ...
    g ) # Действия, предусмотренные опцией 'g'.
  esac
done
shift $(($OPTIND - 1))
# Перейти к следующей опции.
```

Управление сценарием.

source, . (точка)

Когда эта команда вызывается из командной строки, то это приводит к запуску указанного сценария. Внутри сценария, команда **`source file-name`** загружает файл `file-name`. Таким образом она очень напоминает директиву препроцессора языка C/C++ -- "#include". Может найти применение в ситуациях, когда несколько сценариев пользуются одним файлом с данными или библиотекой функций.

exit

Безусловное завершение работы сценария. Команде **exit** можно передать целое число, которое будет возвращено вызывающему процессу как код завершения. Вообще, считается хорошей практикой завершать работу сценария, за исключением простейших случаев, командой **`exit 0`**, чтобы проинформировать родительский процесс об успешном завершении.

Если сценарий завершается командой **exit** без аргументов, то в качестве кода завершения сценария принимается код завершения последней выполненной команды, не считая самой команды **exit**.

exec

Это встроенная команда интерпретатора shell, заменяет текущий процесс новым процессом, запускаемым командой exec. Обычно, когда командный интерпретатор

встречает эту команду, то он порождает дочерний процесс, чтобы исполнить команду. При использовании встроенной команды **exec**, оболочка не порождает еще один процесс, а заменяет текущий процесс другим. Для сценария это означает его завершение сразу после исполнения команды **exec**. По этой причине, если вам встретится **exec** в сценарии, то, скорее всего это будет последняя команда в сценарии.

Пример. Команда exec.

```
exec echo "Завершение \"$0\"."   # Это завершение работы сценария.
# Следующие ниже строки никогда не будут исполнены
echo "Эта строка никогда не будет выведена на экран."

exit 99                         #  Сценарий завершит работу не здесь.
                                #  Проверьте код завершения сценария
                                #+ командой 'echo $?'.
                                #  Он точно не будет равен 99.
```

shopt

Эта команда позволяет изменять ключи (опции) оболочки на лету. Ее часто можно встретить в стартовых файлах, но может использоваться и в обычных сценариях.

```
shopt -s cdspell
# Исправляет незначительные орфографические ошибки в именах
# каталогов в команде 'cd'
cd /hpme  # Oops! Имелось ввиду '/home'.
pwd       # /home
          # Shell исправил опечатку.
```

Команды.

true

Команда возвращает код завершения - ноль, или успешное завершение, и ничего больше.

```
# Бесконечный цикл
while true   # вместо ":"
do
   operation-1
   operation-2
   ...
   operation-n
   # Следует предусмотреть способ завершения цикла.
Done
```

false

Возвращает код завершения, свидетельствующий о неудаче, и ничего более.

```
# Цикл, который никогда не будет исполнен
while false
do
   # Следующий код не будет исполнен никогда.
   operation-1
   operation-2
   ...
   operation-n
done
```

type [cmd]

Очень похожа на внешнюю команду which, **type cmd** выводит полный путь к "cmd". В отличие от **which**, **type** является внутренней командой Bash. С опцией `-a` не только различает ключевые слова и внутренние команды, но и определяет местоположение внешних команд с именами, идентичными внутренним.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[
```

Команды управления заданиями.

Некоторые из нижеследующих команд принимают, в качестве аргумента, "идентификатор задания".

jobs

Выводит список заданий, исполняющихся в фоне. Команда **ps** более информативна.

Задания и *процессы* легко спутать. Некоторые внутренние команды, такие как **kill**, **disown** и **wait** принимают в качестве параметра либо номер задания, либо номер процесса. Команды **fg**, **bg** и **jobs** принимают только номер задания.

```
bash$ sleep 100 &
[1] 1384
bash $ jobs
[1]+  Running                 sleep 100 &
```

"1" -- это номер задания (управление заданиями осуществляет текущий командный интерпретатор), а "1384" -- номер процесса (управление процессами осуществляется системой). Завершить задание/процесс ("прихлопнуть") можно либо командой **kill %1**, либо **kill 1384**.

disown

Удаляет задание из таблицы активных заданий командной оболочки.

fg, **bg**

Команда **fg** переводит задание из фона на передний план. Команда **bg** перезапускает приостановленное задание в фоновом режиме. Если эти команды были вызваны без указания номера задания, то они воздействуют на текущее исполняющееся задание.

wait

Останавливает работу сценария до тех пор пока не будут завершены все фоновые задания или пока не будет завершено задание/процесс с указанным номером задания/PID процесса. Возвращает код завершения указанного задания/процесса.

Вы можете использовать команду **wait** для предотвращения преждевременного завершения сценария до того, как завершит работу фоновое задание.

Пример. Ожидание завершения процесса перед тем как продолжить работу

```
ROOT_UID=0   # Только пользователь с $UID = 0 имеет привилегии root.
E_NOTROOT=65
E_NOPARAMS=66
```

```
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Для запуска этого сценария вы должны обладать привилегиями
root."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` имя-файла"
  exit $E_NOPARAMS
fi

echo "Обновляется база данных 'locate'..."
echo "Это может занять продолжительное время."
updatedb /usr &     # Должна запускаться с правами root.

wait
# В этом месте сценарий приостанавливает свою работу до тех пор,
# пока не отработает 'updatedb'.
# Желательно обновить базу данных перед тем как выполнить поиск файла.

locate $1

# В худшем случае, без команды wait,
# сценарий завершил бы свою работу до того,
# как завершила бы работу утилита 'updatedb',
# сделав из нее "осиротевший" процесс.
```

Команда **wait** может принимать необязательный параметр - номер задания/процесса, например, **wait %1** или **wait $PPID**.

suspend

Действует аналогично нажатию на комбинацию клавиш **Control+-Z**, за исключением того, что она приостанавливает работу командной оболочки.

logout

Завершает сеанс работы командной оболочки, можно указать необязательный код завершения.

times

Выдает статистику исполнения команд в единицах системного времени, в следующем виде:

```
0m0.020s 0m0.020s
```

Имеет весьма ограниченную сферу применения, так как сценарии крайне редко подвергаются профилированию.

kill

Принудительное завершение процесса путем передачи ему соответствующего сигнала.

Пример. Сценарий, завершающий себя сам с помощью команды kill.

```
kill $$  # Сценарий завершает себя сам.
         # Надеюсь вы еще не забыли, что "$$" -- это PID сценария.
echo "Эта строка никогда не будет выведена."
# Вместо него на stdout будет выведено сообщение "Terminated".
```

autoload

Перенесена в Bash из *ksh*. Если функция объявлена как **autoload**, то она будет загружена из внешнего файла в момент первого вызова. Такой прием помогает экономить системные ресурсы.

Обратите внимание: **autoload** не является частью ядра Bash. Ее необходимо загрузить с помощью команды **enable -f** (см. выше).

Таблица Идентификация заданий

Нотация	Описание
%N	Номер задания [N]
%S	Вызов (командная строка) задания, которая начинается со строки *S*
%?S	Вызов (командная строка) задания, которая содержит строку *S*
%%	"текущее" задание (последнее задание приостановленное на переднем плане или запущенное в фоне)
%+	"текущее" задание (последнее задание приостановленное на переднем плане или запущенное в фоне)
%-	Последнее задание
$!	Последний фоновый процесс

Внешние команды, программы и утилиты.

Благодаря стандартизации набора команд UNIX-систем, сценарии, на языке командной оболочки, могут быть легко перенесены из системы в систему практически без изменений. Мощь сценариев складывется из наборв системных команд и директив командной оболочки с простыми программными конструкциями.

Базовые команды.

Первая команда, с которой сталкиваются новички.

ls

Команда вывода "списка" файлов. Многие недооценивают всю мощь этой скромной команды. Например, с ключом `-R`, рекурсивный обход дерева каталогов, командв **ls** выводит содержимое каталогов в виде древовидной структуры. Вот еще ряд любопытных ключей (опций) команды **ls**: `-S` -- сортировка по размеру файлов, `-t` -- сортировка по времени последней модификации файла и `-i` - выводит список файлов с их inode.

Пример. Создание оглавления диска для записи CDR, с помощью команды ls.

```
# Сценарий, автоматизирующий процесс прожигания CDR.

SPEED=2           # Если ваше "железо" поддерживает более высокую
                  # скорость записи -- можете увеличить этот параметр
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
DEFAULTDIR=/opt   # В этом каталоге находятся файлы,
                  # которые будут записаны на CD.
                  # Каталог должен существовать.
# Используется пакет "cdrecord" от Joerg Schilling.
# Если этот сценарий предполагается запускать с правами
# обычного пользователя,
```

```
# то необходимо установить флаг suid на cdrecord
# (chmod u+s /usr/bin/cdrecord,
# эта команда должна быть выполнена root-ом).

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Каталог по-умолчанию, если иной каталог не задан из командной стро-
ки.
else
    IMAGE_DIRECTORY=$1
fi

# Создать файл "table of contents".
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# Ключ "l" -- "расширенный" формат вывода списка файлов.
# Ключ "R" -- рекурсивный обход дерева каталогов.
# Ключ "F" -- добавляет дополнительные метки к именам файлов (к именам
каталогов добавдяет оконечный символ /).
echo "Создано оглавление."

# Создать iso-образ.
mkisofs -r -o $IMAGFILE $IMAGE_DIRECTORY
echo "Создан iso-образ файловой системы ISO9660 ($IMAGEFILE)."

# "Прожигание" CDR.
cdrecord -v -isosize speed=$SPEED dev=0,0 $IMAGEFILE
echo "Запись диска."
echo "Наберитесь терпения, это может потребовать некоторого времени."
```

cat, **tac**

cat -- это акроним от *concatenate*, выводит содержимое списка файлов на `stdout`. Для объединения файлов в один файл может использоваться в комбинации с операциями перенаправления (> или >>).

```
cat filename cat file.1 file.2 file.3 > file.123
```

Ключ `-n`, команды **cat**, вставляет порядковые номера строк в выходном файле. Ключ `-b` -- нумерут только не пустые строки. Ключ `-v` выводит непечатаемые символы в нотации с символом ^. Ключ `-s` заменяет несколько пустых строк, идущих подряд, одной пустой строкой.

tac -- выводит содержимое файлов в обратном порядке, от последней строки к первой.

rev

выводит все строки файла задом наперед на `stdout`. Это не то же самое, что **tac**. Команда **rev** сохраняет порядок следования строк, но переворачивает каждую строку задом наперед.

```
bash$ cat file1.txt
Это строка 1.
 Это строка 2.

bash$ tac file1.txt
Это строка 2.
 Это строка 1.

bash$ rev file1.txt
.1 акортс отЭ
 .2 акортс отЭ
```

Более сложные команды.

find

-exec *COMMAND* \;

Для каждого найденного файла, соответствующего заданному шаблону поиска, выполняет команду *COMMAND*. Командная строка должна завершаться последовательностью символов \; (здесь символ ";" экранирован обратным слэшем, чтобы информировать командную оболочку о том, что символ ";" должен быть передан команде **find** как обычный символ). Если *COMMAND* содержит {}, то **find** подставляет полное имя найденного файла вместо "{}".

```
bash$ find ~/ -name '*.txt'
```

```
find /home/student/projects -mtime 1
#  Найти все файлы в каталоге /home/student/projects
#  и вложенных подкаталогах,
#  которые изменялись в течение последних суток.
#
#  mtime = время последнего изменения файла
#  ctime = время последнего изменения атрибутов файла
# (через 'chmod' или как-то иначе)
#  atime = время последнего обращения к файлу
DIR=/home/student/junk_files
find "$DIR" -type f -atime +5 -exec rm {} \;
#  Удалить все файлы в каталоге "/home/student/junk_files"
#+ к которым не было обращений в течение последних 5 дней.
#
#  "-type filetype", где
#  f = обычный файл
#  d = каталог, и т.п.
#  (Полный список ключей вы найдете в 'man find'.)
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-
9][0-9]*' {} \;
# Поиск всех IP-адресов (xxx.xxx.xxx.xxx) в файлах каталога  /etc.
# Однако эта команда выводит не только IP-адреса, как этого избежать?
# Примерно так:
find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
# [:digit:] -- один из символьных классов
# введен в стандарт POSIX 1003.2.
```

xargs

Команда передачи аргументов указанной команде. Она разбивает поток аргументов на отдельные составляющие и поочередно передает их заданной команде для обработки. Эта команда может рассматриваться как мощная замена обратным одиничным кавычкам. Зачастую, когда команды, заключенные в обратные одиночные кавычки, завершаются с ошибкой too many arguments (слишком много аргументов), использование **xargs** позволяет обойти это ограничение. Обычно, **xargs** считывает список аргументов со стандартного устройства ввода `stdin` или из канала (конвейера), но может считывать информацию и из файла.

Если команда не задана, то по-умолчанию выполняется echo. При передаче аргументов по конвейеру, **xargs** допускает наличие пробельных символов и символов перевода строки, которые затем автоматически отбрасываются.

```
bash$ ls -l
total 0
-rw-rw-r--    1 student  student         0 Jan 29 23:58 file1
```

```
-rw-rw-r--    1 student  student         0 Jan 29 23:58 file2
```

```
bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 student student 0 Jan 29 23:58 file1 -rw-rw-r-- 1
student student 0 Jan 29 23:58 file2
```

ls | xargs -p -l gzip -- упакует с помощью gzip все файлы в текущем каталоге, выводя запрос на подтверждение для каждого файла.

xargs имеет очень любопытный ключ `-n NN`, который ограничивает количество передаваемых аргументов за один "присест" числом *NN*.

ls | xargs -n 8 echo -- выведет список файлов текущего каталога в 8 колонок.

Еще одна полезная опция -- `-0`, в комбинации с **find -print0** или **grep -lZ** позволяет обрабатывать аргументы, содержащие пробелы и кавычки.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f
grep -rliwZ GUI / | xargs -0 rm -f
```

Пример. Использование команды xargs для мониторинга системного журнала.

```
# Создание временного файла мониторинга в текщем каталоге,
# куда переписываются несколько последних строк из /var/log/messages.
# Обратите внимание: если сценарий запускается обычным пользователем,
# то файл /var/log/messages должен быть доступен на чтение этому поль-
зователю.
#         #root chmod 644 /var/log/messages
LINES=5
( date; uname -a ) >>logfile
# Время и информация о системе
echo ---------------------------------------------------- >>logfile
tail -$LINES /var/log/messages | xargs |  fmt -s >>logfile
echo >>logfile
echo >>logfile
# Упражнение:
# --------
#  Измените сценарий таким образом, чтобы он мог
#  отслеживать изменения в /var/log/messages
#  с интервалом в 20 минут.
#  Подсказка: воспользуйтесь командой "watch".
```

expr

Универсальный обработчик выражений: вычисляет заданное выражение (аргументы должны отделяться пробелами). Выражения могут быть арифметическими, логическими или строковыми.

