Теоретические сведения о командном интерпретаторе bash

Основным интерфейсом в операционных системах GNU/Linux является консольный интерфейс с текстовым вводом и выводом данных. Таким образом, управление объектами операционной системы сводится к работе с текстовыми потоками путем выполнения текстовых команд в командном интерпретаторе. Например, состояние процессов отображается в виде набора текстовых файлов в псевдофайловой системе `/proc` (список процессов можно увидеть, отобразив командной `ls /proc` содержимое директории `/proc`) , сведения о событиях в системе хранятся в текстовых файлах журналов (файлы журналов находятся в папке `/var/log`, для их просмотра можно использовать команду `cat`), настройки отдельных пакетов хранятся в текстовых конфигурационных файлах и т.д. Это делает необходимым освоение методов работы с текстовыми потоками для решения дальнейших задач управления операционной системой GNU/Linux.

Управление вводом-выводом команд

У любого процесса по умолчанию всегда открыты три файла – `stdin` (стандартный ввод, клавиатура), `stdout` (стандартный вывод, экран) и `stderr` (стандартный вывод сообщений об ошибках на экран). Эти и любые другие открытые файлы могут быть перенаправлены. В данном случае под "перенаправлением" понимается следующее: получить вывод из файла одной команды (программы, сценария) и передать его на вход в файл другой команды (программы, сценария). Дескрипторы открытых по умолчанию файлов:

```
0 = stdin
1 = stdout
2 = stderr
```

`команда > файл` – перенаправление стандартного вывода `команды` в `файл`, если файл с таким именем уже существует, его старое содержимое удаляется.

`команда >> файл` – перенаправление стандартного вывода `команды` в `файл`, поток дописывается в конец файла.

`команда1 | команда2` – перенаправление стандартного вывода первой команды `команда1` на стандартный ввод второй команды `команда2` (образование **конвейера команд**).

`команда1 $(команда2)` – передача вывода второй команды `команда2` в качестве параметров при запуске первой команды `команда1`. Внутри скрипта конструкция `$(команда2)` может использоваться, например, для передачи результатов работы второй команды в параметры цикла `for … in`.

Внутренние команды bash для работы со строками

`${#string}` – выводит длину строки (`string` – имя переменной);

`${string:position:length}` – извлекает `$length` символов из `$string`, начиная с позиции `$position`. Частный случай:

`${string:position}` извлекает подстроку из `$string`, начиная с позиции `$position`.

`${string#substring}` – удаляет самую короткую из найденных подстрок `$substring` в строке `$string`. Поиск ведется с начала строки. `$substring` – регулярное выражение (см. ниже).

`${string##substring}` – удаляет самую длинную из найденных подстрок `$substring` в строке `$string`. Поиск ведется с начала строки. `$substring` – регулярное выражение.

`${string/substring/replacement}` – замещает первое вхождение `$substring` строкой `$replacement`. `$substring` – регулярное выражение.

`${string//substring/replacement}` – замещает все вхождения `$substring` строкой `$replacement`. `$substring` – регулярное выражение.

Основные внешние команды bash для работы со строками

Рассмотрим основные команды интерпретатора bash для работы со строками:

`sort` – сортирует поток текста в порядке убывания или возрастания, в зависимости от заданных опций.

`uniq` – удаляет повторяющиеся строки из отсортированного файла.

`cut` – извлекает отдельные поля из текстовых файлов (поле – последовательность символов в строке до разделителя).

`head` – выводит начальные строки из файла в `stdout`.

`tail` – выводит последние строки из файла в `stdout`.

`wc` – подсчитывает количество слов/строк/символов в файле или в потоке.

`tr` – заменяет одни символы на другие.

Все вышеперечисленные команды могут принимать на вход различные параметры, влияющие на выдаваемый командой результат. Рекомендуется ознакомиться с полной документацией по этим командам, воспользовавшись командой `man имя_команды`, например, `man sort`. Для просмотра краткого перечня параметров команды можно использовать универсальный ключ `--help` при ее вызове, например, `uniq --help`.

Отдельно стоит выделить полнофункциональные многоцелевые утилиты, предназначенные для обработки строк и текстовых массивов: `grep`, `sed`, `awk`. Рассмотрим их более подробно.

Утилита grep.

