

## Практическая работа №1

### По дисциплине «Системное программное обеспечение»

#### На тему «Использование командной строки Linux»

#### Цель работы:

Получить навыки по работе с командной строкой операционной системы Linux при помощи дистрибутива Debian 10.

#### Часть 1. Основные команды командной строки

- 1 Используя VirtualBox создать виртуальную машину с Debian 10;
- 2 Познакомиться с базовыми командами работы с файловой системой:
  - `cd` – команда для смены директории, работа с абсолютными и относительными путями;
  - `pwd` – команда отображения текущего каталога;
  - `ls` – команда просмотра содержимого директории, разобраться с флагами `-a`, `-l`;
  - `mkdir` – команда создания директорий;
  - `touch` – команда создания файла;
  - `echo` – команда вывода данных в консоль, перенаправление потока вывода в файл при помощи `>` и `>>`;
  - `cat` – команда вывода данных из файла;
    - Вывод текста из файла  
`cat filename`
    - Ввод текста в файл  
`cat > test`  
`test`  
`test`  
`123`  
`Ctrl + D`
    - Ввод многострочного текста в файл  
`cat << EOF >> test`  
`test123`  
`EOF`
  - `rm` и `rmdir` – команды удаления файлов и каталогов;
  - `mv` – команда перемещения файлов и каталогов;
  - `cp` – команда копирования файлов и каталогов;
  - `find` – поиск файлов;
  - `more`, `less`, `head`, `tail` для вывода информации из файла

- **Команда head** выводит первых 10 строк файла на экран эмулятора терминала хотя ее поведение можно изменить при помощи параметров.
  - **Команда tail** в своем стандартном режиме работы выводит последних 10 строк файл на экран, ее поведение также меняется параметрами.
  - **Команда less в Linux** используется для просмотра больших файлов, использовать ее удобнее, чем команду cat, поскольку она разбивает содержимое файла на страницы и дает возможность осуществлять поиск по файлу.
- Команда more** не рекомендована к использованию, так как есть команда less, ее я упоминаю лишь потому, что есть некоторые дистрибутивы Linux, в которых есть more, но нет less.

### 3 Познакомиться с командами управления правами

- Команда chmod  
**chmod [опции] <права> </путь/к/файлу>**  
\$ chmod 766 file  
Есть три основных вида прав:

**r - чтение;**

**w - запись;**

**x - выполнение;**

**s - выполнение от имени суперпользователя (дополнительный);**

Также есть три категории пользователей, для которых вы можете установить эти права на файл linux:

**u - владелец файла;**

**g - группа файла;**

**o - все остальные пользователи.**

Синтаксис настройки прав такой:

**группа\_пользователейдействиевид\_прав**

В качестве действий могут использоваться знаки "+" - включить или "-" - отключить. Рассмотрим несколько примеров:

u+x - разрешить выполнение для владельца;

ugo+x - разрешить выполнение для всех;

ug+w - разрешить запись для владельца и группы;

o-x - запретить выполнение для остальных пользователей;

ugo+gwx - разрешить все для всех;

Но права можно записывать не только таким способом. Есть еще восьмеричный формат записи, он более сложен для понимания, но пишется короче и проще.

- 0 - никаких прав;
- 1 - только выполнение;
- 2 - только запись;
- 3 - выполнение и запись;
- 4 - только чтение;
- 5 - чтение и выполнение;
- 6 - чтение и запись;
- 7 - чтение запись и выполнение.

Права на папку `linux` такие же, как и для файла. Во время установки прав сначала укажите цифру прав для владельца, затем для группы, а потом для остальных. Например:

- 744 - разрешить все для владельца, а остальным только чтение;
- 755 - все для владельца, остальным только чтение и выполнение;
- 764 - все для владельца, чтение и запись для группы, и только чтение для остальных;
- 777 - всем разрешено все.

Каждая из цифр не зависит от предыдущих, вы выбираете именно то, что вам нужно. Теперь давайте рассмотрим несколько опций команды, которые нам понадобятся во время работы:

- c - выводить информацию обо всех изменениях;
- f - не выводить сообщения об ошибках;
- v - выводить максимум информации;
- preserve-root - не выполнять рекурсивные операции для корня "/";
- reference - взять маску прав из указанного файла;
- R - включить поддержку рекурсии;
- version - вывести версию утилиты.
- Команда `chown`  
**`chown <пользователь> [опции] </путь/к/файлу>`**  
`$ chown root ./dir1`  
В поле пользователь надо указать пользователя, которому мы хотим передать файл. Также можно указать через двоеточие группу, например, **пользователь:группа**. Тогда изменится не только пользователь, но и группа. Вот основные опции, которые могут вам понадобиться:
  - c, --changes - подробный вывод всех выполняемых изменений;
  - f, --silent, --quiet - минимум информации, скрыть сообщения об ошибках;

--dereference - изменять права для файла к которому ведет символическая ссылка вместо самой ссылки (поведение по умолчанию);

-h, --no-dereference - изменять права символических ссылок и не трогать файлы, к которым они ведут;

--from - изменять пользователя только для тех файлов, владельцем которых является указанный пользователь и группа;

-R, --recursive - рекурсивная обработка всех подкаталогов;

-H - если передана символическая ссылка на директорию - перейти по ней;

-L - переходить по всем символическим ссылкам на директории;

-P - не переходить по символическим ссылкам на директории (по умолчанию).

#### 4 Перенаправление потока вывода

Как было сказано выше, существует возможность перенаправить вывод команды из консоли в файл при помощи следующих команд:

- > - перезапись файла;
- >> - добавление в файл.

Существует также вариант перенаправления вывода на ввод другой команде.

| - данный символ как раз позволяет это сделать.

Пример:

```
root@debian:/etc# ls -al | grep sub
-rw-r--r-- 1 root root 17 фев 9 16:57 subgid
-rw-r--r-- 1 root root 0 фев 9 16:48 subgid-
-rw-r--r-- 1 root root 17 фев 9 16:57 subuid
-rw-r--r-- 1 root root 0 фев 9 16:48 subuid-
```

Команда **grep** позволяет выводить только те строки, в которых было найдено совпадение с заданным шаблоном. В данном случае при помощи команды **ls -al** был получен список всех файлов из директории **/etc**, который затем был отфильтрован по присутствию в них строки **sub**.