```
expr 3 + 5
```

возвратит 8

```
expr 5 % 3
```

возвратит 2

```
expr 5 \* 3
```

возвратит 15

В арифметических выражениях, оператор умножения обязательно должен экранироваться обратным слэшем.

```
y=`expr $y + 1`
```

Операция инкремента переменной, то же самое, что и **let y=y+1**, или **y=$(($y+1))**. Пример подстановки арифметических выражений.

z=`expr substr $string $position $length`

Извлекает подстроку длиной $length символов, начиная с позиции $position.

Пример. Пример работы с expr.

```
# Арифметические операции
# -------------- --------
echo "Арифметические операции"
a=`expr 5 + 3`
echo "5 + 3 = $a"
a=`expr $a + 1`
echo "a + 1 = $a"
echo "(инкремент переменной)"
a=`expr 5 % 3`
# остаток от деления (деление по модулю)
echo "5 mod 3 = $a"
# Логические операции
# ---------- --------
# Возвращает 1 если выражение истинноо, 0 -- если ложно,
# в противоположность соглашениям, принятым в Bash.
x=24
y=25
b=`expr $x = $y`          # Сравнение.
echo "b = $b"             # 0  ( $x -ne $y )
a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, поэтому...'
echo "Если a > 10, то b = 0 (ложь)"
echo "b = $b"             # 0  ( 3 ! -gt 10 )
b=`expr $a \< 10`
echo "Если a < 10, то b = 1 (истина)"
echo "b = $b"             # 1  ( 3 -lt 10 )
echo
# Обратите внимание на необходимость экранирования операторов.
b=`expr $a \<= 3`
echo "Если a <= 3, то b = 1 (истина)"
echo "b = $b"             # 1  ( 3 -le 3 )
# Существует еще оператор "\>=" (больше или равно).

# Операции сравнения
# -------- ---------
echo "Операции сравнения"
echo
a=zipper
echo "a is $a"
if [ `expr $a = snap` ]
then
   echo "a -- это не zipper"
fi

# Операции со строками
# -------- -- --------
a=1234zipper43231
echo "Строка над которой производятся операции: \"$a\"."

# length: длина строки
b=`expr length $a`
echo "длина строки \"$a\" равна $b."
# index: позиция первого символа подстроки в строке
b=`expr index $a 23`
echo "Позиция первого символа \"2\" в строке \"$a\" : \"$b\"."
# substr: извлечение подстроки, начиная с заданной позиции, указанной
длины
b=`expr substr $a 2 6`
```

```
echo "Подстрока в строке \"$a\", начиная с позиции 2,\
и длиной в 6 символов: \"$b\"."

#  При выполнении поиска по шаблону, по-умолчанию поиск
#  начинается с ***начала*** строки.
#
#       Использование регулярных выражений
b=`expr match "$a" '[0-9]*'`                #  Подсчет количества цифр.
echo Количество цифр с начала строки \"$a\" : $b.
b=`expr match "$a" '\([0-9]*\)'`            #  Обратите внимание на эк-
ранирование круглых скобок
echo "Цифры, стоящие в начале строки \"$a\" : \"$b\"."
```

Команды для работы с датой и временем.

Время/дата и измерение интервалов времени.

date

Команда **date** без параметров выводит дату и время на стандартное устройство вывода `stdout`. Она становится гораздо интереснее при использовании дополнительных ключей форматирования вывода.

Пример. Команда date

```
echo "Количество дней, прошедших с начала года: `date +%j`."
# Символ '+' обязателен при использовании форматирующего аргумента
# %j,  возвращающего количество дней, прошедших с начала года.

echo "Количество секунд, прошедших с 01/01/1970 : `date +%s`."
#  %s количество секунд, прошедших с начала "эпохи UNIX",
#  но насколько этот ключ полезен?

prefix=temp
suffix=`eval date +%s`  # Ключ "+%s" характерен для GNU-версии 'date'.
filename=$prefix.$suffix
echo $filename
#  Прекрасный способ получения "уникального"
#  имени для временного файла,
# даже лучше, чем с использованием $$.
```

Ключ `-u` дает UTC время (Universal Coordinated Time -- время по Гринвичу).

touch

Утилита устанавливает время последнего обращения/изменения файла в текущее системное время или в заданное время, но так же может использоваться для создания нового пустого файла. Команда **`touch zzz`** создаст новый пустой файл с именем `zzz`, если перед этим файл `zzz` отсутствовал. Кроме того, такие пустые файлы могут использоваться для индикации, например, времени последнего изменения в проекте.

Эквивалентом команды **touch** могут служить **`: >> newfile`** или **`>> newfile`** (для обычных файлов).

at

Команда **at** -- используется для запуска заданий в заданное время. В общих чертах она напоминает crond, однако, **at** используется для однократного запуска набора команд.

`at 2pm January 15` -- попросит ввести набор команд, которые необходимо запустить в указанное время. Эти команды должны быть совместимыми со сценариями

командной оболочки. Ввод завершается нажатием комбинации клавиш Ctl-D.

Ключ `-f` или операция перенаправления ввода (<), заставляет **at** прочитать список команд из файла. Этот файл должен представлять из себя обычный сценарий, на языке командной оболочки и, само собой разумеется, такой сценарий должен быть неинтерактивным. Может использоваться совместно с командой run-parts для запуска различных наборов сценариев.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

Команда **batch**, управляющая запуском заданий, напоминает команду **at**, но запускает список команд только тогда, когда загруженность системы упадет ниже `.8`. Подобно команде **at**, с ключом `-f`, может считывать набор команд из файла.

cal

Выводит на `stdout` аккуратно отформатированный календарь на текущий месяц. Может выводить календарь за определенный год.

sleep

Приостанавливает исполнение сценария на заданное количество секунд, ничего не делая. Может использоваться для синхронизации процессов, запущенных в фоне, проверяя наступление ожидаемого события так часто, как это необходимо. Например.

```
sleep 3
# Пауза, длительностью в 3 секунды.
```

Команда **sleep** по-умолчанию принимает количество секунд, но ей можно передать и количество часов и минут и даже дней.

```
sleep 3 h
# Приостановка на 3 часа!
```

Для запуска команд через заданные интервалы времени лучше использовать watch

usleep

Microsleep (здесь символ "u" должен читаться как буква греческого алфавита -- "мю", или префикс микро). Это то же самое, что и **sleep**, только интервал времени задается в микросекундах. Может использоваться для очень тонкой синхронизации процессов.

```
usleep 30
# Приостановка на 30 микросекунд.
```

hwclock, **clock**

Команда **hwclock** используется для получения доступа или коррекции аппаратных часов компьютера. С некоторыми ключами требует наличия привилегий root. Сценарий `/etc/rc.d/rc.sysinit` использует команду **hwclock** для установки системного времени во время загрузки.

Команда **clock** -- это синоним команды **hwclock**.

Команды обработки текста.

sort

Сортирует содержимое файла, часто используется как промежуточный фильтр в конвейерах. Эта команда сортирует поток текста в порядке убывания или возраста-

ния, в зависимости от заданных опций. Ключ `-m` используется для сортировки и объединения входных файлов. В *странице info* перечислено большое количество возможных вариантов ключей.

tsort

Топологическая сортировка, считывает пары строк, разделенных пробельными символами, и выполняет сортировку, в зависимости от заданного шаблона.

uniq

Удаляет повторяющиеся строки из отсортированного файла. Эту команду часто можно встретить в конвейере с командой sort.

expand, **unexpand**

Команда **expand** преобразует символы табуляции в пробелы. Часто используется в конвейерной обработке текста.

Команда **unexpand** преобразует пробелы в символы табуляции. Т.е. она является обратной по отношению к команде **expand**.

cut

Команда предназначена для извлечения отдельных полей из текстовых файлов. Напоминает команду **`print $N`** в awk, но более ограничена в своих возможностях. В простейших случаях может быть неплохой заменой **awk** в сценариях. Особую значимость, для команды **cut**, представляют ключи `-d` (разделитель полей) и `-f` (номер(а) поля(ей)).

Использование команды **cut** для получения списка смонтированных файловых систем:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

Использование команды **cut** для получения версии ОС и ядра:

```
uname -a | cut -d" " -f1,3,11,12
```

Использование команды **cut** для извлечения заголовков сообщений из электронных писем:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME3
 Spam complaint
 Re: Spam complaint
```

Использование команды **cut** при разборе текстового файла:

```
# Список пользователей в /etc/passwd.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME)
do
  echo $user
done


cut -d ' ' -f2,3 filename эквивалентно awk -F'[ ]' '{ print $2, $3 }'
filename
```

paste

Используется для объединения нескольких файлов в один многоколоночный файл.

join

Может рассматриваться как команда, родственная команде **paste**. Эта мощная утилита позволяет объединять два файла по общему полю, что представляет собой упрощенную версию реляционной базы данных.

Команда **join** оперирует только двумя файлами и объедияет только те строки, которые имеют общее поле (обычно числовое), результат объединения выводится на `stdout`. Объединяемые файлы должны быть отсортированы по ключевому полю.

```
File: 1.data
100 Shoes
200 Laces
300 Socks

File: 2.data
100 $40.00
200 $1.00
300 $2.00

bash$ join 1.data 2.data
File: 1.data 2.data

100 Shoes $40.00
200 Laces $1.00
300 Socks $2.00
```

head

Выводит начальные строки из файла на `stdout` (по-умолчанию -- `10` строк, но это число можно задать иным). Эта команда имеет ряд интересных ключей.

Пример. Какие из файлов являются сценариями?

```
TESTCHARS=2     # Проверяются первые два символа.
SHABANG='#!'    # Сценарии как правило начинаются с "sha-bang."

for file in *  # Обход всех файлов в каталоге.
do
  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
  #      head -c2                     #!
  #  Ключ '-c' в команде "head" выводит заданное
  #+ количество символов, а не строк.
  then
    echo "Файл \"$file\" -- сценарий."
  else
    echo "Файл \"$file\" не является сценарием."
  fi
done
```

tail

Выводит последние строки из файла на `stdout` (по-умолчанию -- 10 строк). Обычно используется для мониторинга системных журналов. Ключ `-f`, позволяет вести непрерывное наблюдение за добавляемыми строками в файл.

grep

Многоцелевая поисковая утилита, использующая регулярные выражения. Изначально это была команда в древнем строчном редакторе **ed**, **`g/re/p`**, что означает -- *global - regular expression - print*.

grep *`pattern`* [*`file`*...]

Поиск участков текста в файле(ах), соответствующих шаблону *pattern*, где *pattern* может быть как обычной строкой, так и регулярным выражением.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

Если файл(ы) для поиска не задан, то команда **grep** работает как фильтр для устройства `stdout`, например в конвейере.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
 901 pts/1    S      0:00 grep clock
```

`-i` -- выполняется поиск без учета регистра символов.

`-w` -- поиск совпадений целого слова.

`-l` -- вывод только имен файлов, в которых найдены участки, совпадающие с заданным образцом/шаблоном, без вывода совпадающих строк.

`-r` -- (рекурсивный поиск) поиск выполняется в текущем каталоге и всех вложенных подкаталогах.

The `-n` option lists the matching lines, together with line numbers.

egrep -- то же самое, что и **grep -E**. Эта команда использует несколько отличающийся, расширенный набор регулярных выражений, что позволяет выполнять поиск более гибко.

fgrep -- то же самое, что и **grep -F**. Эта команда выполняет поиск строк символов (не регулярных выражений), что несколько увеличивает скорость поиска.

Утилита **agrep** имеет более широкие возможности поиска приблизительных совпадений. Образец поиска может отличаться от найденной строки на указанное число символов.

Для поиска по сжатым файлам следует использовать утилиты **zgrep**, **zegrep** или **zfgrep**. Они с успехом могут использоваться и для не сжатых файлов, но в этом случае они уступают в скорости обычным **grep**, **egrep** и **fgrep**. Они очень удобны при выполнении поиска по смешенному набору файлов - когда одни файлы сжаты, а другие нет.

`Для поиска по bzip-файлам используйте` **`bzgrep.`**

sed, **awk**

Скриптовые языки, специально разработанные для анализа текстовых данных.

sed

Неинтерактивный "потоковый редактор". Широко используется в сценариях на языке командной оболочки.

awk

Утилита контекстного поиска и преобразования текста, замечательный инструмент для извлечения и/или обработки полей (колонок) в структурированных текстовых файлах. Синтаксис awk напоминает язык C.

wc

wc -- "word count", счетчик слов в файле или в потоке:

```
bash $ wc /usr/doc/sed-3.02/README
20     127     838 /usr/doc/sed-3.02/README
[20 строк  127 слов  838 символов]
```

wc -w подсчитывает только слова.

wc -l подсчитывает только строки.

wc -c подсчитывает только символы.

wc -L возвращает длину наибольшей строки.

Подсчет количества *.txt*-файлов в текущем каталоге с помощью **wc**:

```
$ ls *.txt | wc -l
# Эта команда будет работать, если ни в одном из имен файлов "*.txt"
нет символа перевода строки.
# Альтернативный вариант:
find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)
```

Подсчет общего размера файлов, чьи имена начинаются с символов, в диапазоне d - h

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Использование **wc** для подсчета количества вхождений слова "Linux" в основной исходный файл с текстом этого руководства.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

Отдельные команды располагают функциональностью **wc** в виде своих ключей.

tr

Замена одних символов на другие.

В отдельных случаях символы необходимо заключать в кавычки и/или квадратные скобки. Кавычки предотвращают интерпретацию специальных символов командной оболочкой. Квадратные скобки должны заключаться в кавычки.

Команда **tr "A-Z" "*" <filename** или **tr A-Z * <filename** заменяет все символы верхнего регистра в `filename` на звездочки (вывод производится на `stdout`). В некоторых системах этот вариант может оказаться неработоспособным, тогда попробуйте **tr A-Z '[**]'**.

Ключ `-d` удаляет символы из заданного диапазона.

```
echo "abcdef"                  # abcdef
echo "abcdef" | tr -d b-d      # aef
tr -d 0-9 <filename
# Удалит все цифровые символы из файла "filename".
```

Ключ `--squeeze-repeats` (`-s`) удалит все повторяющиеся последовательности символов. Может использоваться для удаления лишних пробельных символов.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

Ключ `-c` "complement" *заменит* символы в соответствии с шаблоном. Этот ключ воздействует только на те символы, которые НЕ соответствуют заданному шаблону.

```
bash$ echo "acfdeb123" | tr -c b-d +
```

```
+c+d+b++++
```

Обратите внимание: команда **tr** корректно распознает символьные классы POSIX.

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Пример. Преобразование символов в верхний регистр.

```
E_BADARGS=65

if [ -z "$1" ]  # Стандартная проверка командной строки.
then
  echo "Порядок использования: `basename $0` filename"
  exit $E_BADARGS
fi
tr a-z A-Z <"$1"
# Тот же эффект можно получить при использовании символьных классов
POSIX:
#        tr '[:lower:]' '[:upper:]' <"$1"
```

Пример. Преобразование текстового файла из формата DOS в формат UNIX.