Одной из программ, использующих регулярные выражения для работы с текстом, является утилита grep. Она читает текст из файла и выводит те строки, которые совпадают с заданным регулярным выражением. Общий формат вызова утилиты:

```
grep [options] PATTERN [FILE...]
```

где `PATTERN` — регулярное выражение, а `FILE` — один или несколько файлов, к содержимому которых будет применено это регулярное выражение.

Если файл не задан, то `grep` читает текст со стандартного ввода. С помощью опций (*англ.* options) можно управлять поведением `grep`. Например, опция `-v` приводит к выводу всех строк, не совпадающих с заданным регулярным выражением.

Рассмотрим некоторые примеры использования `grep` и регулярных выражений. Команда `ls` выводит список файлов в каталоге. Например, команда `ls /bin` выведет список файлов из каталога `/bin`. Вывод команда `ls` осуществляет в `stdout`.

Предположим, нас интересуют те программы (файлы) из `/bin`, которые содержат подстроку *zip*. Этой подстроке соответствует простейшее регулярное выражение «`zip`». Перенаправляем вывод из `ls` в `grep` и получаем:

```
$ ls /bin | grep 'zip'
bunzip2
bzip2
bzip2recover
gunzip
gzip
```

Здесь регулярное выражение заключено в одиночные кавычки `''`, которые указывают `bash`, что внутри них — обычная строка. Такой синтаксис позволяет использовать в регулярном выражении пробелы, и его разумно придерживаться во всех случаях. Например, регулярное выражение `'a b'` описывает шаблон для строк, содержащих последовательно *a*, пробел и *b*. Если этот шаблон указать grep без кавычек, т.е. `grep a b`, то командный интерпретатор, разобрав строку, вызовет `grep` с двумя параметрами, и `grep` будет искать строки с буквами *a* в файле `b`. При использовании кавычек командный интерпретатор будет считать выражение `'a b'` одним параметром, и передаст его grep целиком, вместе с пробелом внутри.

Файлы из `/bin`, которые кончаются на *2*:

```
$ ls /bin | grep '2$'
bash2
bunzip2
bzip2
```

Файлы из `/bin`, которые начинаются на *b*:

```
$ ls /bin | grep '^b'
basename
bash
bash2
bunzip2
bzcat
bzip2
bzip2recover
```

Файлы из `/bin`, начинающиеся на *b* и содержащие в своём имени букву *a*:

```
$ ls /bin | grep '^b.*a'
basename
bash
bash2
bzcat
```

Здесь в регулярном выражении мы указали, что оно:

- должно совпадать с началом строки — `^`
- в начале строки должна быть буква *b* — `^b`

- дальше может быть любой символ — `^b.`
- и таких «любых» символов может быть сколько угодно — 0 или больше — `^b.*`
- а дальше должна быть буква *a* — `^b.*a`

Файлы из `/bin`, начинающиеся на *b* и содержащие в своём имени буквы *a*, *e* или *k*:

```
$ ls /bin | grep '^b.*[aek]'
basename
bash
bash2
bzcat
bzip2recover
```

Здесь используется описание набора символов — `[aek]`.

Рассмотрим более полезный пример.

Рассмотрим конфигурационный файл сервера `lighttpd` — `/etc/lighttpd/lighttpd.conf`. В нём (как и в большинстве других конфигурационных файлов) содержится большое количество комментариев, как с поясняющим текстом, так и с примерами различных опций настройки. Предположим, нам нужно посмотреть текущую конфигурацию сервера. Однако посмотреть её простой командой `cat /etc/lighttpd/lighttpd.conf` неудобно: текст не помещается на экране. Мы можем, конечно, использовать команду `less` для прокрутки текста, но комментарии при этом всё равно будут мешать. Мы можем удалить их из файла, но тогда сложно будет что-либо изменять в нём в дальнейшем.

Проще отфильтровать ненужный текст непосредственно при выводе файла на экран.

Комментарии в `lighttpd.conf` начинаются с символа # (хеш). Перед ним в начале строки может или не быть ничего, или быть один или несколько пробелов.

Таким образом, регулярное выражение для выделения строк с комментариями — «`^ *#`»: начало строки, ноль или несколько пробелов, и затем — *#*.

Кроме того, нас не очень интересуют просто пустые строки, в которых нет никакого текста. Такие строки можно описать выражением «`^$`»: начало строки, и сразу — её конец. Может быть и другой вариант: строка, состоящая из одних пробелов, которая также не несёт никакой информации. Таким образом, общее регулярное выражение приобретает вид «`^ *$`».