## Часть 2. Анализ текста

Данная часть практической работы посвящена разбору и анализу текста при помощи командной строки linux.

Для этого возможно использовать команды **sed** и **awk** а также регулярные выражения.

Предлагается ознакомиться с данными темами по следующим ссылкам:

**sed:** <https://habr.com/ru/company/ruvds/blog/327530/>

Замена слова test на another test

```
$ echo "This is a test" | sed 's/test/another test/'
```

Аналогично для файлов

```
$ sed 's/test/another test' ./myfile
```

Несколько команд

```
$ sed -e 's/This/That/; s/test/another test/' ./myfile
```

### Флаги

При передаче номера учитывается порядковый номер вхождения шаблона в строку, заменено будет именно это вхождение.

- Флаг **g** указывает на то, что нужно обработать все вхождения шаблона, имеющиеся в строке.  

```
$ sed 's/test/another test/g' myfile
```
- Флаг **p** указывает на то, что нужно вывести содержимое исходной строки.  

```
$ sed -n 's/test/another test/p' myfile
```
- Флаг вида **w file** указывает команде на то, что нужно записать результаты обработки текста в файл.  

```
$ sed 's/test/another test/w output' myfile
```

Заменить разделители

Заменить /bin/bash на /bin/csh в файле /etc/passwd

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

Заменить test на another test во второй-третьей строках

```
$ sed '2,3s/test/another test/' myfile
```

Заменить со второй до конца

```
$ sed '2,$s/test/another test/' myfile
```

Удалить 3 строку

```
$ sed '3d' myfile
```

Удалить по шаблону: везде, где есть слово test

```
$ sed '/test/d' myfile
```

Удалить строки между теми, в которых содержатся слова

```
$ sed '/second/,/fourth/d' myfile
```

Вставка текста в поток

\* Команда i добавляет новую строку перед заданной.

```
$ echo "Another test" | sed 'i\First test '
```

\* Команда a добавляет новую строку после заданной.

```
$ echo "Another test" | sed 'a\First test '
```

Замена строк

Заменить содержание 3-ей строки

```
$ sed '3c\This is a modified line.' myfile
```

Вывести номера строк

```
$ sed '=' myfile
```

Вывести номера строк без текста

```
$ sed -n '/test/=' myfile
```

Чтение из файла

```
$ sed '3r newfile' myfile
```

**awk:** <https://habr.com/ru/company/ruvds/blog/327754/>

Ключи:

- -F fs — позволяет указать символ-разделитель для полей в записи.
- -f file — указывает имя файла, из которого нужно прочесть awk-скрипт.
- -v var=value — позволяет объявить переменную и задать её значение по умолчанию, которое будет использовать awk.
- -mf N — задаёт максимальное число полей для обработки в файле данных.
- -mr N — задаёт максимальный размер записи в файле данных.
- -W keyword — позволяет задать режим совместимости или уровень выдачи предупреждений awk.

Вывод текста. Мы не указали файл с данными, поэтому сначала введем что-нибудь  
awk '{print "Welcome to awk command tutorial"}'

По умолчанию `awk` назначает следующие переменные каждому полю данных, обнаруженному им в записи:

`$0` — представляет всю строку текста (запись).

`$1` — первое поле.

`$2` — второе поле.

`$n` — `n`-ное поле.

Вывод первого слова из каждой строки

```
$ awk '{print $1}' myfile
```

Использование нескольких команд

Здесь Том замениться на Адама

```
$ echo "My name is Tom" | awk '{$4="Adam"; print $0}'
```

Вывод данных

```
awk 'BEGIN {print "Hello World!"}'
```

Вывод всех первых слов из строк файла

```
$ awk 'BEGIN {print "The File Contents:"}  
{print $0}' myfile
```

`BEGIN` — начало команд, `END` — команды после обработки данных

## Что такое регулярные выражения

У многих, когда они впервые видят регулярные выражения, сразу же возникает мысль, что перед ними бессмысленное нагромождение символов. Но это, конечно, далеко не так. Взгляните, например, на это регулярное выражение

```
^([a-zA-Z0-9_-\.\+])@([a-zA-Z0-9_-\.\+])\.([a-zA-Z]{2,5})$
```

На наш взгляд даже абсолютный новичок сходу поймёт, как оно устроено и зачем нужно :) Если же вам не вполне понятно — просто читайте дальше и всё встанет на свои места.

Регулярное выражение — это шаблон, пользуясь которым программы вроде `sed` или `awk` фильтруют тексты. В шаблонах используются обычные ASCII-символы, представляющие сами себя, и так называемые метасимволы, которые играют особую роль, например, позволяя ссылаться на некие группы символов.

## Типы регулярных выражений

Реализации регулярных выражений в различных средах, например, в языках программирования вроде `Java`, `Perl` и `Python`, в инструментах `Linux` вроде `sed`, `awk` и `grep`, имеют определённые особенности. Эти особенности зависят от так называемых движков обработки регулярных выражений, которые занимаются

интерпретацией шаблонов.

В Linux имеется два движка регулярных выражений:

- Движок, поддерживающий стандарт POSIX Basic Regular Expression (BRE).
- Движок, поддерживающий стандарт POSIX Extended Regular Expression (ERE).

Большинство утилит Linux соответствуют, как минимум, стандарту POSIX BRE, но некоторые утилиты (в их числе — `sed`) понимают лишь некое подмножество стандарта BRE. Одна из причин такого ограничения — стремление сделать такие утилиты как можно более быстрыми в деле обработки текстов.