```
#!/bin/bash
# du.sh: Преобразование текстового файла из формата DOS в формат UNIX.
E_WRONGARGS=65
if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` filename-to-convert"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # Возврат каретки.
# Строки в текстовых файлах DOS завершаются комбинацией символов CR-LF.
tr -d $CR < $1 > $NEWFILENAME
# Удалить символы CR и записать в новый файл.

echo "Исходный текстовый файл: \"$1\"."
echo "Преобразованный файл: \"$NEWFILENAME\"."
```

fold

Выравнивает текст по ширине, разрывая, если это необходимо, слова. Особый интерес представляет ключ `-s`, который производит перенос строк по пробелам, стараясь не разрывать слова.

fmt

Очень простая утилита форматирования текста, чаще всего используемая как фильтр в конвейерах для того, чтобы выполнить "перенос" длинных строк текста.

Пример. Отформатированный список файлов.

```
#!/bin/bash
WIDTH=40                    # 40 символов в строке.
b=`ls /usr/local/bin`       # Получить список файлов...
echo $b | fmt -w $WIDTH
# То же самое можно выполнить командой
#  echo $b | fold - -s -w $WIDTH
```

column

Форматирование по столбцам. Эта утилита преобразует текст, например какой ли-

бо список, в табличное, более "удобочитаемое", представление, вставляя символы табуляции по мере необходимости.

Пример. Пример форматирования списка файлов в каталоге.

```
 (printf "PERMISSIONS LINKS OWNER GROUP SIZE DATE TIME PROG-NAME\n" \
; ls -l | sed 1d) | column -t

#  Команда "sed 1d" удаляет первую строку, выводимую командой ls,
#+ (для локали "C" это строка:  "total        N",
#+ где "N" -- общее количество файлов.
# Ключ -t, команды "column", означает "табличное" представление.
```

colrm

Утилита удаления колонок. Удаляет колонки (столбцы) сиволов из файла и выводит результат на `stdout`. **`colrm 2 4 <filename`** -- удалит символы со 2-го по 4-й включительно, в каждой строке в файле `filename`.

Если файл содержит символы табуляции или непечатаемые символы, то результат может получиться самым неожиданным. В таких случаях, как правило, утилиту **colrm**, в конвейере, окружают командами expand и **unexpand**.

nl

Нумерует строки в файле. **`nl filename`** -- выведет файл `filename` на `stdout`, и в начале каждой строки вставит ее порядковый номер, счет начинается с первой непустой строки. Если файл не указывается, то принимается ввод со `stdin`.

Вывод команды **nl** очень напоминает **`cat -n`**, однако, по-умолчанию **nl** не нумерует пустые строки.

Пример. Самонумерующийся сценарий.

```
# Сценарий выводит себя сам на stdout дважды, нумеруя строки сценария.
# 'nl' вставит для этой строки номер 3,
# поскольку она не нумерует пустые строки.
# 'cat -n' вставит для этой строки номер 5.
nl `basename $0`
echo; echo  # А теперь попробуем вывести текст сценария с помощью 'cat
-n'
cat -n `basename $0`
# Различия состоят в том, что 'cat -n' нумерует все строки.
# Обратите внимание: 'nl -ba' -- сделает то же самое.
```

pr

Подготовка файла к печати. Утилита производит разбивку файла на страницы, приводя его в вид пригодный для печати или для вывода на экран. Разнообразные ключи позволяют выполнять различные манипуляции над строками и колонками, соединять строки, устанавливать поля, нумеровать строки, добавлять колонтитулы и многое, многое другое. Утилита **pr** соединяет в себе функциональность таких команд, как **nl**, **paste**, **fold**, **column** и **expand**.

`pr -o 5 --width=65 fileZZZ | more` -- выдаст хорошо оформленное и разбитое на страницы содержимое файла `fileZZZ`.

Хочу особо отметить ключ `-d`, который выводит строки с двойным интервалом (тот же эффект, что и **sed -G**).

iconv

Утилита преобразования текста из одной кодировки в другую. В основном используется для нужд локализации.

recode

Может рассматриваться как разновилность утилиты **iconv**, описанной выше. Универсальная утилита для преобразования текстовой информации в различные кодировки.

groff, **tbl**, **eqn**

groff -- это еще один язык разметки текста и форматированного вывода. Является расширенной GNU-версией пакета **roff/troff** в UNIX-системах.

tbl -- утилита обработки таблиц, должна рассматриваться как составная часть **groff**, так как ее задачей является преобразование таблиц в команды **groff**.

eqn -- утилита преобразования математических выражений в команды **groff**.

lex, **yacc**

lex -- утилита лексического разбора текста. В Linux-системах заменена на свободно распространяемую утилиту **flex**.

yacc -- утилита для создания синтаксических анализаторов, на основе набора грамматик, задаваемых разработчиком. В Linux-системах, эта утилита заменена на свободно распространяемую утилиту **bison**.

Команды для работы с файлами и архивами.

Архивация.

tar

Стандартная, для UNIX, утилита архивирования. Первоначально -- это была программа *Tape ARchiving*, которая впоследствии переросла в универсальный пакет, который может работать с любыми типами устройств. В GNU-версию tar была добавлена возможность одновременно производить сжатие tar-архива, например команда **tar czvf archive_name.tar.gz *** создает tar-архив дерева подкаталогов и вызывает gzip для выполнения сжатия, исключение составляют скрытые файлы в текущем каталоге (**$PWD**).

Некоторые, часто используемые, ключи команды **tar**:

1. `-c` -- создать (create) новый архив
2. `-x` -- извлечь (extract) файлы из архива
3. `--delete` -- удалить (delete) файлы из архива
4. `-r` -- добавить (append) файлы в существующий архив
5. `-A` -- добавить (append) *tar*-файлы в существующий архив
6. `-t` -- список файлов в архиве (содержимое архива)
7. `-u` -- обновить (update) архив
8. `-d` -- операция сравнения архива с заданной файловой системой
9. `-z` -- обработка архива с помощью gzip

   (Сжатие или разжатие, в зависимости от комбинации сопутствующих ключей `-c` или `-x`)
10. `-j` -- обработка архива с помошью bzip2

Сжатие.

gzip

Стандартная GNU/UNIX утилита сжатия, заменившая более слабую, и к тому же проприетарную, утилиту **compress**. Соответствующая утилита декомпрессии (разжатия) -- **gunzip**, которая является эквивалентом команды **gzip -d**.

Для работы со сжатыми файлами в конвейере используется фильтр **zcat**, который выводит результат своей работы на `stdout`, допускает перенаправление вывода. Фактически это та же команда **cat**, только приспособленная для работы со сжатыми файлами (включая файлы, сжатые утилитой **compress**). Эквивалент команды **zcat** -- **gzip -dc**.

В некоторых коммерческих версиях UNIX, команда **zcat** является синонимом команды **uncompress -c**, и не может работать с файлами, сжатыми с помощью *gzip*.

bzip2

Альтернативная утилита сжатия, обычно дает более высокую степень сжатия (но при этом работает медленнее), чем **gzip**, особенно это проявляется на больших файлах. Соответствующая утилита декомпрессии -- **bunzip2**.

В современные версии tar добавлена поддержка **bzip2**.

zip, **unzip**

Кроссплатформенная утилита архивирования и сжатия, совместимая, по формату архивного файла, с утилитой DOS -- *pkzip.exe*.

unarc, **unarj**, **unrar**

Этот набор утилит предназначен для распаковки архивов, созданных с помощью DOS архиваторов -- *arc.exe*, *arj.exe* и *rar.exe*.

Получение сведений о файлах.

file

Утилита идентификации файлов. Команда **`file file-name`** верне тип файла `file-name`, например, `ascii text` или `data`. Для этого она анализирует сигнатуру, или магическое число и сопоставляет ее со списком известных сигнатур из `/usr/share/magic`, `/etc/magic` или `/usr/lib/magic`

`-f` -- ключ пакетного режима работы утилиты **file**, в этом случае утилита принимает список анализируемых имен файлов из заданного файла. Ключ `-z` используется для анализа файлов в архиве.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16
13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last
modified: Sun Sep 16 13:34:51 2001, os: Unix)
```

which

Команда **which command-xxx** вернет полный путь к "command-xxx". Очень полезна для того, чтобы узнать -- установлена ли та или иная утилита в системе.

```
$bash which rm

/usr/bin/rm
```

whereis

Очень похожа на **which**, упоминавшуюся выше. Команда **whereis command-xxx**

вернет полный путь к "command-xxx", но кроме того, еще и путь к *manpage* -- файлу, странице справочника по заданной утилите.

```
$bash whereis rm
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

Утилита **whatis filexxx** отыщет "filexxx" в своей базе данных. Может рассматриваться как упрощенный вариант команды **man**.

```
$bash whatis whatis
whatis               (1)  - search the whatis database for complete
words
```

Пример. Исследование каталога `/usr/X11R6/bin`

```
# Что находится в каталоге /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Попробуйте также "/bin", "/usr/bin", "/usr/local/bin", и т.д.

for file in $DIRECTORY/*
do
  whatis `basename $file`   # Вывод информации о файле.
Done
```

locate, **slocate**

Команда **locate** определяет местонахождение файла, используя свою базу данных, создаваемую специально для этих целей. Команда **slocate** -- это защищенная версия **locate** (которая может оказаться простым псевдонимом команды **slocate**).

```
$bash locate hickson

/usr/lib/xephem/catalogs/hickson.edb
```

readlink

Возвращает имя файла, на который указывает символическая ссылка.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

Команда **strings** используется для поиска печатаемых строк в двоичных файлах. Она выводит последовательности печатаемых символов, обнаруженных в заданном файле. Может использоваться для прикидочного анализа дамп-файлов (core dump) или для отыскания информации о типе файла, например для графических файлов неизвестного формата (например, `strings image-file | more` может вывести такую строчку: `JFIF`, что говорит о том, что мы имеем дело с графическим файлом в формате *jpeg*). В сценариях, вероятнее всего, вам придется использовать эту команду в связке с grep или sed.

Сравнение.

diff, **patch**

diff: очень гибкая утилита сравнения файлов. Она выполняет построчное сравнение файлов. В отдельных случаях, таких как поиск по словарю, может оказаться полезной фильтрация файлов с помощью sort и **uniq** перед тем как отдать поток данных через конвейер утилите **diff**. `diff file-1 file-2` -- выведет строки, имеющие от-

личия, указывая -- какому файлу, какая строка принадлежит.

С ключом `--side-by-side`, команда **diff** выведет сравниваемые файлы в две колонки, с указанием несовпадающих строк. Ключи `-c` и `-u` так же служат для облегчения интерпретации результатов работы **diff**.

Существует ряд интерфейсных оболочек для утилиты **diff**, среди них можно назвать: **spiff**, **wdiff**, **xdiff** и **mgdiff**.

Команда **diff** возвращает код завершения 0, если сравниваемые файлы идентичны и 1, если они отличаются. Это позволяет использовать **diff** в условных операторах внутри сценариев на языке командной оболочки (см. ниже).

В общем случае, **diff** используется для генерации файла различий, который используется как аргумент команды **patch**. Ключ `-e` отвечает за вывод файла различий в формате, пригодном для использования с **ed** или **ex**.

patch: гибкая утилита для "наложения заплат". С помощью файла различий, сгенерированного утилитой **diff**, утилита **patch** может использоваться для обновления устаревших версий файлов. Это позволяет распространять относительно небольшие "diff"-файлы вместо целых пакетов. Распространение "заплат" к ядру стало наиболее предпочтительным методом распространения более новых версий ядра Linux.

```
patch -p1 <patch-file
# Применит все изменения из 'patch-file'
# к файлам, описанным там же.
# Так выполняется обновление пакетов до более высоких версий.
```

diff3

Расширенная версия **diff**, которая сравнивает сразу 3 файла. В случае успеха возвращает 0, но, к сожалению, не дает никакой информации о результатах сравнения.

comm

Универсальная утилита сравнения. Работает с отсортированными файлами.

comm ***-options first-file second-file***

`comm file-1 file-2` -- вывод в три колонки:

- колонка 1 = уникальные строки для `file-1`
- колонка 2 = уникальные строки для `file-2`
- колонка 3 = одинаковые строки.

Ключи, подавляющие вывод в одной или более колонках.

- `-1` -- подавление вывода в колонку `1`
- `-2` -- подавление вывода в колонку `2`
- `-3` -- подавление вывода в колонку `3`
- `-12` -- подавление вывода в колонки `1` и `2`, и т.д.

Утилиты.

basename

Выводит только название файла, без каталога размещения. Конструкция **`basename $0`** -- позволяет сценарию узнать свое имя, то есть имя файла, который был запущен. Это имя может быть использовано для вывода сообщений, напрмиер:

```
echo "Порядок использования: `basename $0` arg1 arg2 ... argn"
```

dirname

Отсекает **basename** от полного имени файла и выводит только путь к файлу.

Утилитам **basename** и **dirname** может быть передана любая строка, в качестве аргумента. Этот аргумент необязательно должен быть именем существующего файла.

split

Утилита разбивает файл на несколько частей. Обычно используется для разбиения больших файлов, чтобы их можно было записать на дискеты или передать по электронной почте по частям.

sum, **cksum**, **md5sum**

Эти утилиты предназначены для вычисления контрольных сумм. Контрольная сумма -- это некоторое число, вычисляемое исходя из содержимого файла, и служит для контроля целостности информации в файле. Сценарий может выполнять проверку контрольных сумм для того, чтобы убедиться, что файл не был изменен или поврежден. Для большей безопасности, рекомендуется использовать 128-битную сумму, генерируемую утилитой **md5sum** (**m**essage **d**igest check**sum**).

shred

Надежное, с точки зрения безопасности, стирание файла, посредством предварительной, многократной записи в файл случайной информации, перед тем как удалить его.

Кодирование и шифрование.

uuencode

Эта утилита используется для кодирования двоичных файлов в символы ASCII, после такого кодирования файлы могут, с достаточной степенью безопасности, передаваться по сети, вкладываться в электронные письма и т.п..

uudecode

Утилита декодирования файлов, прошедших обработку утилитой uuencode.

mimencode, **mmencode**

Утилиты **mimencode** и **mmencode** предназначены для обработки закодированных мультимедийных вложений в электронные письма. Хотя *почтовые программы* (такие как **pine** или **kmail**) имеют возможность автоматической обработки таких вложений, тем не менее эти утилиты позволяют обрабатывать вложения вручную, из командной строки или в пакетном режиме, из сценария на языке командной оболочки.

mktemp

Создает временный файл с "уникальным" именем.