Итого, строкам комментариев соответствует выражение «`^ *#`», а пустым строкам — «`^ *$`». Как было отмечено ранее, фильтру `grep` можно приказать выводить строки, которые не совпадают с регулярным выражением, вызвав его с ключом `-v`.

Выводим файл `lighttpd.conf` в *stdout* и последовательно пропускаем вывод через два фильтра:

```
# cat /etc/lighttpd/lighttpd.conf | grep -v '^ *#' | grep -v '^ *$'
```

Этот вариант не очень эффективен, хотя и приносит желаемый результат. Можно избежать двух последовательных вызовов `grep`, объединив шаблоны. Видно, что они очень похожи: возможные пробелы в начале строки и или # (хеш), или конец строки. Т.е. общий шаблон — «`^ *(#|$)`».

`grep` поддерживает несколько вариантов синтаксиса регулярных выражений и в варианте по умолчанию рассматривает круглые скобки как обычные символы. Поэтому надо или приказать

grep'у рассматривать их как оператор выбора, экранировав скобки символом \ (обратный слеш), или переключить grep в режим работы с расширенным синтаксисом регулярных выражений, вызвав его с ключом -E, или использовать версию grep с включённой по умолчанию поддержкой расширенных регулярных выражений — egrep:

```
# cat /etc/lighttpd/lighttpd.conf | grep -v '^ *\(#\|$\)'
# cat /etc/lighttpd/lighttpd.conf | grep -E -v '^ *(#|$)'
# cat /etc/lighttpd/lighttpd.conf | egrep -v '^ *(#|$)'
```

Ну и наконец, нам не обязательно передавать файл lighttpd.conf на стандартный вход grep/egrep, эти утилиты могут сами прочитать файл с диска:

```
# egrep -v '^ *(#|$)' /etc/lighttpd/lighttpd.conf
```

Утилита sed.

Программа grep выполняет только поиск строк и выводит найденные результаты без изменений. Однако часто бывает необходимо не только найти какой-либо текст, но и изменить его. Для редактирования потока текста можно использовать утилиту sed (от *англ.* Stream EDitor, потоковый редактор). sed используется для выполнения основных преобразований текста, читаемого из файла или поступающего из стандартного потока ввода, и совершает одно действие над вводом за проход. Общий формат вызова sed:

```
sed [options] COMMAND [FILE...]
```

Из большого числа возможных команд sed мы рассмотрим только команду поиска и замены текста. Эта команда имеет вид s/PATTERN/EXPRESSION/ и осуществляет поиск в каждой из входящих строк текста регулярного выражения PATTERN. Результаты совпадения заменяются на выражение EXPRESSION. Результирующий текст выводится в стандартный поток вывода.

Рассмотрим использование команды замены в sed на примерах. В простейшем случае просто поменяем один фрагмент текста на другой:

```
$ ls -1 /var/cache
apt
fontconfig
man
$ ls /var/cache/ | sed 's/apt/APT/'
APT
fontconfig
man
```

В каталоге /var/cache есть несколько файлов, список их можно получить командной ls. Регулярное выражение «apt» совпадает с одной из строк вывода, и мы меняем совпадение на APT.

```
$ ls /var/cache/ | sed 's/a/A/'
Apt
fontconfig
mAn
```

В этом случае мы заменили в выводе ls букву *a* на *A*. sed применяет свои команды для каждой из строк вывода, поэтому в обеих строках, где была буква *a*, она была заменена.

Утилита `uptime` выдаёт определённую статистику по работе системы:

```
$ uptime
07:48:42 up 27 days, 22:13, 1 user, load average: 0.00, 0.00, 0.00
```

Для того, чтобы выделить из этой строки текущее число пользователей в системе, используем `sed`. Число пользователей — это одна или несколько цифр — «`[0-9]\+`», за которыми после пробела (или нескольких пробелов в общем случае) — «`[0-9]\+ \+`» следует слово *user* (или *users*). Нам интересно число пользователей — выберем его в подвыражении: «`\([0-9]\+\) \+user`». В начале строки идёт некоторый текст, отделённый от числа пользователей пробелом: «`^.* \([0-9]\+\) \+user`». Конец строки тоже может быть любой: «`^.* \([0-9]\+\) \+user.*`». Данное выражение совпадает со всей строкой и выделяет в подстроку `\1` число пользователей. Заменив целиком строку на `\1`, мы получим в результате только это число:

```
$ uptime | sed 's/^.* \([0-9]\+\) \+user.*/\1/'
1
```