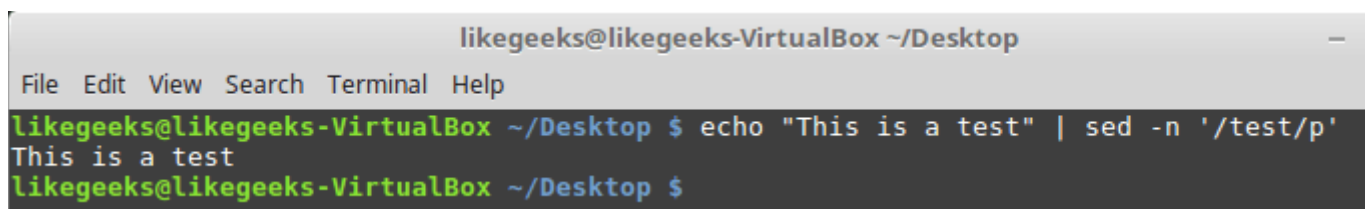
Стандарт POSIX ERE часто реализуют в языках программирования. Он позволяет пользоваться большим количеством средств при разработке регулярных выражений. Например, это могут быть специальные последовательности символов для часто используемых шаблонов, вроде поиска в тексте отдельных слов или наборов цифр. Awk поддерживает стандарт ERE.

Существует много способов разработки регулярных выражений, зависящих и от мнения программиста, и от особенностей движка, под который их создают. Непросто писать универсальные регулярные выражения, которые сможет понять любой движок. Поэтому мы сосредоточимся на наиболее часто используемых регулярных выражениях и рассмотрим особенности их реализации для `sed` и `awk`.

## Регулярные выражения POSIX BRE

Пожалуй, самый простой шаблон BRE представляет собой регулярное выражение для поиска точного вхождения последовательности символов в тексте. Вот как выглядит поиск строки в `sed` и `awk`:

```
$ echo "This is a test" | sed -n '/test/p'
$ echo "This is a test" | awk '/test/{print $0}'
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows two commands being executed. The first command is 'echo "This is a test" | sed -n "/test/p"', which outputs 'This is a test'. The second command is 'echo "This is a test" | awk "/test/{print \$0}"', which also outputs 'This is a test'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

*Поиск текста по шаблону в sed*



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/test/{print
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Поиск текста по шаблону в awk*

Можно заметить, что поиск заданного шаблона выполняется без учёта точного места нахождения текста в строке. Кроме того, не имеет значение и количество вхождений. После того, как регулярное выражение найдёт заданный текст в любом месте строки, строка считается подходящей и передаётся для дальнейшей обработки.

Работая с регулярными выражениями нужно учитывать то, что они чувствительны к регистру символов:

```
$ echo "This is a test" | awk '/Test/{print $0}'
$ echo "This is a test" | awk '/test/{print $0}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/Test/{print
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/test/{print
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Регулярные выражения чувствительны к регистру*

Первое регулярное выражение совпадений не нашло, так как слово «test», начинающееся с заглавной буквы, в тексте не встречается. Второе же, настроенное на поиск слова, написанного прописными буквами, обнаружило в потоке подходящую строку.

В регулярных выражениях можно использовать не только буквы, но и пробелы, и цифры:

```
$ echo "This is a test 2 again" | awk '/test 2/{print $0}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test 2 again" | awk '/test
This is a test 2 again
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Поиск фрагмента текста, содержащего пробелы и цифры*

Пробелы воспринимаются движком регулярных выражений как обычные символы.

## Специальные символы

При использовании различных символов в регулярных выражениях надо учитывать некоторые особенности. Так, существуют некоторые специальные символы, или метасимволы, использование которых в шаблоне требует особого подхода. Вот они:

```
. * [ ] ^ $ { } \ + ? | ( )
```

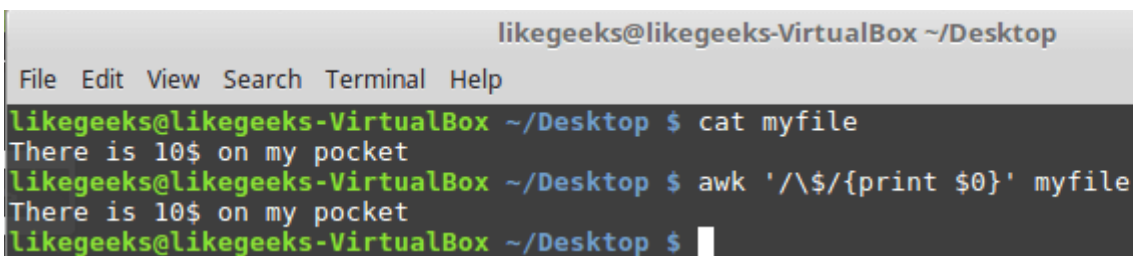
Если один из них нужен в шаблоне, его нужно будет экранировать с помощью обратной косой черты (обратного слэша) — \.

Например, если в тексте нужно найти знак доллара, его надо включить в шаблон, предварив символом экранирования. Скажем, имеется файл myfile с таким текстом:

```
There is 10$ on my pocket
```

Знак доллара можно обнаружить с помощью такого шаблона:

```
$ awk '/\${print $0}' myfile
```

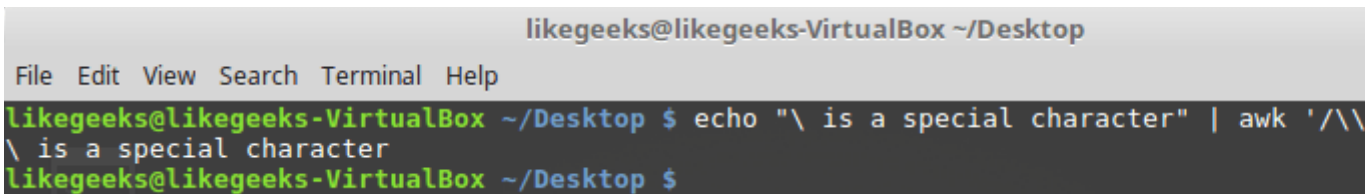


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
There is 10$ on my pocket
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/\${print $0}' myfile
There is 10$ on my pocket
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Использование в шаблоне специального символа

Кроме того, обратная косая черта — это тоже специальный символ, поэтому, если нужно использовать его в шаблоне, его тоже надо будет экранировать. Выглядит это как два слэша, идущих друг за другом:

```
$ echo "\ is a special character" | awk '/\\/ {print $0}'
```

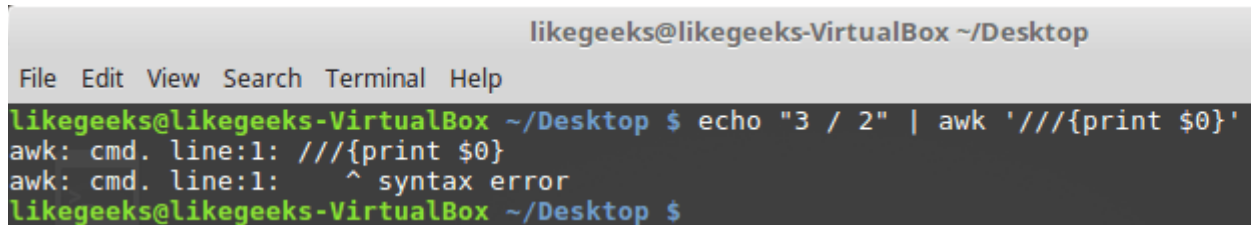


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "\ is a special character" | awk '/\\/ {print $0}'
\ is a special character
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Экранирование обратного слэша

Хотя прямой слэш и не входит в приведённый выше список специальных символов, попытка воспользоваться им в регулярном выражении, написанном для sed или awk, приведёт к ошибке:

```
$ echo "3 / 2" | awk '///{print $0}'
```

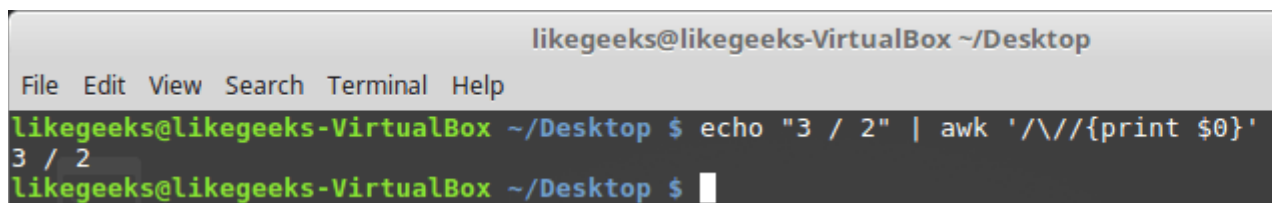


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "3 / 2" | awk '///{print $0}'
awk: cmd. line:1: ///{print $0}
awk: cmd. line:1:      ^ syntax error
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Неправильное использование прямого слэша в шаблоне*

Если он нужен, его тоже надо экранировать:

```
$ echo "3 / 2" | awk '/\\/{print $0}'
```



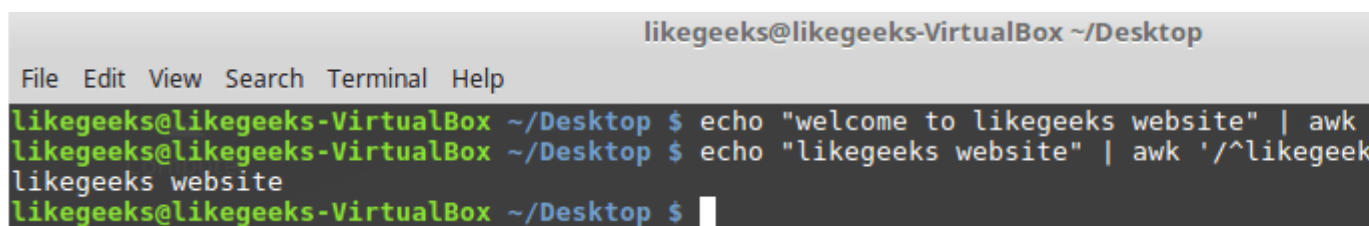
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "3 / 2" | awk '/\\/{print $0}'
3 / 2
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Экранирование прямого слэша*

## Якорные символы

Существуют два специальных символа для привязки шаблона к началу или к концу текстовой строки. Символ «крышка» — ^ позволяет описывать последовательности символов, которые находятся в начале текстовых строк. Если искомый шаблон окажется в другом месте строки, регулярное выражение на него не отреагирует. Выглядит использование этого символа так:

```
$ echo "welcome to likegeeks website" | awk '/^likegeeks/{print $0}'
$ echo "likegeeks website" | awk '/^likegeeks/{print $0}'
```

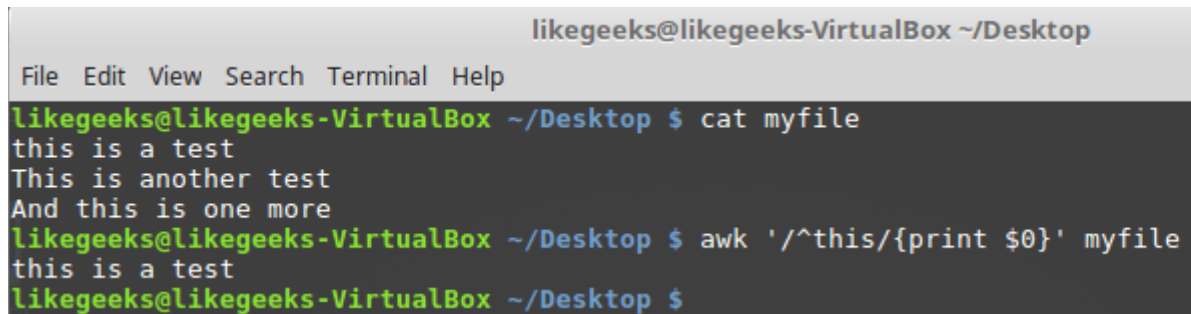


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "welcome to likegeeks website" | awk '/^likegeeks/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "likegeeks website" | awk '/^likegeeks/{print $0}'
likegeeks website
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Поиск шаблона в начале строки*

Символ ^ предназначен для поиска шаблона в начале строки, при этом регистр символов так же учитывается. Посмотрим, как это отразится на обработке текстового файла:

```
$ awk '/^this/{print $0}' myfile
```

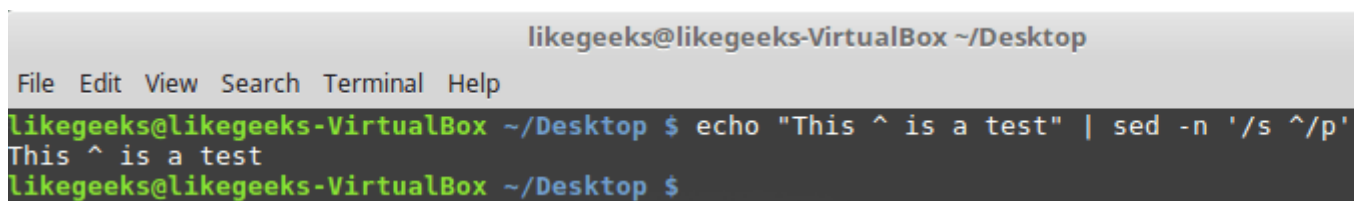


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/^this/{print $0}' myfile
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Поиск шаблона в начале строки в тексте из файла*

При использовании sed, если поместить крышку где-нибудь внутри шаблона, она будет восприниматься как любой другой обычный символ:

```
$ echo "This ^ is a test" | sed -n '/s ^/p'
```

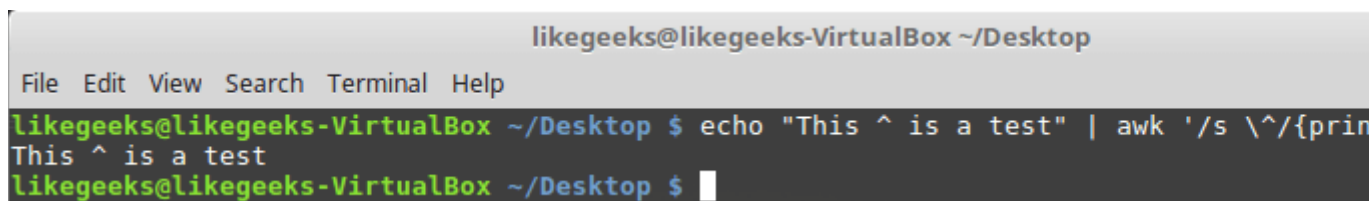


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Крышка, находящаяся не в начале шаблона в sed*

В awk, при использовании такого же шаблона, данный символ надо экранировать:

```
$ echo "This ^ is a test" | awk '/s \^/{print $0}'
```



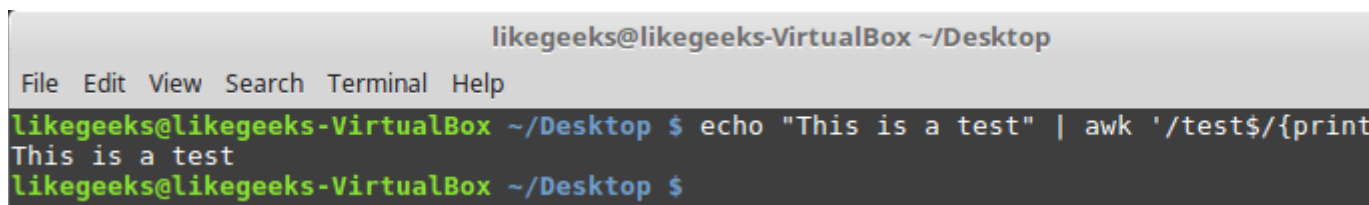
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This ^ is a test" | awk '/s \^/{print $0}'
This ^ is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Крышка, находящаяся не в начале шаблона в awk*

С поиском фрагментов текста, находящихся в начале строки мы разобрались. Что, если надо найти нечто, расположенное в конце строки?

В этом нам поможет знак доллара — \$, являющийся якорным символом конца строки:

```
$ echo "This is a test" | awk '/test$/{print $0}'
```

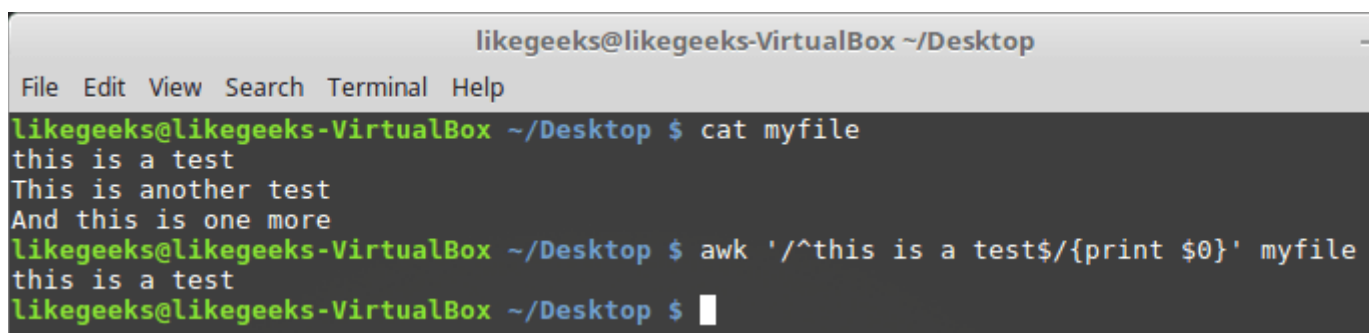


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/test$/{print $0}'
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Поиск текста, находящегося в конце строки*

В одном и том же шаблоне можно использовать оба якорных символа. Выполним обработку файла myfile, содержимое которого показано на рисунке ниже, с помощью такого регулярного выражения:

```
$ awk '/^this is a test$/{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/^this is a test$/{print $0}' myfile
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Шаблон, в котором использованы специальные символы начала и конца строки*

Как видно, шаблон среагировал лишь на строку, полностью соответствующую заданной последовательности символов и их расположению.