```
PREFIX=filename
tempfile=`mktemp $PREFIX.XXXXXX`
#                         ^^^^^^ Необходимо по меньшей мере 6 заполните-
лей
echo "имя временного файла = $tempfile"
# имя временного файла = filename.QA2ZpY
#                  или нечто подобное...
```

make

Утилита для компиляции и сборки программ. Но может использоваться для выполнения любых других операций, основанных на анализе наличия изменений в исходных файлах.

Команда **make** использует в своей работе `Makefile`, который содержит перечень зависимостей и операций, которые необходимо выполнить для удовлетворения этих зависимостей.

dos2unix

Предназначена для преобразования текстовых файлов из формата DOS (в котором строки завершаются комбинацией символов CR-LF) в формат UNIX (в котором строки завершаются одним символом LF) и обратно.

more, **less**

Команды постраничного просмотра текстовых файлов или потоков на `stdout`. Могут использоваться в сценариях в качестве фильтров.

Команды для работы с сетью.

Команды, описываемые в этом разделе, могут найти применение при исследовании и анализе процессов передачи данных по сети, а также могут использоваться в борьбе со спамерами.

Информация и статистика.

host

Возвращает информацию об узле Интернета, по заданному имени или IP адресу, выполняя поиск с помощью службы DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

Производит поиск IP адреса. С ключом `-h`, **ipcalc** выполняет поиск имени хоста в DNS, по заданному IP адресу.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Выполняет "поиск имени узла" Интернета по заданному IP адресу. По сути, эквивалентна командам **ipcalc -h** и **dig -x**. Команда может исполняться как в интерактивном, так и в неинтерактивном режиме, т.е. в пределах сценария.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
 Server:         135.116.137.2
 Address:        135.116.137.2#53

 Non-authoritative answer:
 Name:   kuhleersparnis.ch
```

traceroute

Утилита предназначена для исследования топологии сети посредством передачи ICMP пакетов удаленному узлу. Эта программа может работать в LAN, WAN и в Интернет. Удаленный узел может быть указан как по имени, так и по IP адресу. Вывод команды traceroute может быть передан по конвейеру утилитам grep или sed, для дальнейшего анализа.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
 1  tc43.xjbnnbrb.com (136.30.178.8) 191.303 ms  179.400 ms  179.767 ms
 2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
 3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
 ...
```

ping

Выполняет передачу пакета "ICMP ECHO_REQUEST" другой системе в сети. Чаще всего служит в качестве инструмента диагностики соединений, должна использоваться с большой осторожностью.

В случае успеха, **ping** возвращает код завершения 0, поэтому команда ping может использоваться в условных операторах.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of
data.
 Warning: time of day goes back, taking countermeasures.
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255
time=709 usec
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255
time=286 usec

 --- localhost.localdomain ping statistics ---
 2 packets transmitted, 2 packets received, 0% packet loss
 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois

Выполняет поиск в DNS (Domain Name System). Ключом `-h` можно указать какой из *whois* серверов будет запрошен.

finger

Возвращает информацию о пользователях в сети. По желанию, эта команда может выводить содержимое файлов `~/.plan`, `~/.project` и `~/.forward`, указанного пользователя.

```
bash$ finger
Login  Name           Tty      Idle  Login Time   Office     Office
Phone
 student   Student Bozeman   tty1        8  Jun 25 16:59
 student   Student Bozeman   ttyp0          Jun 25 16:59
 student   Student Bozeman   ttyp1          Jun 25 17:07

bash$ finger student
Login: student                          Name: Student Bozeman
 Directory: /home/student                Shell: /bin/bash
 On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
 On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
 On since Fri Aug 31 20:13 (MST) on pts/1
 On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
 No mail.
 No Plan.
```

По соображениям безопасности, в большинстве сетей служба **finger**, и соответствующий демон, отключена.

Доступ к удаленным системам.

ftp

Под этим именем подразумевается утилита и протокол передачи файлов. Сеансы ftp могут устанавливаться из сценариев.

uucp

UNIX to UNIX copy. Это коммуникационный пакет для передачи файлов между UNIX серверами. Сценарий на языке командной оболочки - один из самых эффективных способов автоматизации такого обмена.

Похоже, что с появлением Интернет и электронной почты, **uucp** постепенно уходит в небытие, однако, она с успехом может использоваться в изолированных, не имеющих выхода в Интернет, сетях.

cu

*C*all *U*p -- выполняет соединение с удаленной системой, как простой терминал. Эта команда является частью пакета **uucp** и, своего рода, упрощенным вариантом команды telnet.

telnet

Утилита и протокол для подключения к удаленной системе.

Протокол telnet небезопасен по своей природе, поэтому следует воздерживаться от его использования.

wget

wget -- *неинтерактивная* утилита для скачивания файлов с Web или ftp сайтов.

```
wget -p http://www.xyz23.com/file01.html
wget -r ftp://ftp.xyz24.net/~student/project_files/ -o $SAVEFILE
```

rlogin

Remote login -- инициирует сессию с удаленной системой. Эта команда небезопасна, вместо нее лучше использовать ssh.

rsh

Remote shell -- исполняет команду на удаленной системе. Эта команда небезопасна, вместо нее лучше использовать ssh.

rcp

Remote copy -- копирование файлов между двумя машинами через сеть. Подобно прочим r* утилитам, команда **rcp** небезопасна и потому, использовать ее в сценариях нежелательно. В качестве замены можно порекомендовать **ssh** или **expect**.

ssh

Secure shell -- устанавливает сеанс связи и выполняет команды на удаленной системе. Выступает в качестве защищенной замены для **telnet**, **rlogin**, **rcp** и **rsh**. Использует идентификацию, аутентификацию и шифрование информации, передаваемой через сеть. Подробности вы найдете в *man ssh*.

Команды управления терминалом.

Команды, имеющие отношение к консоли или терминалу.

tput

инициализация терминала или выполнение запроса к базе данных терминалов `terminfo`. С помощью **tput** можно выполнять различные операции. **tput clear** -- эк-

вивалентно команде **clear**. **tput reset** -- эквивалентно команде **reset**. **tput sgr0** -- так же сбрасывет настройки терминал, но без очистки экрана.

```
bash$ tput longname
xterm terminal emulator (XFree86 4.0 Window System)
```

Команда **tput cup X Y** перемещает курсор в координаты (X,Y). Обычно этой команде предшествует **clear**, очищающая экран.

Обратите внимание: stty предлагает более широкий диапазон возможностей.

reset

Сбрасывает настройки терминала и очищает экран. Как и в случае команды **clear**, курсор и приглашение к вводу (prompt) выводятся в верхнем левом углу терминала.

clear

Команда **clear** просто очищает экран терминала или окно xterm. Курсор и приглашение к вводу (prompt) выводятся в верхнем левом углу терминала. Эта команда может запускаться как из командной строки, так и из сценария.

script

Эта утилита позволяет сохранять в файле все символы, введенные пользователем с клавиатуры (вывод тоже). Получая, фактически, подробнейший синхронный протокол сессии.

Команды выполнения математических операций.

factor

Разложение целого числа на простые множители.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc

Bash не в состоянии выполнять действия над числами с плавающей запятой и не содержит многих важных математических функций. К счастью существует **bc**.

Универсальная, выполняющая вычисления с произвольной точностью, утилита **bc** обладает некоторыми возможностями, характерными для языков программирования.

Синтаксис **bc** немного напоминает язык C.

Поскольку это утилита UNIX, то она может достаточно широко использоваться в сценариях на языке командной оболочки, в том числе и в конвейерной обработке данных.

Ниже приводится простой шаблон работы с утилитой **bc** в сценарии. Здесь используется прием подстановки команд.

variable=$(echo "OPTIONS; OPERATIONS" | bc)

Один из вариантов вызова **bc** -- использование вложенного документа, внедряемого в блок с подстановкой команд. Это особенно актуально, когда сценарий должен передать **bc** значительный по объему список команд и аргументов.

```
variable=`bc << LIMIT_STRING
options
statements
operations
```

```
LIMIT_STRING
`
```

...ИЛИ...

```
variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

Пример. Пример взаимодействия bc со "встроенным документом".

```
#!/bin/bash
# Комбинирование 'bc' с
# 'вложенным документом'.
var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1        # 362.56

#  запись $( ... ) тоже работает.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2        # 593487.8452

var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Возвращается значение синуса от 1.7 радиана.
# Ключом "-l" вызывается математическая библиотека 'bc'.
echo $var3        # .991664810
# Попробуем функции...
hyp=              # Объявление глобальной переменной.
hypotenuse ()     # Расчет гипотенузы прямоугольного треугольника.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# К сожалению, функции Bash не могут возвращать числа с плавающей запя-
той.
}

hypotenuse 3.68 7.31
echo "гипотенуза = $hyp"    # 8.184039344
```

awk

Еще один способ выполнения математических операций, над числами с плавающей запятой, состоит в создании сценария-обертки, использующего математические функции awk.

Пример. Расчет гипотенузы прямоугольного треугольника.

```
#!/bin/bash
# hypotenuse.sh: Возвращает "гипотенузу" прямоугольного треугольника.
#                ( корень квадратный от суммы квадратов катетов)
ARGS=2                # В сценарий необходимо передать два катета.
E_BADARGS=65          # Ошибка в аргументах.
if [ $# -ne "$ARGS" ] # Проверка количества аргументов.
then
  echo "Порядок использования: `basename $0` катет_1 катет_2"
  exit $E_BADARGS
fi
AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#           команды и параметры, передаваемые в awk
echo -n "Гипотенуза прямоугольного треугольника, с катетами $1 и $2, =
"
echo $1 $2 | awk "$AWKSCRIPT"
```

Прочие команды.

dd

Эта немного непонятная и "страшная" команда ("data duplicator") изначально использовалась для переноса данных на магнитной ленте между микрокомпьютерами с ОС UNIX и майнфреймами IBM. Команда **dd** просто создает копию файла (или `stdin/stdout`), выполняя по пути некоторые преобразования. Один из вариантов: преобразование из ASCII в EBCDIC, **`dd --help`** выведет список возможных вариантов преобразований и опций этой мощной утилиты.

```
# Изучаем 'dd'.
n=3
p=5
input_file=project.txt
output_file=log.txt
dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-n+1))
2> /dev/null
# Извлечет из $input_file символы с n-го по p-й.

echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
# Выведет "hello world" вертикально.
```

Для демонстрации возможностей **dd**, попробуем перехватить нажатия на клавиши.

Команда **dd** может использоваться для создания образов дисков, считывая данные прямо с устройств, таких как дискеты, компакт диски, магнитные ленты. Обычно она используется для создания загрузочных дискет.

```
dd if=kernel-image of=/dev/fd0H1440
```

Точно так же, **dd** может скопировать все содержимое дискеты, даже с неизвестной файловой системой, на жесткий диск в виде файла-образа.

```
dd if=/dev/fd0 of=/home/student/projects/floppy.img
```

Еще одно применение **dd** -- создание временного swap-файла и ram-дисков. Она может создавать даже образы целых разделов жесткого диска, хотя и не рекомендуется делать это без особой на то необходимости.

hexdump

Выводит дамп двоичных данных из файла в восьмеричном, шестнадцатиричном,

десятичном виде или в виде ASCII. Эту команду, с массой оговорок, можно назвать эквивалентом команды of **od**.

objdump

Отображает содержимое исполняемого или объектного файла либо в шестнадцатиричной форме, либо в виде дизассемблерного листинга (с ключом `-d`).

```
bash$ objdump -d /bin/ls
/bin/ls:     file format elf32-i386

 Disassembly of section .init:

 080490bc <.init>:
  80490bc:       55                      push   %ebp
  80490bd:       89 e5                   mov    %esp,%ebp
  . . .
```

mcookie

Эта команда создает псевдослучайные шестнадцатиричные 128-битные числа, так называемые "magic cookie", обычно используется X-сервером в качестве "сигнатуры" авторизации. В сценариях может использоваться как малоэффективный генератор случайных чисел.

```
random000=`mcookie | sed -e '2p'`
# 'sed' удаляет посторонние символы.
```

Конечно, для тех же целей, сценарий может использовать md5.

Сценарий вычисляет контрольную сумму для самого себя.

```
random001=`md5sum $0 | awk '{print $1}'`
# 'awk' удаляет имя файла.
```

С помощью **mcookie** можно создавать "уникальные" имена файлов.

Команды системного администрирования.

Примеры использования большинства этих команд вы найдете в сценариях начальной загрузки и остановки системы, в каталогах `/etc/rc.d`. Они, обычно, вызываются пользователем root и используются для администрирования системы или восстановления файловой системы. Эти команды должны использоваться с большой осторожностью, так как некоторые из них могут разрушить систему, при неправильном использовании.

Пользователи и группы.

users

Выведет список всех зарегистрировавшихся пользователей. Она, до некоторой степени, является эквивалентом команды **who -q**.

groups

Выводит список групп, в состав которых входит текущий пользователь. Эта команда соответствует внутренней переменной $GROUPS, но выводит названия групп, а не их числовые идентификаторы.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
```

```
501
```

chown, **chgrp**

Команда **chown** изменяет владельца файла или файлов. Эта команда полезна в случаях, когда *root* хочет передать монопольное право на файл от одного пользователя другому. Обычный пользователь не в состоянии изменить владельца файла, за исключением своих собственных файлов.

```
root# chown student *.txt
```

Команда **chgrp** изменяет группу, которой принадлежит файл или файлы. Чтобы изменить группу, вы должны быть владельцем файла (при этом должны входить в состав указываемой группы) или привилегированным пользователем (*root*).

```
chgrp --recursive dunderheads *.data
#  Группа "dunderheads" станет владельцем всех файлов "*.data"
#+ во всех подкаталогах текущей директории ($PWD) (благодаря ключу "--
recursive").
```

useradd, **userdel**

Команда **useradd** добавляет учетную запись нового пользователя в систему и создает домашний каталог для данного пользователя. Противоположная, по смыслу, команда **userdel** удаляет учетную запись пользователя из системы, и удалит соответствующие файлы.

Команда **adduser** является синонимом для **useradd** и, как правило, является обычной символической ссылкой на **useradd**.

id

Команда **id** выводит идентификатор пользователя (реальный и эффективный) и идентификаторы групп, в состав которых входит пользователь. По сути -- выводит содержимое переменных $UID, $EUID и $GROUPS.

```
bash$ id
uid=501(student) gid=501(student)
groups=501(student),22(cdrom),80(cdwriter),81(audio)
bash$ echo $UID
501
```

who

Выводит список пользователей, работающих в настоящий момент в системе.

```
bash$ who
student  tty1     Apr 27 17:45
 student  pts/0    Apr 27 17:46
 student  pts/1    Apr 27 17:47
 student  pts/2    Apr 27 17:49
```

С ключом `-m` -- выводит информацию только о текущем пользователе. Если число аргументов, передаваемых команде, равно двум, то это эквивалентно вызову **who -m**, например **who am i** или **who The Man**.

```
bash$ who -m
localhost.localdomain!student  pts/2    Apr 27 17:49
```

whoami -- похожа на **who -m**, но выводит только имя пользователя.