Аналогично можно получить, например, время работы системы (подстроку вида *27 days, 22:13*):

```
$ uptime | sed 's/^.* up \+\(.\+\), \+[0-9]\+ \+user.*/\1/'
27 days, 22:13
```

Здесь мы отметили, что время работы системы начинается за словом *up*, а после него идёт число пользователей. Соответственно, требующееся регулярное выражение для помещения времени работы системы в подстроку можно описать как:

- любое число любых символов от начала строки, далее пробел и слово *up* — `^.* up`
- за которым следует через один или несколько пробелов время работы системы — `^.* up \+\(\)`
- само время работы системы может содержать фактически любые символы, в т.ч. пробелы, знаки пунктуации и пр. — `^.* up \+\(.\+\)`
- однако за ним через запятую и один или несколько пробелов — `^.* up \+\(.\+\), \+`
- следует количество пользователей (число, одна или несколько цифр) — `^.* up \+\(.\+\), \+[0-9]\+`
- и слово *user* (или *users*). Далее до конца строки может быть что угодно — `^.* up \+\(.\+\), \+[0-9]\+ \+user.*`

Отметим, что то же самое мы могли бы сделать и по-другому: просто удаляя из вывода ненужный нам текст. Например:

```
$ uptime | sed 's/user.*//'
08:18:07 up 27 days, 22:43, 2
```

убирает весь текст от *user* включительно и до конца строки. Также убираем в полученном результате и всё в конце строки от запятой включительно:

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*$//'
08:24:13 up 27 days, 22:49
```

Отметим, что более простой вариант без привязки к концу строки

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*//'
08:24:18 up 27 days, 2
```

из-за «ленивости» регулярных выражений совпадёт с первым вхождением запятой (, *22:43*), а ещё более простой вариант

```
$ uptime | sed 's/user.*//'| sed 's/,.*$//'
08:25:11 up 27 days
```

из-за «жадности» будет совпадать с текстом от первой запятой до конца строки (, *22:43*, *2*).

Далее нам нужно удалить текст от начала строки до *up* включительно:

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*$//' | sed 's/^.*up \+//'
27 days, 22:54
```

и мы получаем требуемый результат. (Символ `\` (обратный слеш) в конце строки здесь означает, что команда будет продолжена на следующей строке).

Утилита awk.

AWK — интерпретируемый скриптовый язык, предназначенный для обработки текстовой информации. Первая версия *AWK* была написана в 1977 году в AT&T Bell Laboratories и получила название по фамилиям своих разработчиков: Альфреда Ахо (Alfred V. Aho), Питера Вейнбергера (Peter J. Weinberger) и Брайана Кернигана (Brian W. Kernighan).

AWK рассматривает входной поток как набор записей, каждая из которых состоит из набора полей. По умолчанию для *AWK* записью является строка, а разделителями полей в строке — пробелы. Внутри программы на *AWK* значение поля можно получить как значение переменной `$1, $2, $3, ...` Переменная `$0` содержит в себе всю запись.

Программа на *AWK* имеет вид

```
PATTERN {ACTION}
PATTERN {ACTION}
...
```

Для каждой строки, совпадающей с шаблоном `PATTERN`, выполняется указанное действие `ACTION`. Если шаблон не указан, то действие выполняется для всех строк. Шаблон — это регулярное выражение, из большого числа возможных действий мы рассмотрим только команду `print`.

Рассмотрим использование команды `awk` на примерах.

Список файлов с указанием их владельцев, прав, и даты последнего изменения можно получить командой `ls -l`. Он имеет вид:

```
$ ls -l /bin | head -n 5
total 5596
lrwxrwxrwx 1 root root       4 Feb 25 05:30 awk -> gawk
-rwxr-xr-x 1 root root   19064 Apr 20  2008 basename
-rwxr-xr-x 1 root root  549368 Mar 27  2008 bash
lrwxrwxrwx 1 root root       4 Feb 25 05:30 bash2 -> bash
```

Преобразуем этот список в формат