Вот как, пользуясь якорными символами, отфильтровать пустые строки:

```
$ awk '!/^$/{print $0}' myfile
```

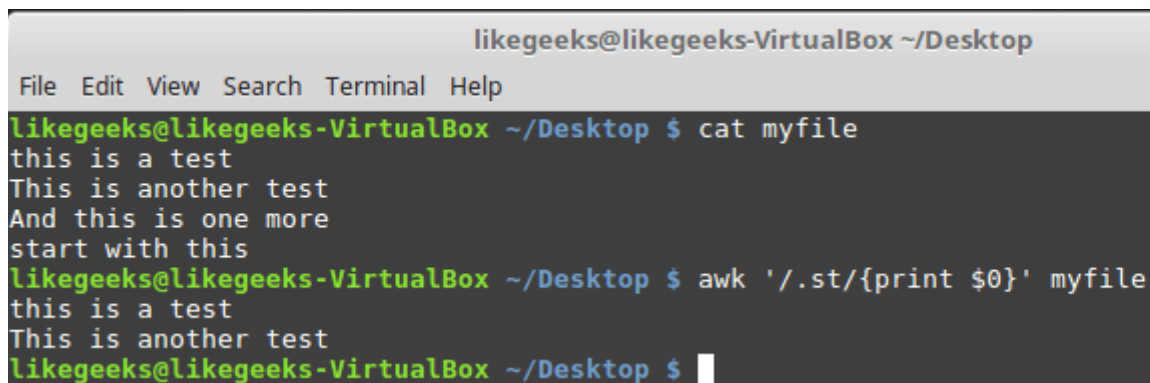
В данном шаблоне использовал символ отрицания, восклицательный знак — !. Благодаря использованию такого шаблона выполняется поиск строк, не содержащих ничего между началом и концом строки, а благодаря восклицательному знаку на печать выводятся лишь строки, которые не соответствуют этому шаблону.

## Символ «точка»

Точка используется для поиска любого одиночного символа, за исключением

символа перевода строки. Передадим такому регулярному выражению файл myfile, содержимое которого приведено ниже:

```
$ awk '/.st/{print $0}' myfile
```



The screenshot shows a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'. The user enters 'cat myfile', and the output is: 'this is a test', 'This is another test', 'And this is one more', 'start with this'. Then the user enters 'awk '/.st/{print \$0}' myfile', and the output is: 'this is a test', 'This is another test'. The prompt is now 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' with a cursor.

### *Использование точки в регулярных выражениях*

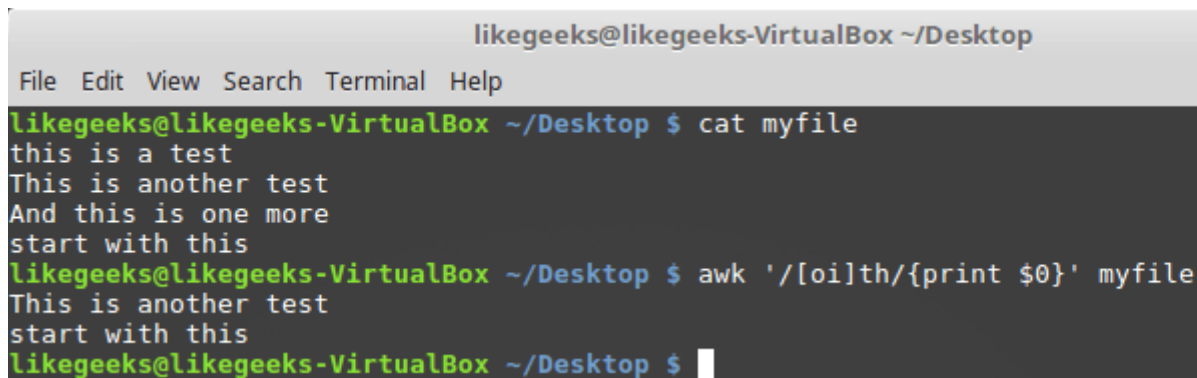
Как видно по выведенным данным, шаблону соответствуют лишь первые две строки из файла, так как они содержат последовательность символов «st», предварённую ещё одним символом, в то время как третья строка подходящей последовательности не содержит, а в четвёртой она есть, но находится в самом начале строки.

## Классы символов

Точка соответствует любому одиночному символу, но что если нужно более гибко ограничить набор искомых символов? В подобной ситуации можно воспользоваться классами символов.

Благодаря такому подходу можно организовать поиск любого символа из заданного набора. Для описания класса символов используются квадратные скобки — []:

```
$ awk '/[oi]th/{print $0}' myfile
```



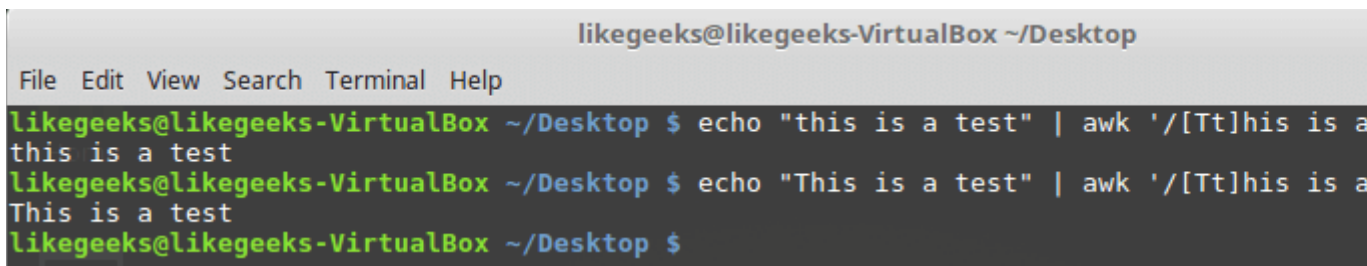
The screenshot shows a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'. The user enters 'cat myfile', and the output is: 'this is a test', 'This is another test', 'And this is one more', 'start with this'. Then the user enters 'awk '/[oi]th/{print \$0}' myfile', and the output is: 'This is another test', 'start with this'. The prompt is now 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' with a cursor.

### *Описание класса символов в регулярном выражении*

Тут мы ищем последовательность символов «th», перед которой есть символ «o» или символ «i».