```
bash$ whoami
student
```

su

Команда предназначена для запуска программы или сценария от имени другого

пользователя. **su rjones** -- запускает командную оболочку от имени пользователя *rjones*. Запуск команды **su** без параметров означает запуск командной оболочки от имени привилегированного пользователя *root*.

sudo

Исполняет заданную команду от имени пользователя root (или другого пользователя).

```
#!/bin/bash
# Доступ к "секретным" файлам.
sudo cp /root/secretfile /home/student/secret
```

Имена пользователей, которым разрешено использовать команду **sudo**, хранятся в файле `/etc/sudoers`.

Терминалы.

tty

Выводит имя терминала текущего пользователя. Обратите внимание: каждое отдельное окно xterm считается отдельным терминалом.

```
bash$ tty
/dev/pts/1
```

stty

Выводит и/или изменяет настройки терминала. Эта сложная команда используется в сценариях для управления поведением терминала.

getty, **agetty**

Программа **getty** или **agetty** запускается процессом init и обслуживает процедуру входа пользователя в систему. Эти команды не используются в сценариях.

dmesg

Выводит все сообщения, выдаваемые системой во время загрузки на `stdout`. Очень полезная утилита для отладочных целей. Вывод **dmesg** может анализироваться с помощью grep, sed или awk внутри сценария.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
 hda: IBM-DLGA-23080, ATA DISK drive
 hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
 hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

Информационные и статистические утилиты.

uname

Выводит на `stdout` имя системы. С ключом `-a`, выводит подробную информацию, содержащую имя системы, имя узла (то есть имя, под которым система известна в сети), версию операционной системы, наименование модификации операционной системы, аппаратную архитектуру.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000
i686 unknown

bash$ uname -s
Linux
```

arch

Выводит тип аппаратной платформы компьютерв. Эквивалентна команде **uname -m**.

```
bash$ arch
i686

bash$ uname -m
i686
```

du

Выводит сведения о занимаемом дисковом пространстве в каталоге и вложенных подкаталогах. Если каталог не указан, то по-умолчанию выводятся сведения о текущем каталоге.

```
bash$ du -ach
1.0k    ./wi.sh
 1.0k    ./tst.sh
 1.0k    ./random.file
 6.0k    .
 6.0k    total
```

df

Выводит в табличной форме сведения о смонтированных файловых системах.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5               273262     92607    166547  36% /
/dev/hda8               222525    123951     87085  59% /home
/dev/hda7              1408796   1075744    261488  80% /usr
```

stat

Дает подробную информацию о заданном файле (каталоге или файле устройства) или наборе файлов.

```
bash$ stat test.cru
  File: "test.cru"
   Size: 49970        Allocated Blocks: 100          Filetype: Regular
File
   Mode: (0664/-rw-rw-r--)         Uid: (  501/ student)  Gid: (  501/
student)
 Device:  3,8   Inode: 18185     Links: 1
 Access: Sat Jun  2 16:40:24 2001
 Modify: Sat Jun  2 16:40:24 2001
 Change: Sat Jun  2 16:40:24 2001
```

netstat

Показывает сведения о сетевой подсистеме, такие как: таблицы маршрутизации и активные соединения. Эта утилита получает сведения из `/proc/net`.

netstat -r -- эквивалентна команде route.

hostname

Выводит имя узла (сетевое имя системы). С помощью этой команды устанавливается сетевое имя системы в сценарии `/etc/rc.d/rc.sysinit`. Эквивалентна команде **uname -n** и внутренней переменной $HOSTNAME.

```
bash$ hostname
localhost.localdomain
```

```
bash$ echo $HOSTNAME
localhost.localdomain
```

Системный журнал.

logger

Добавляет в системный журнал (`/var/log/messages`) сообщение от пользователя. Для добавления сообщения пользователь не должен обладать привилегиями суперпользователя.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# Теперь попробуйте дать команду 'tail /var/log/messages'.
```

Встраивая вызов **logger** в сценарии, вы получаете возможность заносить отладочную информацию в системный журнал `/var/log/messages`.

```
logger -t $0 -i Logging at line "$LINENO".
# Ключ "-t" задает тэг записи в журнале.
# Ключ "-i" -- записывает ID процесса.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

Эта утилита производит манипуляции над системным журналом: ротация, сжатие, удаление и/или отправляет его по электронной почте, по мере необходимости. Как правило, утилита **logrotate** вызывается демоном crond ежедневно.

Добавляя соответствующие строки в `/etc/logrotate.conf`, можно заставить **logrotate** обрабатывать не только системный журнал, но и ваш личный.

Управление заданиями.

ps

*P*rocess *S*tatistics: Список исполняющихся в данный момент процессов. Обычно вызывается с ключами `ax`, вывод команды может быть обработан командами grep или sed, с целью поиска требуемого процесса.

```
bash$ ps ax | grep sendmail
295 ?        S      0:00 sendmail: accepting connections on port 25
```

pstree

Список исполняющихся процессов в виде "дерева". С ключом `-p` -- вместе с именами процессов отображает их PID.

top

Выводит список наиболее активных процессов. С ключом `-b` -- отображение ведется в обычном текстовом режиме, что дает возможность анализа вывода от команды внутри сценария.

nice

Запускает фоновый процесс с заданным приоритетом. Приоритеты могут задаваться числом из диапазона от 19 (низший приоритет) до -20 (высший приоритет). Но только *root* может указать значение приоритета меньше нуля (отрицательные значения). См. так же команды **renice**, **snice** и **skill**.

nohup

Запуск команд в режиме игнорирования сигналов прерывания и завершения, что

предотвращает завершение работы команды даже если пользователь, запустивший ее, вышел из системы. Если после команды не указан символ &, то она будет исполняться как процесс "переднего плана". Если вы собираетесь использовать **nohup** в сценариях, то вам потребуется использовать его в связке с командой wait, чтобы не породить процесс "зомби".

pidof

Возвращает идентификатор процесса *(pid)* по его имени. Поскольку многие команды управления процессами, такие как kill и **renice**, требуют указать *pid* процесса, а не его имя, то **pidof** может сослужить неплохую службу при идентификации процесса по его имени. Эта коменда может рассматриваться как приблизительный эквивалент внутренней переменной $PPID.

```
bash$ pidof xclock
880
```

Использование команды pidof при остановке процесса.

```
#!/bin/bash
# kill-process.sh
NOPROCESS=2
process=xxxyyyzzz  # Несуществующий процесс.
# Только в демонстрационных целях...
# ... чтобы не уничтожить этим сценарием какой-нибудь процесс.
#
# Если с помощью этого сценария вы задумаете разрыватть связь с
Internet, то
#     process=pppd

t=`pidof $process`       # Поиск pid (process id) процесса $process.
# pid требует команда 'kill' (невозможно остановить процесс, указав его
имя).

if [ -z "$t" ]           # Если процесс с таким именем не найден, то
'pidof' вернет null.
then
  echo "Процесс $process не найден."
  exit $NOPROCESS
fi

kill $t                  # В некоторых случаях может потребоваться
'kill -9'.

# Здесь нужно проверить -- был ли уничтожен процесс.
# Возможно так: " t=`pidof $process` ".
# Этот сценарий мог бы быть заменен командой
#    kill $(pidof -x process_name)
# но это было бы не так поучительно.

exit 0
```

Команды управления процессами и загрузкой.

init

init -- предок (родитель) всех процессов в системе. Вызывается на последнем этапе загрузки системы и определяет уровень загрузки (runlevel) из файла `/etc/inittab`.

telinit

Символическая ссылка на **init** - инструмент для смены уровня загрузки (runlevel),

как правило используется при обслуживании системы или восстановлении файловой системы. Может быть вызвана только суперпользователем. Эта команда может быть очень опасна, при неумелом обращении - прежде чем использовать ее, убедитесь в том, что вы совершенно точно понимаете что делаете!

runlevel

Выводит предыдущий и текущий уровни загрузки (runlevel). Уровень загрузки может иметь одно из 6 значений: `0` -- остановка системы, `1` -- однопользовательский режим, `2` или `3` -- многопользовательский режим, `5` -- многопользовательский режим и запуск X Window, `6` -- перезагрузка. Уровни загрузки определяются из файла `/var/run/utmp`.

halt, **shutdown**, **reboot**

Набор команд для остановки системы, обычно перед выключением питания.

Команды для работы с сетью.

ifconfig

Утилита конфигурирования и запуска сетевых интерфейсов. Чаще всего используется в сценариях начальной загрузки системы, для настройки и запуска сетевых интерфейсов или для их остановки перед остановкой или перезагрузкой.

route

Выводит сведения о таблице маршрутизации ядра или вносит туда изменения.

tcpdump

"Сниффер" ("sniffer") сетевых пакетов. Инструмент для перехвата и анализа сетевого трафика по определенным критериям.

Дамп трафика ip-пакетов между двумя узлами сети -- *studentville* и *caduceus*:

`bash$` **tcpdump ip host studentville and caduceus**

Конечно же, вывод команды **tcpdump** может быть проанализирован с помощью команд обработки текста, обсуждавшихся выше.

Команды для работы с файловыми системами.

mount

Выполняет монтирование файловой системы, обычно на устройстве со сменными носителями, такими как дискеты или CDROM. Файл `/etc/fstab` содержит перечень доступных для монтирования файловых систем, разделов и устройств, включая опции монтирования, благодаря этому файлу, монтирование может производиться автоматически или вручеую. Файл `/etc/mtab` содержит список смонтированных файловых систем и разделов (включая виртуальные, такие как `/proc`).

mount -a -- монтирует все (all) файловые системы и разделы, перечисленные в `/etc/fstab`, за исключением тех, которые имеют флаг `noauto`. Эту команду можно встретить в сценариях начальной загрузки системы из `/etc/rc.d` (`rc.sysinit` или нечто похожее).

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Монтирование CDROM-а
mount /mnt/cdrom
# Более короткий и удобный вариант, если точка монтирования /mnt/cdrom
описана в /etc/fstab
```

Эта команда может даже смонтировать обычный файл как блочное устройство. Достигается это за счет связывания файла с loopback-устройством. Эту возможность можно использовать для проверки ISO9660 образа компакт-диска перед его

записью на болванку.

Пример. Проверка образа CD.

```
# С правами root...
mkdir /mnt/cdtest  # Подготовка точки монтирования.
mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Монтирование
образа диска.
#             ключ "-o loop" эквивалентен "losetup /dev/loop0"
cd /mnt/cdtest     # Теперь проверим образ диска.
ls -alR            # Вывод списка файлов
```

umount

Отмонтирует смонтированную файловую систему. Перед тем как физически вынуть компакт-диск или дискету из устройства, это устройство должно быть отмонтировано командой **umount**, иначе файловая система может оказаться поврежденной (особенно это относится к накопителям на гибких магнитных дисках, прим. перев.).

```
umount /mnt/cdrom
# Теперь вы можете извлечь диск из привода.
```

sync

Принудительный сброс содержимого буферов на жесткий диск (синхронизация содержимого буферов ввода-вывода и устройства-носителя). Несмотря на то, что нет такой уж острой необходимости в этой утилите, тем не менее **sync** придает уверенности системным администраторам или пользователям в том, что их данные будут сохранены на жестком диске, и не будут потеряны в случае какого-либо сбоя. В былые дни, команда **`sync; sync`** (дважды - для абсолютной уверенности) была упреждающей мерой перед перезагрузкой системы.

Иногда возникает необходимость принудительной синхронизации буферов ввода-вывода с содержимым на магнитном носителе, как, например, при надежном удалении файла или когда наблюдаются скачки напряжения в сети электроснабжения.

chroot

CHange ROOT -- смена корневого каталога. Обычно, команды и утилиты ориентируются в файловой системе посредством переменной $PATH, относительно корневого каталога `/`. Команда **chroot** изменяет корневой каталог по-умолчанию на другой (рабочий каталог также изменяется). Эта утилита, как правило, используется с целью защиты системы, например, с ее помощью можно ограничить доступ к разделам файловой системы для пользователей, подключающихся к системе с помощью telnet (это называется - "поместить пользователя в chroot окружение"). Обратите внимание: после выполнения команды **chroot** изменяется путь к исполняемым файлам системы.

Команда **`chroot /opt`** приведет к тому, что все обращения к каталогу `/usr/bin` будут переводиться на каталог `/opt/usr/bin`. Аналогично, **`chroot /aaa/bbb /bin/ls`** будет пытаться вызвать команду **ls** из каталога `/aaa/bbb/bin`, при этом, корневым каталогом для ls станет каталог /aaa/bbb. Поместив строчку **alias XX 'chroot /aaa/bbb ls'** в пользовательский `~/.bashrc`, можно эффективно ограничить доступ команде "XX", запускаемой пользователем, к разделам файловой системы.

Команды резервного копирования.

dump, **restore**

Команда **dump** создает резервные копии целых файловых систем, обычно используется в крупных системах и сетях. Она считывает дисковые разделы и сохраняет их в файле, в двоичном формате. Созданные таким образом файлы, могут быть со-

хранены на каком-либо носителе - на жестком диске или магнитной ленте. Команда **restore** -- "разворачивает" файлы, созданные утилитой **dump**.

fdformat

Выполняет низкоуровневое форматирование дискет.

Команды для работы с модулями ядра.

lsmod

Выводит список загруженных модулей.

```
bash$ lsmod
Module                  Size  Used by
 autofs                  9456   2 (autoclean)
 opl3                   11376   0
 serial_cs               5456   0 (unused)
 sb                     34752   0
 uart401                 6384   0 [sb]
 sound                  58368   0 [opl3 sb uart401]
 soundlow                 464   0 [sound]
 soundcore               2800   6 [sb sound]
 ds                      6448   2 [serial_cs]
 i82365                 22928   2
 pcmcia_core            45984   0 [serial_cs ds i82365]
```

insmod

Принудительная загрузка модуля ядра (старайтесь вместо **insmod** использовать команду **modprobe**). Должна вызываться с привилегиями пользователя root.

rmmod

Выгружает модуль ядра. Должна вызываться с привилегиями пользователя root.

modprobe

Загрузчик модулей, который обычно вызывается из сценариев начальной загрузки системы. Должна вызываться с привилегиями пользователя root.

depmod

Создает файл зависимостей между модулями, обычно вызывается из сценариев начальной загрузки системы.

Прочие команды.

env

Запускает указанную программу или сценарий с модифицированными переменными окружения (не изменяя среду системы в целом, изменения касаются только окружения запускаемой программы/сценария). Посредством `[varname=xxx]`, устанавливает значение переменной окружения `varname`, которая будет доступна из запускаемой программы/сценария. Без параметров -- просто выводит список переменных окружения с их значениями.

watch

Периодически запускает указанную программу с заданным интервалом времени.

По-умолчанию интервал между запусками принимается равным 2 секундам, но может быть изменен ключом `-n`.

```
watch -n 5 tail /var/log/messages
# Выводит последние 10 строк из системного журнала, /var/log/messages,
каждые пять секунд.
```

Регулярные выражения.