```
<имя файла> <владелец>:<группа> <права>
```

`awk` обрабатывает каждую строку списка отдельно, и самостоятельно разбивает её на поля по границам слов. Права файла — поле 1, владелец и группа — поля 3 и 4, имя файла — поле 9. Тогда:

```
$ ls -l /bin | awk '{print $9,$3":"$4,$1;}' | head
: total
awk root:root lrwxrwxrwx
basename root:root -rwxr-xr-x
bash root:root -rwxr-xr-x
bash2 root:root lrwxrwxrwx
bunzip2 root:root lrwxrwxrwx
bzcat root:root lrwxrwxrwx
bzip2 root:root -rwxr-xr-x
bzip2recover root:root -rwxr-xr-x
cat root:root -rwxr-xr-x
```

Можно отфильтровать список и вывести только файлы. Для файлов первый символ поля прав — - (дефис). Для форматирования вывода разделим выводящиеся значения символами табуляции (код символа `\t`). С учётом этого получаем:

```
$ ls -l /bin | awk '/^-/ {print $9"\t->\t"$3":"$4"\t"$1;}' | head
basename -> root:root -rwxr-xr-x
bash -> root:root -rwxr-xr-x
bzip2 -> root:root -rwxr-xr-x
bzip2recover -> root:root -rwxr-xr-x
cat -> root:root -rwxr-xr-x
chgrp -> root:root -rwxr-xr-x
chmod -> root:root -rwxr-xr-x
chown -> root:root -rwxr-xr-x
clock_unsynced -> root:root -rwxr-xr-x
cp -> root:root -rwxr-xr-x
```

Создание скриптов.

До сих пор нами рассматривался запуск программ из командной строки оболочки. Однако для повторяющихся последовательностей команд это неудобно. В таких случаях можно сохранить последовательность команд в файл и запускать их не из командной строки, а из такого файла. Обычно такие файлы с записанными командами называют скриптами.

В простейшем случае, скрипт можно создать, например, так:

```
$ echo "ls | grep script" > script
$ cat script
ls | grep script
$ sh script
script
```

Здесь мы создали текстовый файл, содержащий команды `ls` и `grep`, и далее выполнили эти команды, вызвав интерпретатор команд и передав ему в качестве аргумента имя скрипта. Интерпретатор команд, получив в качестве аргумента имя файла, считал из него команды и выполнил их.

Такой способ запуска скриптов не очень удобен. Он отличается от вызова команд системы: здесь требуется в командной строке указывать имя интерпретатора команд и, в общем случае, полный путь к выполняемому скрипту, в то время как для скомпилированных команд системы достаточно ввести имя самой команды. Кроме того, для операционных систем *nix существует несколько альтернативных командных интерпретаторов с различным синтаксисом команд. Существует и большое количество различных интерпретирующих языков программирования, программы для которых также оформляются в виде скриптов и запускаются с помощью соответствующих программ-интерпретаторов. Таким образом, требуется способ указать системе, каким именно интерпретатором следует выполнять тот или иной скрипт.

Имя программы, которая должна интерпретировать записанную в текстовый файл (скрипт) последовательность команд, можно указать в самом скрипте. Это делается с помощью специальным образом оформленной первой строки скрипта, которая обычно выглядит примерно так:

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

Первая строка состоит из двух символов `#!` (хеш и восклицательный знак), за которыми указывается полный путь к программе, которая будет обрабатывать данный скрипт. В данном случае это интерпретатор команд `bash`. Как правило, интерпретируемые языки программирования (и командный интерпретатор в частности) используют символ `#` (хеш) для выделения комментариев, т. е. интерпретировать подобным образом оформленную строку они не будут.

В операционных системах *nix существуют права доступа к файлам. Если для файла задано право его выполнения, то интерпретатор команд откроет его и прочитает несколько первых символов файла. Если там обнаружится начало скомпилированной программы, то она будет запущена, если же там обнаружится последовательность символов `#!`, то будет запущен указанный после неё интерпретатор, которому будет передано в качестве аргумента имя файла.

Итого:

```
$ echo '#!/bin/bash' > script
$ echo 'ls | grep script' >> script
$ chmod a+x script
$ cat script
#!/bin/bash
ls | grep script
$ ls -l script
-rwxr-xr-x 1 student student 29 Map 20 09:35 script
$ ./script
script
```

Здесь мы создали путём вызова двух команд `echo` файл (обратите внимание, что во второй команде мы дописали строку в имеющий файл), задали этому файлу право на выполнение, проверили результат (выведя файл через `cat` и проверив права на него через `ls -l`) и запустили его на выполнение.

Отметим, что командный интерпретатор ищет выполняемые файлы в определённых местах: `/bin`, `/usr/bin` и т.п. Для запуска программы из нестандартного места требуется указывать путь к ней, т.е., в данном случае, запустить программу как `script` нельзя — вместо созданного нами скрипта

командный интерпретатор запустит стандартную утилиту `script` из `/usr/bin`. Поэтому в команде `./script` явно указывается, что запустить следует файл `script` из текущего каталога.

Часто простого последовательного выполнения команд недостаточно: для эффективного программирования требуются переменные, условные операторы и т.п. Командный интерпретатор имеет собственный язык, который по своим возможностям приближается к высокоуровневым языкам программирования. Этот язык позволяет создавать программы (*shell*-файлы, *shell*-скрипты), которые могут включать операторы языка и команды UNIX.

Такие файлы не требуют компиляции и выполняются в режиме интерпретации, но они, как отмечалось ранее, должны обладать правом на исполнение (устанавливается с помощью команды `chmod`).

Скрипту могут быть переданы аргументы при запуске. Каждому из первых девяти аргументов ставится в соответствие позиционный параметр от `$1` до `$9` (`$0` — имя самого скрипта), и по этим именам к ним можно обращаться из текста скрипта.

Прежде чем начать рассмотрение некоторых операторов *shell*, следует обратить внимание на использование в командах некоторых символов.

- `$` (знак доллара) – используется для подстановки в строку значения переменной, имя которой указывается сразу за ним (`$VAR`).
- `` `` (обратные апострофы) — служат для выполнения команды, заключённой между ними, и подстановки в текущую строку вывода этой команды.
- `\` (обратный слеш) — знак отмены специального значения («экранирования») следующего за ним символа, такого как `$` или `` ` ``. Будучи последним символом в строке, обратный слеш экранирует символ перевода строки и позволяет разбивать запись команд с многочисленными и длинными аргументами на несколько строк.
- `""` (двойные кавычки) — используются для обрамления текста, внутри которого командная оболочка выполняет поиск и интерпретацию специальных символов.
- `''` (одинарные кавычки или апострофы) — используются для обрамления текста, передаваемого как единый аргумент команды или присваиваемого переменной без интерпретирования в нём специальных символов.

Кроме того, для удобства работы с файлами почти все командные интерпретаторы интерпретируют символы `?` (знак вопроса) и `*` (астериск), используя их как шаблоны имен файлов (т.н. метасимволы):

- `?` — один любой символ;
- `*` — произвольное количество любых символов.

Например, `*.c` обозначает все файлы с расширением `c`, `pr???.*` обозначает файлы, имена которых начинаются с `pr`, содержат пять символов и имеют любое расширение.

Переменные языка shell.

Язык *shell* позволяет работать с переменными без предварительного объявления. Имена переменных начинаются с латинской буквы и могут содержать латинские буквы, цифры и символ подчеркивания. Обращение к переменным начинается со знака `$` (знак доллара).

Имеется большое количество уже определённых переменных — т.н. переменных окружения. Их полный список можно получить командой `set`. Переменные окружения используются для настройки различных параметров окружения пользователя, например, в переменной `TMP` задаётся каталог для временных файлов, используемый рядом программ:

```
$ echo $TMP
/tmp/.private/student
$ ls $TMP
mc-student
```

Переопределить (в т.ч. случайно) такие системные переменные можно, но стоит учесть, что это может привести к нежелательным последствиям.

Оператор присваивания.

Присвоение значений переменным осуществляется с помощью оператора `=` (знак равенства). Пробелов между именем переменной, `=` и значением быть не должно. Например:

```
$ A=5
$ B=пять
$ C=$A+$B
$ echo A
A
$ echo B=$B
B=пять
$ echo C=$C
C=5+пять
```

Как мы видим, интерпретатор команд все переменные рассматривает как строки. Однако есть возможность и вычисления арифметических выражений — через внешние программы.

Вычисление выражений.

Вычисление выражений осуществляется с помощью команды `expr` и арифметических и логических операторов:

```
$ a=5 b=12
$ a=`expr $a + 4`
$ d=`expr $b - $a`
$ echo $a $b $d $A
9 12 3 5
```

Для `expr` аргументы и операции обязательно разделяются пробелами (они должны передаться команде как отдельные параметры). Кроме того, мы видим, что имена переменных чувствительны к регистру, `a` и `A` — разные переменные.

Команда `expr` позволяет производить операции только над целочисленными значениями. Для выполнения вычислений с числами с фиксированной точностью или с вещественными значениями можно использовать другие команды (например, калькуляторы `dc` или `bc`) – хотя, в целом, язык *shell* не предназначен для решения вычислительных задач.

Условные выражения.

Ветвление вычислительного процесса осуществляется с помощью оператора `if`:

```
if список_команд1; then
    список_команд2
```

```
[else
    список_команд3]
fi
```

В квадратных скобках указана необязательная часть команды.

`Список_команд` — это одна или несколько команд (для задания пустого списка используется `:` (двоеточие). `Список_команд1` передает оператору `if` код возврата последней команды из списка. Если код равен `0`, то выполняются команды из `списка_команд2`, таким образом нулевой код возврата эквивалентен значению «истина». В противном случае выполняются команды из `списка_команд3`, если он указан.

Проверка условия может осуществляется с помощью команды `test`. Аргументами этой команды могут быть имена файлов, числовые и нечисловые строки. Она используется в следующих режимах:

- Проверка файлов: `test -ключ имя_файла`
  Ключи:
  `-r` файл существует и доступен для чтения;
  `-w` файл существует и доступен для записи;
  `-x` файл существует и доступен для исполнения;
  `-f` файл существует и является обычным файлом (т. е. не каталогом, не файлом устройства и т.п.);
  `-s` файл существует, является обычным файлом и не пуст, т. е. его размер больше 0 байт;
  `-d` файл существует и является каталогом.
- Сравнение чисел: `test число1 -ключ число2`
  Ключи:
  `-eq` равно;
  `-ne` не равно;
  `-lt` меньше;
  `-le` меньше или равно;
  `-gt` больше
  `-ge` больше или равно.
- Сравнение строк: `test [строка1] выражение строка2`

`[-n] строка`	строка не пуста;
`-z строка`	строка пуста;
`строка1 = строка2`	строки равны;
`строка1 != строка2`	строки не равны.

В качестве альтернативой записи `test` можно использовать команду `[` (открывающая квадратная скобка), при этом, например, для проверки существования файла вместо

```
$ if test -f /bin/bash; then echo 'bash найден!'; fi
bash найден!
```

можно использовать более аккуратно выглядящую конструкцию

```
$ if [ -f /bin/bash ]; then echo 'bash найден!'; fi
bash найден!
```

Построение циклов.

В языке командного интерпретатора существует три типа циклов: `while`, `until` и `for`.

Цикл `while`:

```
while список_команд1; do
    список_команд2
done
```

В условии учитывается код возврата последней выполненной команды из `списка_команд1`, при этом `0` интерпретируется как «истина». Если код возврата последней команды из `списка_команд1` не равен `0`, то выполнение цикла прекращается.

Цикл `until`:

```
until список_команд1; do
    список_команд2{;|перевод строки}
done
```

Проверка условия выполняется **перед** выполнением цикла. Учитывается код возврата последней выполненной команды из `списка_команд1`, при этом цикл выполняется до тех пор, пока код возврата не примет значение «истина», т. е. будет равным нулю.

Цикл `for`:

```
for переменная [in список_значений]; do
    список_команд
done
```

Переменной присваивается значение очередного слова из `списка_значений`, и для этого значения выполняется `список_команд`. Количество итераций равно количеству цепочек символов в `списке_значений`, разделённых пробелами. Если ключевое слово `in` и `список_значений` опущены как необязательные, то переменной поочередно присваиваются значения параметров, переданных при запуске программы-скрипта. В качестве передаваемых параметров можно использовать шаблоны имён файлов, тогда интерпретатор превращает эти шаблоны в список имён файлов, удовлетворяющих шаблону.

Например,

```
$ A=1; for i in `ls /bin | grep '^b'`; do
> echo "$A :$i"
> A=`expr $A + 1`
> done
1 :basename
2 :bash
```