Классы оказываются очень кстати, если выполняется поиск слов, которые могут начинаться как с прописной, так и со строчной буквы:

```
$ echo "this is a test" | awk '/[Tt]his is a test/{print $0}'
this is a test
$ echo "This is a test" | awk '/[Tt]his is a test/{print $0}'
This is a test
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows two commands being executed. The first command is 'echo "this is a test" | awk '/[Tt]his is a test/{print \$0}'' and the output is 'this is a test'. The second command is 'echo "This is a test" | awk '/[Tt]his is a test/{print \$0}'' and the output is 'This is a test'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

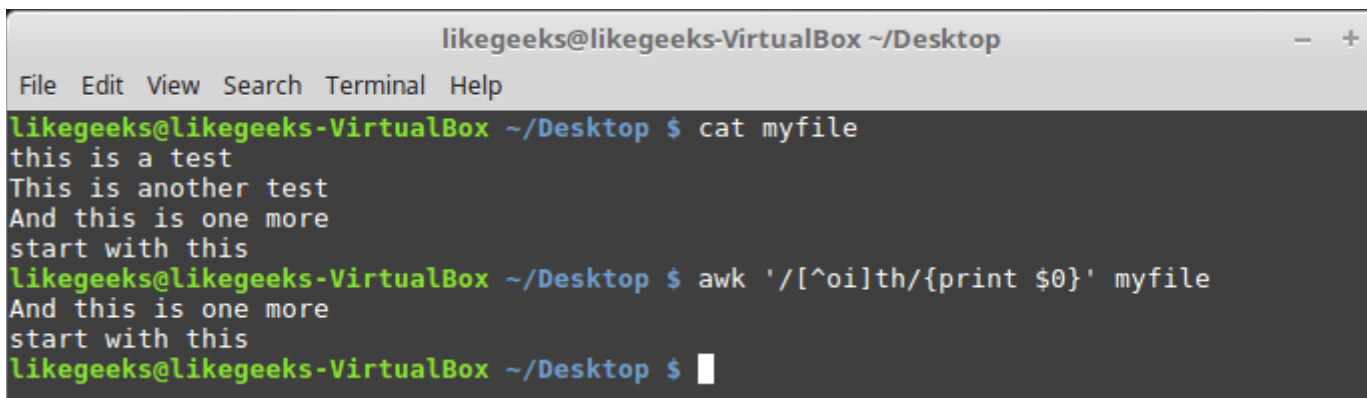
*Поиск слов, которые могут начинаться со строчной или прописной буквы*

Классы символов не ограничены буквами. Тут можно использовать и другие символы. Нельзя заранее сказать, в какой ситуации понадобятся классы — всё зависит от решаемой задачи.

## Отрицание классов символов

Классы символов можно использовать и для решения задачи, обратной описанной выше. А именно, вместо поиска символов, входящих в класс, можно организовать поиск всего, что в класс не входит. Для того, чтобы добиться такого поведения регулярного выражения, перед списком символов класса нужно поместить знак ^. Выглядит это так:

```
$ awk '/[^oi]th/{print $0}' myfile
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows two commands being executed. The first command is 'cat myfile' and the output is 'this is a test', 'This is another test', 'And this is one more', and 'start with this'. The second command is 'awk '/[^oi]th/{print \$0}' myfile' and the output is 'And this is one more' and 'start with this'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

*Поиск символов, не входящих в класс*

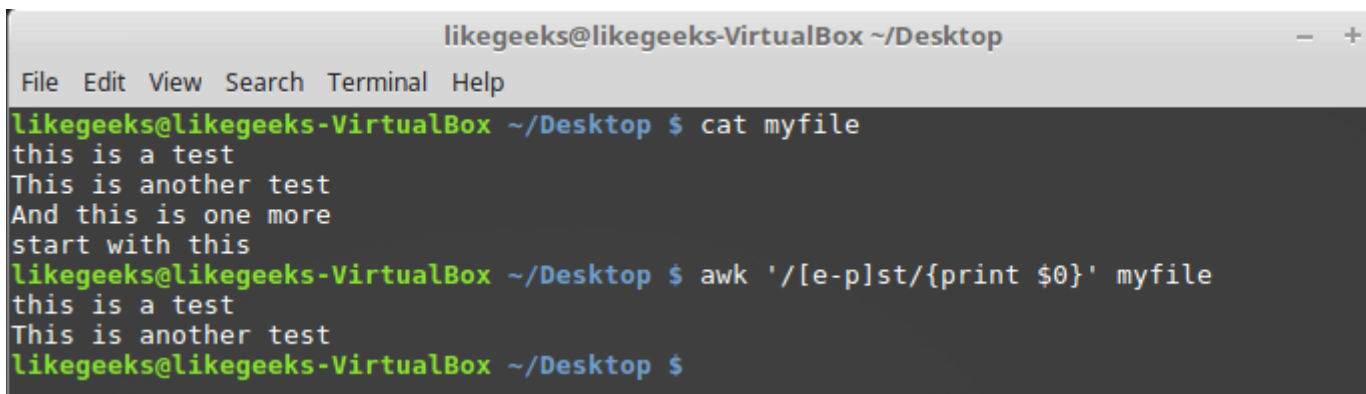


В данном случае будут найдены последовательности символов «th», перед которыми нет ни «o», ни «i».