Для того, чтобы полностью реализовать потенциал командной оболочки, вам придется овладеть Регулярными Выражениями. Многие команды и утилиты, обычно используемые в сценариях, такие как grep, expr, sed и awk, используют Регулярные Выражения.

Краткое введение в регулярные выражения.

Выражение - это строка символов. Символы, которые имеют особое назначение, называются *метасимволами*. Так, например, кавычки могут выделять прямую речь, т.е. быть *метасимволами* для строки, заключенной в эти кавычки. Регулярные выражения - это набор символов и/или метасимволов, которые наделены особыми свойствами.

Основное назначение регулярных выражений - это поиск текста по шаблону и работа со строками.

- Звездочка **-- * --** означает любое количество символов в строке, предшествующих "звездочке", *в том числе и нулевое число символов*.

  Выражение "1133*" -- означает *11 + один или более символов "3" + любые другие символы*: *113*, *1133*, *113312*, и так далее.

- Точка **-- . --** означает не менее одного любого символа, за исключением символа перевода строки (\n).

  Выражение "13." будет означать *13 + по меньшей мере один любой символ (включая пробел)*: *1133*, *11333*, но не *13* (отсутствуют дополнительные символы).

- Символ **-- ^ --** означает начало строки, но иногда, в зависимости от контекста, означает отрицание в регулярных выражениях.

- Знак доллара **-- $ --** в конце регулярного выражения соответствует концу строки.

  Выражение **"^$"** соответствует пустой строке.

  Символы ^ и $ иногда еще называют якорями*, поскольку они означают, или закрепляют, позицию в регулярных выражениях.*

- Квадратные скобки **-- [...] --** предназначены для задания подмножества символов. Квадратные скобки, внутри регулярного выражения, считаются одним символом, который может принимать значения, перечисленные внутри этих скобок..

  Выражение **"[xyz]"** -- соответствует одному из символов *x*, *y* или *z*.

  Выражение **"[c-n]"** соответствует одному из символов в диапазоне от *c* до *n*, включительно.

  Выражение **"[B-Pk-y]"** соответствует одному из символов в диапазоне от *B* до *P* или в диапазоне от *k* до *y*, включительно.

  Выражение **"[a-z0-9]"** соответствует одному из символов латиницы в нижнем регистре или цифре.

  Выражение **"[^b-d]"** соответствует любому символу, кроме символов из диапазона от *b* до *d*, включительно. В данном случае, метасимвол ^ означает отрицание.

  Объединяя квадратные скобки в одну последовательность, можно задать шаблон искомого слова. Так, выражение **"[Yy][Ee][Ss]"** соответствует словам *yes*, *Yes*,

YES, *yEs* и так далее.

- Обратный слэш -- \ -- служит для экранирования специальных символов, это означает, что экранированные символы должны интерпретироваться буквально, т.е. как простые символы.

  Комбинация **"\$"** указывает на то, что символ "$" трактуется как обычный символ, а не как признак конца строки в регулярных выражениях. Аналогично, комбинация "\\" соответствует простому символу "\".

- Экранированные "угловые скобки" -- **\<...\>** -- отмечают границы слова.

  Угловые скобки должны экранироваться, иначе они будут интерпретироваться как простые символы.

  Выражение **"\<the\>"** соответствует слову "the", и не соответствует словам "them", "there", "other" и т.п.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.
This is line 4.

bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

- **Дополнительные метасимволы.** Использующиеся при работе с egrep, awk и Perl
- Знак вопроса -- **?** -- означает, что предыдущий символ или регулярное выражение встречается 0 или 1 раз. В основном используется для поиска одиночных символов.
- Знак "плюс" -- **+** -- указывает на то, что предыдущий символ или выражение встречается 1 или более раз. Играет ту же роль, что и символ "звездочка" (*), за исключением случая нулевого количества вхождений.

```
# GNU версии sed и awk допускают использование "+",
# но его необходимо экранировать.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# Все три варианта эквивалентны.
```

- Экранированные "фигурные скобки**"** -- **\{ \}** -- задают число вхождений предыдущего выражения.

  Экранирование фигурных скобок -- обязательное условие, иначе они будут интерпретироваться как простые символы. Такой порядок использования, технически, не является частью основного набора правил построения регулярных выражений.

  Выражение "[0-9]\{5\}" -- в точности соответствует подстроке из пяти десятичных

цифр (символов из диапазона от 0 до 9, включительно).

- Круглые скобки -- **( )** -- предназначены для выделения групп регулярных выражений. Они полезны при использовании с оператором "|" и при извлечении подстроки с помощью команды expr.
- Вертикальная черта -- **|** -- выполняет роль логического оператора "ИЛИ" в регулярных выражениях и служит для задания набора альтернатив.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
The clarinet produces sound by the vibration of its reed.
```

- **Классы символов POSIX. `[:class:]`**

  Это альтернативный способ указания диапазона символов.
- Класс **`[:alnum:]`** -- соответствует алфавитным символам и цифрам. Эквивалентно выражению **`[A-Za-z0-9]`**.
- Класс **`[:alpha:]`** -- соответствует символам алфавита. Эквивалентно выражению **`[A-Za-z]`**.
- Класс **`[:blank:]`** -- соответствует символу пробела или символу табуляции.
- Класс **`[:cntrl:]`** -- соответствует управляющим символам (control characters).
- Класс **`[:digit:]`** -- соответствует набору десятичных цифр. Эквивалентно выражению **`[0-9]`**.
- Класс **`[:graph:]`** (печатаемые и псевдографические символы) -- соответствует набору символов из диапазона ASCII 33 - 126. Это то же самое, что и класс **`[:print:]`**, за исключением символа пробела.
- Класс **`[:lower:]`** -- соответствует набору алфавитных символов в нижнем регистре. Эквивалентно выражению **`[a-z]`**.
- Класс **`[:print:]`** (печатаемые символы) -- соответствует набору символов из диапазона ASCII 32 - 126. По своему составу этот класс идентичен классу **`[:graph:]`**, описанному выше, за исключением того, что в этом классе дополнительно присутствует символ пробела.
- Класс **`[:space:]`** -- соответствует пробельным символам (пробел и горизонтальная табуляция).
- Класс **`[:upper:]`** -- соответствует набору символов алфавита в верхнем регистре. Эквивалентно выражению **`[A-Z]`**.
- Класс **`[:xdigit:]`** -- соответствует набору шестнадцатиричных цифр. Эквивалентно выражению **`[0-9A-Fa-f]`**.

  Вообще, символьные классы POSIX требуют заключения в кавычки или двойные квадратные скобки ([[ ]]).

  ```
  bash$ grep [[:digit:]] test.file
  abc=723
  ```

  Эти символьные классы могут использоваться, с некоторыми ограничениями, даже в операциях подстановки имен файлов (globbing).

  ```
  bash$ ls -l ?[[:digit:]][[:digit:]]?
  -rw-rw-r--    1 student  student         0 Aug 21 14:47 a33b
  ```

Globbing -- Подстановка имен файлов.

Bash, сам по себе, не распознает регулярные выражения. Но в сценариях можно использовать команды и утилиты, такие как sed и awk, которые прекрасно справляются с обработкой регулярных выражений.

Фактически, Bash может выполнять подстановку имен файлов, этот процесс называется "globbing", но при этом *не* используется стандартный набор регулярных выражений. Вместо этого, при выполнении подстановки имен файлов, производится распознавание и интерпретация шаблонных символов. В число интерпретируемых шаблонов входят символы * и ?, списки символов в квадратных скобках и некоторые специальные символы (например ^, используемый для выполнения операции отрицания). Применение шаблонных символов имеет ряд важных ограничений. Например, если имена файлов начинаются с точки (например так: `.bashrc`), то они не будут соответствовать шаблону, содержащему символ `*`. Аналогично, символ `?` в операции подстановки имен файлов имеет иной смысл, нежели в регулярных выражениях.

```
bash$ ls -l
total 2
 -rw-rw-r--    1 student  student         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 student  student         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 student  student         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 student  student       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 student  student       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 student  student       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 student  student         0 Aug  6 18:42 a.1
-rw-rw-r--    1 student  student         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 student  student         0 Aug  6 18:42 a.1
-rw-rw-r--    1 student  student         0 Aug  6 18:42 b.1
-rw-rw-r--    1 student  student         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 student  student         0 Aug  6 18:42 c.1
-rw-rw-r--    1 student  student       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 student  student       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 student  student         0 Aug  6 18:42 b.1
-rw-rw-r--    1 student  student         0 Aug  6 18:42 c.1
-rw-rw-r--    1 student  student       758 Jul 30 09:02 test1.txt


bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```

Подоболочки, или Subshells.

Запуск сценария приводит к запуску дочернего командного интерпретатора. Кото-

рый выполняет интерпретацию и исполнение списка команд, содержащихся в файле сценария, точно так же, как если бы они были введены из командной строки. Любой сценарий запускается как дочерний процесс родительской командной оболочки, той самой, которая выводит перед вами строку приглашения к вводу на консоли или в окне xterm.

Сценарий может, так же, запустить другой дочерний процесс, в своей подоболочке. Это позволяет сценариям распараллелить процесс обработки данных по нескольким задачам, исполняемым одновременно.

Список команд в круглых скобках

(command1; command2; command3; ...)

> Список команд, в круглых скобках, исполняется в подоболочке.
>
> *Значения переменных, определенных в дочерней оболочке,* не *могут быть переданы родительской оболочке. Они недоступны родительскому процессу. Фактически, они ведут себя как локальные переменные*

Пример. Область видимости переменных.

```
outer_variable=Outer
 (
inner_variable=Inner
echo "Дочерний процесс, \"inner_variable\" = $inner_variable"
echo "Дочерний процесс, \"outer\" = $outer_variable"
)
if [ -z "$inner_variable" ]
then
  echo "Переменная inner_variable не определена в родительской оболочке"
else
  echo "Переменная inner_variable определена в родительской оболочке"
fi

echo "Родительский процесс, \"inner_variable\" = $inner_variable"
# Переменная $inner_variable не будет определена
# потому, что переменные, определенные в дочернем процессе,
# ведут себя как "локальные переменные".
```

Смена текущего каталога в дочернем процессе (подоболочке) не влечет за собой смену текущего каталога в родительской оболочке.

Пример. Личные настройки пользователей.

```
# allprofs.sh: вывод личных настроек (profiles) всех пользователей

FILE=.bashrc  #  Файл настроек пользователя,
              #+ в оригинальном сценарии называется ".profile".

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    # Перейти к следующей итерации, если нет до-
машнего каталога.
  [ -r "$home" ] || continue    # Перейти к следующей итерации, если не дос-
тупен для чтения.
  (cd $home; [ -e $FILE ] && less $FILE)
done

#  По завершении сценария -- нет необходимости выполнять команду 'cd', чтобы
вернуться в первоначальный каталог,
#+ поскольку 'cd $home' выполняется в подоболочке.

exit 0
```

Подоболочка может использоваться для задания "специфического окружения" для группы команд.

```
COMMAND1
COMMAND2
COMMAND3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMMAND4
  COMMAND5
  exit 3 # Выход только из подоболочки.
)
# Изменение переменных окружения не коснется родительской оболочки.
COMMAND6
COMMAND7
```

Как вариант использования подоболочки - проверка переменных.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "Переменная определена."
fi

# Можно сделать то же самое по другому: [[ ${variable-x} != x || ${variable-
y} != y ]]
# или                                   [[ ${variable-x} != x$variable ]]
# или                                   [[ ${variable+x} = x ]])
```

Еще одно применение - проверка файлов блокировки:

```
if (set -C; : > lock_file) 2> /dev/null
then
  echo "Этот сценарий уже запущен другим пользователем."
  exit 65
fi
```

Процессы в подоболочках могут исполняться параллельно. Это позволяет разбить сложную задачу на несколько простых подзадач, выполняющих параллельную обработку информации.

Пример. Запуск нескольких процессов в подоболочках.

```
(cat list1 list2 list3 | sort | uniq > list123) &
 (cat list4 list5 list6 | sort | uniq > list456) &
 # Слияние и сортировка двух списков производится одновременно.
 # Запуск в фоне гарантирует параллельное исполнение.
 #
 # Тот же эффект дает
 #   cat list1 list2 list3 | sort | uniq > list123 &
 #   cat list4 list5 list6 | sort | uniq > list456 &

 wait   # Ожидание завершения работы подоболочек.

 diff list123 list456
```

Перенаправление ввода/вывода в/из подоболочки производится оператором построения конвейера "|", например, **`ls -al | (command)`**.

Функции.

Подобно "настоящим" языкам программирования, Bash тоже имеет функции, хотя и в несколько ограниченном варианте. Функция -- это подпрограмма, блок кода который реализует набор операций, своего рода "черный ящик", предназначенный для выполнения конкретной задачи. Функции могут использоваться везде, где имеются участки повторяющегося кода.

function *function_name* {
command...
}

или

function_name () {
command...
}

Вторая форма записи ближе к сердцу C-программистам (она же более переносимая).

Как и в языке C, скобка, открывающая тело функции, может помещаться на следующей строке.

function_name ()
{
command...
}

Вызов функции осуществляется простым указанием ее имени в тексте сценария.

Пример. Простая функция.

```
funky ()
{
  echo "Это обычная функция."
} # Функция должна быть объявлена раньше, чем ее можно будет использовать.
  # Вызов функции.
funky
```

Функция должна быть объявлена раньше, чем ее можно будет использовать. К сожалению, в Bash нет возможности "опережающего объявления" функции, как например в C.

Допускается даже создание вложенных функций, хотя пользы от этого немного.

Сложные функции и сложности с функциями.

Функции могут принимать входные аргументы и возвращать код завершения.

```
function_name $arg1 $arg2
```

Доступ к входным аргументам, в функциях, производится посредством позиционных параметров, т.е. `$1`, `$2` и так далее.

Пример. Функция с аргументами.

```
DEFAULT=default
# Значение аргумента по-умолчанию.

func2 () {
   if [ -z "$1" ]
# Длина аргумента #1 равна нулю?
```

```
   then
     echo "-Аргумент #1 имеет нулевую длину.-"
# Или аргумент не был передан функции.
   else
     echo "-Аргумент #1: \"$1\".-"
   fi

   variable=${1-$DEFAULT}                    # Что делает
   echo "variable = $variable"               # показанная подстановка
                                             # параметра?
                                             #  --------------------------
                                             # Она различает отсутствующий
                                             # аргумент
                                             # от "пустого" аргумента.

   if [ "$2" ]
   then
     echo "-Аргумент #2: \"$2\".-"
   fi

   return 0
}

echo

echo "Вызов функции без аргументов."
func2
echo

echo "Вызов функции с \"пустым\" аргументом."
func2 ""
echo

echo "Вызов функции с неинициализированным аргументом."
func2 "$uninitialized_param"
echo

echo "Вызов функции с одним аргументом."
func2 first
echo

echo "Вызов функции с двумя аргументами."
func2 first second
echo

echo "Вызов функции с аргументами \"\" \"second\"."
func2 "" second       # Первый параметр "пустой"
echo                  # и второй параметр -- ASCII-строка.
```

Exit и Return.

код завершения

Функции возвращают значение в виде *кода завершения*. Код завершения может быть задан явно, с помощью команды **return**, в противном случае будет возвращен код завершения последней команды в функции (0 -- в случае успеха, иначе -- ненулевой код ошибки). Код завершения в сценарии может быть получен через переменную $?.

return

Завершает исполнение функции. Команда **return** может иметь необязательный аргумент типа *integer*, который возвращается в вызывающий сценарий как "код за-

вершения" функции, это значение так же записывается в переменную $?.

Пример. Наибольшее из двух чисел.

```
E_PARAM_ERR=-198     # Если функции передано меньше двух параметров.
EQUAL=-199           # Возвращаемое значение, если числа равны.

max2 ()              # Возвращает наибольшее из двух чисел.
{                    # Внимание: сравниваемые числа должны быть меньше
257.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "Функции должно быть передано два аргумента."
elif [ "$return_val" -eq $EQUAL ]
  then
    echo "Числа равны."
else
    echo "Наибольшее из двух чисел: $return_val."
fi
```

Пример. Преобразование чисел в римскую форму записи.

```
# Преобразование чисел из арабской формы записи в римскую
# Диапазон: 0 - 200
# Расширение диапазона представляемых чисел и улучшение сценария
# оставляю вам, в качестве упражнения.

# Порядок использования: roman number-to-convert

LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` number-to-convert"
  exit $E_ARG_ERR
fi

num=$1
if [ "$num" -gt $LIMIT ]
then
```

```
  echo "Выход за границы диапазона!"
  exit $E_OUT_OF_RANGE
fi

to_roman ()   # Функция должна быть объявлена до того как она будет вы-
звана.
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ "$remainder" -ge 0 ]
do
  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number
}

to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
num=$?
to_roman $num 1 I
```

Перенаправление.

Перенаправление ввода для функций

Функции - суть есть блок кода, а это означает, что устройство `stdin` для функций может быть переопределено (перенаправление stdin).

Пример. Настоящее имя пользователя.

```
# По имени пользователя получить его "настоящее имя" из /etc/passwd.
ARGCOUNT=1  # Ожидается один аргумент.
E_WRONGARGS=65

file=/etc/passwd
pattern=$1

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Порядок использования: `basename $0` USERNAME"
  exit $E_WRONGARGS
fi
file_excerpt ()  # Производит поиск в файле по заданному шаблону, выво-
дит требуемую часть строки.
{
while read line
```

```
do
  echo "$line" | grep $1 | awk -F":" '{ print $5 }'  # Указывет awk ис-
пользовать ":" как разделитель полей.
done
} <$file  # Подменить stdin для функции.

file_excerpt $pattern

# Да, этот сценарий можно уменьшить до
#       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
# или
#       awk -F: '/PATTERN/ {print $5}'
# или
#       awk -F: '($1 == "username") { print $5 }'
```

Ниже приводится альтернативный, и возможно менее запутанный, способ перенаправления ввода для функций. Он заключается в использовании перенаправления ввода для блока кода, заключенного в фигурные скобки, в пределах функции.

```
# Вместо:
Function ()
{
 ...
 } < file

# Попробуйте так:
Function ()
{
  {
    ...
   } < file
}

# Похожий вариант,

Function ()  # Тоже работает.
{
  {
   echo $*
  } | tr a b
}

Function ()  # Этот вариант не работает.
{
  echo $*
} | tr a b   # Наличие вложенного блока кода -- обязательное условие.
```

Локальные переменные.

Что такое "локальная" переменная?

локальные переменные

Переменные, объявленные как *локальные*, имеют ограниченную область видимости, и доступны только в пределах блока, в котором они были объявлены. Для функций это означает, что локальная переменная "видна" только в теле самой функции.

Пример. Область видимости локальных переменных.

```
func ()
```

```
{
  local loc_var=23       # Объявление локальной переменной.
  echo
  echo "\"loc_var\" в функции = $loc_var"
  global_var=999         # Эта переменная не была объявлена локальной.
  echo "\"global_var\" в функции = $global_var"
}

func

# Проверим, "видна" ли локальная переменная за пределами функции.

echo
echo "\"loc_var\" за пределами функции = $loc_var"
                                       # "loc_var" за пределами функции
=
                                       # Итак, $loc_var не видна в гло-
бальном контексте.
echo "\"global_var\" за пределами функции = $global_var"
                                       # "global_var" за пределами функ-
ции = 999
                                       # $global_var имеет глобальную
область видимости.
```

Переменные, объявляемые в теле функции, считаются необъявленными до тех пор, пока функция не будет вызвана. Это касается *всех* переменных.

```
#!/bin/bash

func ()
{
global_var=37    #  Эта переменная будет считаться необъявленной
                 #+ до тех пор, пока функция не будет вызвана.
}                # КОНЕЦ ФУНКЦИИ

echo "global_var = $global_var"  # global_var =
                                 #  Функция "func" еще не была вызвана,
                                 #+ поэтому $global_var пока еще не
"видна" здесь.

func
echo "global_var = $global_var"  # global_var = 37
                                 # Переменная была инициализирована в
функции.
```

Списки команд.

Средством обработки последовательности из нескольких команд служат списки: "И-списки" и "ИЛИ-списки". Они эффективно могут заменить сложную последовательность вложенных **if/then** или даже **case**.

Объединение команд в цепочки.

И-список

```
command-1 && command-2 && command-3 && ... command-n
```

Каждая последующая команда, в таком списке, выполняется только тогда, когда предыдущая команда вернула код завершения true (ноль). Если какая-либо из команд возвращает false (не ноль), то исполнение списка команд в этом месте завершается, т.е. следующие далее команды не выполняются.

Пример. Проверка аргументов командной строки с помощью "И-списка".

```
# "И-список"
if [ ! -z "$1" ] && echo "Аргумент #1 = $1" && [ ! -z "$2" ] && echo
"Аргумент #2 = $2"
then
  echo "Сценарию передано не менее 2 аргументов."
  # Все команды в цепочке возвращают true.
else
  echo "Сценарию передано менее 2 аргументов."
  # Одна из команд в списке вернула false.
fi
# Обратите внимание: "if [ ! -z $1 ]" тоже работает, но, казалось бы
эквивалентный вариант
#  if [ -n $1 ] -- нет. Однако, если добавить кавычки
#  if [ -n "$1" ] то все работает.  Будьте внимательны!
# Проверяемые переменные лучше всегда заключать в кавычки.


# То же самое, только без списка команд.
if [ ! -z "$1" ]
then
  echo "Аргумент #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Аргумент #2 = $2"
  echo "Сценарию передано не менее 2 аргументов."
else
  echo "Сценарию передано менее 2 аргументов."
fi
# Получилось менее элегантно и длиннее, чем с использованием "И-
списка".
```

ИЛИ-список

```
command-1 || command-2 || command-3 || ... command-n
```

Каждая последующая команда, в таком списке, выполняется только тогда, когда предыдущая команда вернула код завершения false (не ноль). Если какая-либо из команд возвращает true (ноль), то исполнение списка команд в этом месте завершается, т.е. следующие далее команды не выполняются. Очевидно, что "ИЛИ-списки" имеют смысл обратный, по отношению к "И-спискам"

Пример. Комбинирование "ИЛИ-списков" и "И-списков".

```
#!/bin/bash

#  delete.sh, утилита удаления файлов.
#  Порядок использования: delete имя_файла

E_BADARGS=65

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` имя_файла"
  exit $E_BADARGS  # Если не задано имя файла.
else
  file=$1          # Запомнить имя файла.
fi

[ ! -f "$file" ] && echo "Файл \"$file\" не найден. \
```

```
      Робкий отказ удаления несуществующего файла."
      # И-СПИСОК, выдать сообщение об ошибке, если файл не существует.
      # Обратите внимание: выводимое сообщение продолжается во второй строке,
      # благодаря экранированию символа перевода строки.

      [ ! -f "$file" ] || (rm -f $file; echo "Файл \"$file\" удален.")
      # ИЛИ-СПИСОК, удаляет существующий файл.
      # Обратите внимание на логические условия.
      # И-СПИСОК отрабатывает по true, ИЛИ-СПИСОК -- по false.
```

Комбинируя "И" и "ИЛИ" списки, легко "перемудрить" с логическими условиями, поэтому, в таких случаях может потребоваться детальная отладка.

```
false && true || echo false         # false
# Тот же результат дает
( false && true ) || echo false     # false
# Но не эта комбинация
false && ( true || echo false )     # (нет вывода на экран)

#  Обратите внимание на группировку и порядок вычисления условий -
#  слева - направо,
# поскольку логические операции "&&" и "||" имеют равный приоритет.
#  Если вы не уверены в своих действиях, то лучше избегать таких сложных конструкций.
```

Массивы.

Bash поддерживает одномерные массивы. Инициализация элементов массива может быть произведена в виде: **variable[xx]**. Можно явно объявить массив в сценарии, с помощью директивы declare: **declare -a variable**. Обращаться к отдельным элементам массива можно с помощью *фигурных скобок*, т.е.: **${variable[xx]}**.

Пример. Простой массив.

```
area[11]=23
area[13]=37
area[51]=UFOs

# Массивы не требуют, чтобы последовательность элементов
# в массиве была непрерывной.
# Некоторые элементы массива могут оставаться неинициализированными.
# "Дырки" в массиве не являются ошибкой.

echo -n "area[11] = "
echo ${area[11]}    #  необходимы {фигурные скобки}
echo -n "area[13] = "
echo ${area[13]}

echo "содержимое area[51] = ${area[51]}."

# Обращение к неинициализированным элементам дает пустую строку.
echo -n "area[43] = "
echo ${area[43]}
echo "(элемент area[43] -- неинициализирован)"

# Сумма двух элементов массива, записанная в третий элемент
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
```

```
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# Эта попытка закончится неудачей, поскольку сложение целого
# числа со строкой не допускается.
# ---------------------------------------------------------------
# Другой массив, "area2".
# И другой способ инициализации массива...
# array_name=( XXX YYY ZZZ ... )

area2=( ноль один два три четыре )

echo -n "area2[0] = "
echo ${area2[0]}
# индексация начинается с нуля (первый элемент массива имеет индекс [0]).

echo -n "area2[1] = "
echo ${area2[1]}    # [1] -- второй элемент массива.
# ---------------------------------------------
# Еще один массив, "area3".
# И еще один способ инициализации...
# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=семнадцать [21]=двадцать_один)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[21] = "
echo ${area3[21]}
```

Bash позволяет оперировать переменными, как массивами, даже если они не были явно объявлены таковыми.

```
string=abcABC123ABCabc
echo ${string[@]}               # abcABC123ABCabc
echo ${string[*]}               # abcABC123ABCabc
echo ${string[0]}               # abcABC123ABCabc
echo ${string[1]}               # Ничего не выводится!
                                # Почему?
echo ${#string[@]}              # 1
                                # Количество элементов в массиве.
```

Как видно из предыдущего примера, обращение к **${array_name[@]}** или **${array_name[*]}** относится ко *всем* элементам массива. Чтобы получить количество элементов массива, можно обратиться к **${#array_name[@]}** или к **${#array_name[*]}**. **${#array_name}** -- это длина (количество символов) первого элемента массива, т.е. **${array_name[0]}**.

/dev и /proc.

Как правило, Linux или UNIX система имеет два каталога специального назначения: `/dev` и `/proc`.

/dev

Каталог `/dev` содержит файлы физических *устройств*, которые могут входить в состав аппаратного обеспечения компьютера. Каждому из разделов не жестком диске соответст-

вует свой файл-устройство в каталоге /dev, информация о которых может быть получена простой командой df.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6               495876    222748    247527  48% /
 /dev/hda1                50755      3887     44248   9% /boot
 /dev/hda8               367013     13262    334803   4% /home
 /dev/hda5              1714416   1123624    503704  70% /usr
```

Кроме того, каталог /dev содержит *loopback*-устройства ("петлевые" устройства), например /dev/loop0. С помощью такого устройства можно представить обычный файл как блочное устройство ввода/вывода. Это позволяет монтировать целые файловые системы, находящиеся в отдельных больших файлах.

Отдельные псевдоустройства в /dev имеют особое назначение, к таким устройствам можно отнести /dev/null, /dev/zero и /dev/urandom.

./proc

Фактически, каталог /proc - это виртуальная файловая система. Файлы, в каталоге /proc, содержат информацию о процессах, о состоянии и конфигурации ядра и системы.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md
```

```
bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
```

```
 ERR:          0
```

```
bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse
running use aveq

    3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0
111550 644030
    3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
    3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
    3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
    ...
```

```
bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119
```

Сценарии командной оболочки могут извлекать необходимую информацию из соответствующих файлов в каталоге `/proc`.

```
bash$ cat /proc/filesystems | grep iso9660
        iso9660
kernel_version=$( awk '{ print $3 }' /proc/version )

CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )

if [ $CPU = Pentium ]
then
  выполнить_ряд_специфичных_команд
  ...
else
  выполнить_ряд_других_специфичных_команд
  ...
fi
```

В каталоге `/proc` вы наверняка заметите большое количество подкаталогов, с не совсем обычными именами, состоящими только из цифр. Каждый из них соответствует исполняющемуся процессу, а имя каталога - это ID (идентификатор) процесса. Внутри каждого такого подкаталога находится ряд файлов, в которых содержится полезная информация о соответствующих процессах. Файлы `stat` и `status` хранят статистику работы процесса, `cmdline` -- команда, которой был запущен процесс, `exe` -- символическая ссылка на исполняемый файл программы. Здесь же вы найдете ряд других файлов, но, с точки зрения написания сценариев, они не так интересны, как эти четыре.

Пример. Поиск файла программы по идентификатору процесса.

```
#!/bin/bash
# pid-identifier.sh: Возвращает полный путь к исполняемому файлу программы по
идентификатору процесса (pid).

ARGNO=1  # Число, ожидаемых из командной строки, аргументов.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe

if [ $# -ne $ARGNO ]
then
```

```
  echo "Порядок использования: `basename $0` PID-процесса" >&2  # Сообщение
об ошибке на >stderr.
  exit $E_WRONGARGS
fi

ps ax

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# Проверка наличия процесса с заданным pid в списке, выданном командой  "ps",
поле #1.
# Затем следует убедиться, что этот процесс не был запущен этим сценарием
('ps').
# Это делает последний "grep $1".
if [ -z "$pidno" ]  # Если после фильтрации получается пустая строка,
then                # то это означает, что в системе нет процесса с заданым
pid.
  echo "Нет такого процесса."
  exit $E_NOSUCHPROCESS
fi

# Альтернативный вариант:
#   if ! ps $1 > /dev/null 2>&1
#   then                # в системе нет процесса с заданым pid.
#     echo "Нет такого процесса."
#     exit $E_NOSUCHPROCESS
#    fi

if [ ! -r "/proc/$1/$PROCFILE" ]  # Проверить право на чтение.
then
  echo "Процесс $1 найден, однако..."
  echo "у вас нет права на чтение файла /proc/$1/$PROCFILE."
  exit $E_NOPERMISSION  # Обычный пользователь не имеет прав
                        # на доступ к некоторым файлам в каталоге /proc.
fi
# Последние две проверки могут быть заменены на:
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' -- это не сигнал, но
                                      # команда все равно проверит наличие
                                      # процесса-получателя.
#    then echo "Процесс с данным PID не найден, либо вы не являетесь его вла-
дельцем" >&2
#    exit $E_BADPID
#    fi

exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
# Или      exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
# /proc/pid-number/exe -- это символическая ссылка
# на исполняемый файл работающей программы.
if [ -e "$exe_file" ]  # Если файл /proc/pid-number/exe существует...
then                  # то существует и соответствующий процесс.
  echo "Исполняемый файл процесса #$1: $exe_file."
else
  echo "Нет такого процесса."
fi
# В большинстве случаев, этот, довольно сложный сценарий, может быть заменен
командой
# ps ax | grep $1 | awk '{ print $5 }'
# В большинстве, но не всегда...
# поскольку пятое поле листинга,выдаваемого командой 'ps', это argv[0] про-
цесса,
# а не путь к исполняемому файлу.
#
# Однако, оба следующих варианта должны работать безотказно.
#       find /proc/$1/exe -printf '%l\n'
```

```
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
```

/dev/zero и /dev/null

`/dev/null`

Псевдоустройство `/dev/null` -- это, своего рода, "черная дыра" в системе. Это, пожалуй, самый близкий смысловой эквивалент. Все, что записывается в этот файл, "исчезает" навсегда. Попытки записи или чтения из этого файла не дают, ровным счетом, никакого результата. Тем не менее, псевдоустройство `/dev/null` вполне может пригодиться.

Подавление вывода на `stdout`.

```
cat $filename >/dev/null
# Содержимое файла $filename не появится на stdout.
```

Подавление вывода на `stderr`.

```
rm $badname 2>/dev/null
#           Сообщение об ошибке "уйдет в никуда".
```

Подавление вывода, как на `stdout`, так и на `stderr`.

```
cat $filename 2>/dev/null >/dev/null
# Если "$filename" не будет найден, то вы не увидите сообщения об ошиб-
ке.
# Если "$filename" существует, то вы не увидите его содержимое.
# Таким образом, вышеприведенная команда ничего не выводит на экран.
#
#  Такая методика бывает полезной, когда необходимо лишь проверить код
завершения команды
#+ и нежелательно выводить результат работы команды на экран.
#
# cat $filename &>/dev/null
#     дает тот же результат, автор примечания Baris Cicek.
```

Удаление содержимого файла, сохраняя, при этом, сам файл, со всеми его правами доступа (очистка файла):

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   дает тот же эффект, но не порождает дочерний
процесс.

cat /dev/null > /var/log/wtmp
```

Автоматическая очистка содержимого системного журнала (logfile) (особенно хороша для борьбы с надоедливыми рекламными идентификационными файлами ("cookies")):

Пример. Удаление cookie-файлов.

```
if [ -f ~/.netscape/cookies ]  # Удалить, если имеются.
then
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
# Теперь, все cookie-файлы, вместо того, чтобы сохраняться на диске,
будут "вылетать в трубу".
```

`/dev/zero`

Подобно псевдоустройству `/dev/null`, `/dev/zero` так же является псевдоустройст-

вом, с той лишь разницей, что содержит нули. Информация, выводимая в этот файл, так же бесследно исчезает. Чтение нулей из этого файла может вызвать некоторые затруднения, однако это можно сделать, к примеру, с помощью команды od или шестнадцатиричного редактора. В основном, `/dev/zero` используется для создания заготовки файла с заданой длиной.

Пример. Создание файла подкачки (swapfile), с помощью `/dev/zero`.

```
#!/bin/bash
# Создание файла подкачки.
# Этот сценарий должен запускаться с правами root.
ROOT_UID=0          # Для root -- $UID 0.
E_WRONG_USER=65     # Не root?

FILE=/swap
BLOCKSIZE=1024
MINBLOCKS=40
SUCCESS=0
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "Этот сценарий должен запускаться с правами root."; echo
  exit $E_WRONG_USER
fi

blocks=${1:-$MINBLOCKS}          #  По-умолчанию -- 40 блоков,
                                 #+ если размер не задан из командной
строки.
# Ниже приводится эквивалентный набор команд.
# --------------------------------------------------
# if [ -n "$1" ]
# then
#   blocks=$1
# else
#   blocks=$MINBLOCKS
# fi
# --------------------------------------------------

if [ "$blocks" -lt $MINBLOCKS ]
then
  blocks=$MINBLOCKS              # Должно быть как минимум 40 блоков.
fi

echo "Создание файла подкачки размером $blocks блоков (KB)."
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # "Забить"
нулями.
mkswap $FILE $blocks             # Назначить как файл подкачки.
swapon $FILE                     # Активировать.
echo "Файл подкачки создан и активирован."
exit $SUCCESS
```

Еще одна область применения `/dev/zero` -- "очистка" специального файла заданного размера, например файлов, монтируемых как loopback-устройства или для безопасного удаления файла.

Отладка сценариев.

Командная оболочка Bash не имеет своего отладчика, и не имеет даже каких либо отладочных команд или конструкций. Синтаксические ошибки или опечатки часто вызывают сообщения об ошибках, которые практически никак не помогают при отладке.

Пример. Сценарий, содержащий ошибку.

```
#!/bin/bash
# ex74.sh
# Этот сценарий содержит ошибку.
a=37
if [$a -gt 27 ]
then
  echo $a
fi
```

В результате исполнения этого сценария вы получите такое сообщение:

```
./ex74.sh: [37: command not found
```

Что в этом сценарии может быть неправильно (подсказка: после ключевого слова **if**)?

Пример. Пропущено ключевое слово.

```
#!/bin/bash
# missing-keyword.sh:
# Какое сообщение об ошибке будет выведено, при попытке запустить этот сцена-
рий?

for a in 1 2 3
do
  echo "$a"
# done     # Необходимое ключевое слово 'done' закомментировано.

exit 0
```

На экране появится сообщение:

```
missing-keyword.sh: line 11: syntax error: unexpected end of file
```

Обратите внимание, сообщение об ошибке будет содержать номер не той строки, в которой возникла ошибка, а той, в которой Bash точно установил наличие ошибочной ситуации.

Сообщения об ошибках могут вообще не содержать номера строки, при исполнении которой эта ошибка появилась.

А что делать, если сценарий работает, но не так как ожидалось? Вот пример весьма распространенной логической ошибки.

Пример. test24.

```
#  Ожидается, что этот сценарий будет удалять в текущем каталоге
#+ все файлы, имена которых содержат пробелы.
#  Но он не работает.  Почему?
badname=`ls | grep ' '`
# echo "$badname"
rm "$badname"
exit 0
```

Попробуйте найти ошибку, раскомментарив строку **echo "$badname"**. Инструкция echo очень полезна при отладке сценариев, она позволяет узнать - действительно ли вы получаете то, что ожидали получить.

В данном конкретном случае, команда **rm "$badname"** не дает желаемого результата потому, что переменная $badname взята в кавычки. В результате, **rm** получает единственный аргумент (т.е. команда будет считать, что получила имя одного файла). Частично эта проблема может быть решена за счет удаления кавычек вокруг **$badname** и установки переменной $IFS так, чтобы она содержала только символ перевода строки, **IFS=$'\n'**. Одна-

ко, существует более простой способ выполнить эту задачу.

```
# Правильный способ удаления файлов, в чьих именах содержатся пробелы.
rm *\ *
rm *" "*
rm *' '*
```

В общих чертах, ошибочными можно считать такие сценарии, которые

1. "сыплют" сообщениями о "синтаксических ошибках" или
2. запускаются, но работают не так как ожидалось (логические ошибки).
3. запускаются, делают то, что требуется, но имеют побочные эффекты (логическая бомба).

Инструменты, которые могут помочь при отладке неработающих сценариев

1. команда echo, в критических точках сценария, поможет отследить состояние переменных и отобразить ход исполнения.
2. команда-фильтр **tee**, которая поможет проверить процессы и потоки данных в критических местах.
3. ключи `-n -v -x`

   sh -n scriptname -- проверит наличие синтаксических ошибок, не запуская сам сценарий. Того же эффекта можно добиться, вставив в сценарий команду **set -n** или **set -o noexec**. Обратите внимание, некоторые из синтаксических ошибок не могут быть выявлены таким способом.

   sh -v scriptname -- выводит каждую команду прежде, чем она будет выполнена. Того же эффекта можно добиться, вставив в сценарий команду **set -v** или **set -o verbose**.

   Ключи `-n` и `-v` могут употребляться совместно: **sh -nv scriptname**.

   sh -x scriptname -- выводит, в краткой форме, результат исполнения каждой команды. Того же эффекта можно добиться, вставив в сценарий команду **set -x** или **set -o xtrace**.

   Вставив в сценарий **set -u** или **set -o nounset**, вы будете получать сообщение об ошибке unbound variable всякий раз, когда будет производиться попытка обращения к необъявленной переменной.
4. Функция "assert", предназначенная для проверки переменных или условий, в критических точках сценария. (Эта идея заимствована из языка программирования C.)

   Пример. Проверка условия с помощью функции "assert".

```
#!/bin/bash
# assert.sh
assert ()                 #  Если условие ложно,
{                         #+ выход из сценария с сообщением об ошибке.
  E_PARAM_ERR=98
  E_ASSERT_FAILED=99
  if [ -z "$2" ]          # Недостаточное количество входных парамет-
ров.
  then
    return $E_PARAM_ERR
  fi

  lineno=$2
  if [ ! $1 ]
```

```
  then
    echo "Утверждение ложно:  \"$1\""
    echo "Файл: \"$0\", строка: $lineno"
    exit $E_ASSERT_FAILED
  # else
  #   return
  #   и продолжить исполнение сценария.
  fi
}
a=5
b=4
condition="$a -lt $b"     # Сообщение об ошибке и завершение сценария.
                          #  Попробуйте поменять условие "condition"
                          #+ на что нибудь другое и
                          #+ посмотреть -- что получится.
assert "$condition" $LINENO
# Сценарий продолжит работу только в том случае, если утверждение ис-
тинно.
# Прочие команды.
# ...
echo "Эта строка появится на экране только если утверждение истинно."
# ...
# Прочие команды.
# ...
```

Команда **exit**, в сценарии, порождает сигнал 0, по которому процесс завершает работу, т.е. - сам сценарий. Часто бывает полезным по выходу из сценария выдать "распечатку" переменных.

Установка ловушек на сигналы.

trap

Определяет действие при получении сигнала; так же полезна при отладке.

Сигнал (*signal*) - это просто сообщение, передается процессу либо ядром, либо другим процессом, чтобы побудить процесс выполнить какие либо действия (обычно -- завершить работу). Например, нажатие на **Control-C**, вызывает передачу сигнала SIGINT, исполняющейся программе.

```
trap '' 2
# Игнорировать прерывание 2 (Control-C), действие по сигналу не указа-
но.

trap 'echo "Control-C disabled."' 2
# Сообщение при нажатии на Control-C.
```

Пример. Ловушка на выходе.

```
trap 'echo Список переменных --- a = $a  b = $b' EXIT
# EXIT -- это название сигнала, генерируемого при выходе из сценария.
a=39
b=36
exit 0
# Примечательно, что если закомментировать команду 'exit',
# то это никак не скажется на работе сценария,
# поскольку "выход" из сценария происходит в любом случае.
```

Необязательные параметры (ключи).

Необязательные параметры - это дополнительные ключи (опции), которые оказывают влияние на поведение сценария и/или командной оболочки.

Команда set позволяет задавать дополнительные опции прямо внутри сценария. В том месте сценария, где необходимо, чтобы та или иная опция вступила в силу, вставьте такую конструкцию **set -o option-name**, или в более короткой форме -- **set -option-abbrev**. Эти две формы записи совершенно идентичны по своему действию.

```
#!/bin/bash
set -o verbose
# Вывод команд перед их исполнением.

#!/bin/bash
set -v
# Имеет тот же эффект, что и выше.
```

Для того, чтобы отключить действие той или иной опции, следует вставить конструкцию **set +o option-name**, или **set +option-abbrev**.

```
set -o verbose
# Вывод команд перед их исполнением.
set +o verbose
# Запретить вывод команд перед их исполнением.
command
# команда не выводится.
set -v
# Вывод команд перед их исполнением.
command
set +v
# Запретить вывод команд перед их исполнением.
command
```

Как вариант установки опций, можно предложить указывать их в заголовке сценария (в строке sha-bang) -- *#!*.

```
#!/bin/bash -x
# Далее следует текст сценария.
```

Так же можно указывать дополнительные ключи в командной строке, при запуске сценария. Некоторые из опций работают только если они заданы из командной строки, например *-i* -- ключ интерактивного режима работы скрипта.

```
bash -v script-name
bash -o verbose script-name
```

Ниже приводится список некоторых полезных опций, которые могут быть указаны как в полной форме так и в сокращенной.

Таблица. Ключи Bash

Краткое имя	Полное имя	Описание
-C	noclobber	Предотвращает перезапись файла в операциях перенаправления вывода (не распространяется на конвейеры (каналы) -- >|)
-D	(нет)	Выводит список строк в двойных кавычках, которым предшествует символ $, сам сценарий не исполняется
-a	allexport	Экспорт всех, определенных в сценарии, переменных
-b	notify	Выводит уведомление по завершении фоновой задачи (job) (довольно редко используется в сценариях)

Краткое имя	Полное имя	Описание
`-c ...`	(нет)	Читает команды из **...**
`-f`	noglob	Подстановка имен файлов (globbing) запрещена
`-i`	interactive	Сценарий запускается в *интерактивном* режиме
`-p`	privileged	Сценарий запускается как "suid" (осторожно!)
`-r`	restricted	Сценарий запускается в *ограниченном* режиме.
`-u`	nounset	При попытке обращения к неопределенным переменным, выдает сообщение об ошибке и прерывает работу сценария
`-v`	verbose	Выводит на `stdout` каждую команду прежде, чем она будет исполнена
`-x`	xtrace	Подобна `-v`, но выполняет подстановку команд
`-e`	errexit	Прерывает работу сценария при появлении первой же ошибки (когда команда возвращает ненулевой код завершения)
`-n`	noexec	Читает команды из сценария, но не исполняет их (проверка синтаксиса)
`-s`	stdin	Читает команды с устройства `stdin`
`-t`	(нет)	Выход после исполнения первой команды
`-`	(нет)	Конец списка ключей (опций), последующие аргументы будут восприниматься как позиционные параметры.
`--`	(нет)	Эквивалент предыдущей опции (-).