## Диапазоны символов

В символьных классах можно описывать диапазоны символов, используя тип:

```
$ awk '/[e-p]st/{print $0}' myfile
```



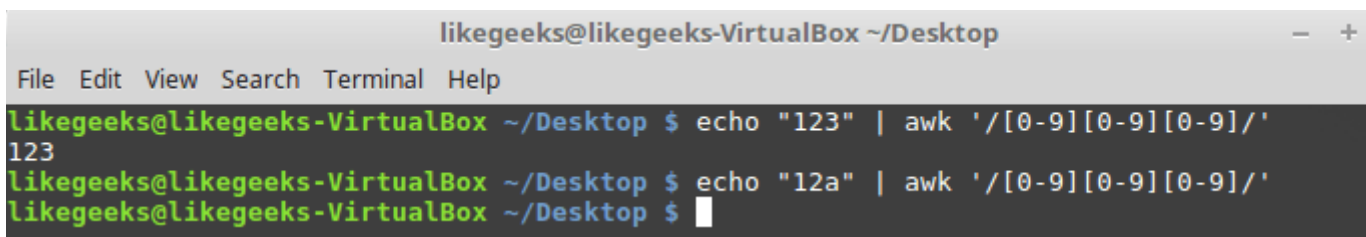
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/[e-p]st/{print $0}' myfile
this is a test
This is another test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Описание диапазона символов в символьном классе*

В данном примере регулярное выражение реагирует на последовательность символов «st», перед которой находится любой символ, расположенный, в алфавитном порядке, между символами «e» и «p».

Диапазоны можно создавать и из чисел:

```
$ echo "123" | awk '/[0-9][0-9][0-9]/'
123
$ echo "12a" | awk '/[0-9][0-9][0-9]/'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "123" | awk '/[0-9][0-9][0-9]/'
123
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "12a" | awk '/[0-9][0-9][0-9]/'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Регулярное выражение для поиска трёх любых чисел*

В класс символов могут входить несколько диапазонов:

```
$ awk '/[a-fm-z]st/{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/[a-fm-z]st/{print $0}' myfile
this is a test
This is another test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Класс символов, состоящий из нескольких диапазонов*

Данное регулярное выражение найдёт все последовательности «st», перед которыми есть символы из диапазонов a-f и m-z.

## Специальные классы символов

В BRE имеются специальные классы символов, которые можно использовать при написании регулярных выражений:

- `[:alpha:]` — соответствует любому алфавитному символу, записанному в верхнем или нижнем регистре.
- `[:alnum:]` — соответствует любому алфавитно-цифровому символу, именно — символам в диапазонах 0-9, A-Z, a-z.
- `[:blank:]` — соответствует пробелу и знаку табуляции.
- `[:digit:]` — любой цифровой символ от 0 до 9.
- `[:upper:]` — алфавитные символы в верхнем регистре — A-Z.
- `[:lower:]` — алфавитные символы в нижнем регистре — a-z.
- `[:print:]` — соответствует любому печатаемому символу.
- `[:punct:]` — соответствует знакам препинания.
- `[:space:]` — пробельные символы, в частности — пробел, знак табуляции, символы NL, FF, VT, CR.

Использовать специальные классы в шаблонах можно так:

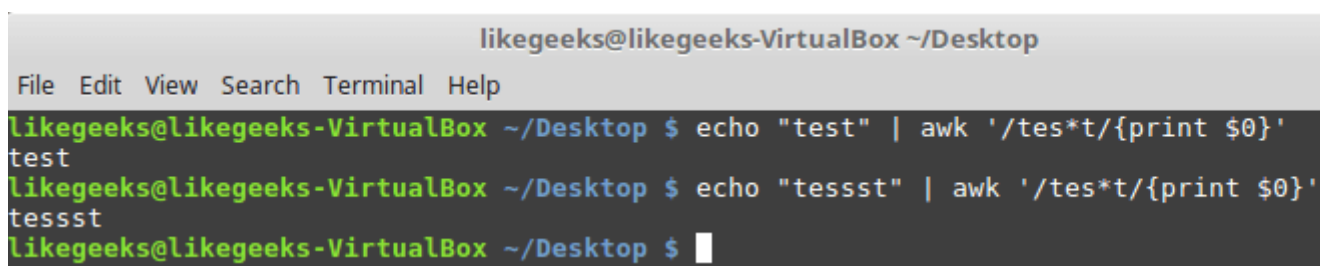
```
$ echo "abc" | awk '/[:alpha:]/{print $0}'
abc
$ echo "abc" | awk '/[:digit:]/{print $0}'
$ echo "abc123" | awk '/[:digit:]/{print $0}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "abc" | awk '/[:alpha:]/{print $0}'
abc
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "abc" | awk '/[:digit:]/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "abc123" | awk '/[:digit:]/{print $0}'
abc123
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

## Символ «звёздочка»

Если в шаблоне после символа поместить звёздочку, это будет означать, что регулярное выражение сработает, если символ появляется в строке любое количество раз — включая и ситуацию, когда символ в строке отсутствует.

```
$ echo "test" | awk '/tes*t/{print $0}'
$ echo "tessst" | awk '/tes*t/{print $0}'
```

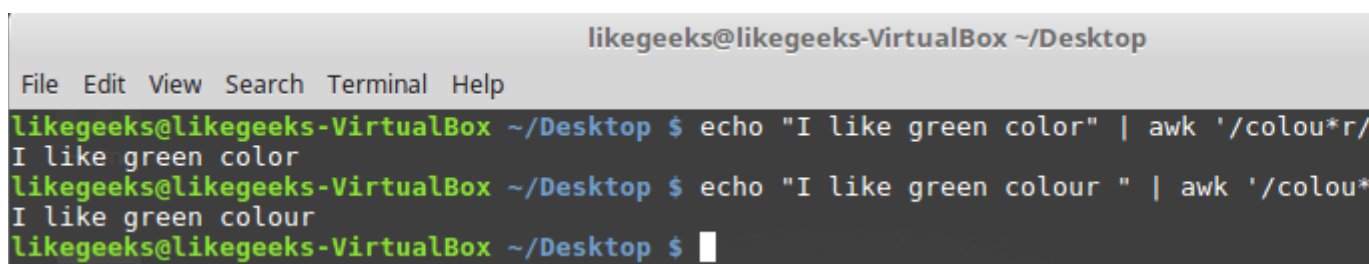


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/tes*t/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tessst" | awk '/tes*t/{print $0}'
tessst
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Использование символа \* в регулярных выражениях

Этот шаблонный символ обычно используют для работы со словами, в которых постоянно встречаются опечатки, или для слов, допускающих разные варианты корректного написания:

```
$ echo "I like green color" | awk '/colou*r/{print $0}'
$ echo "I like green colour " | awk '/colou*r/{print $0}'
```



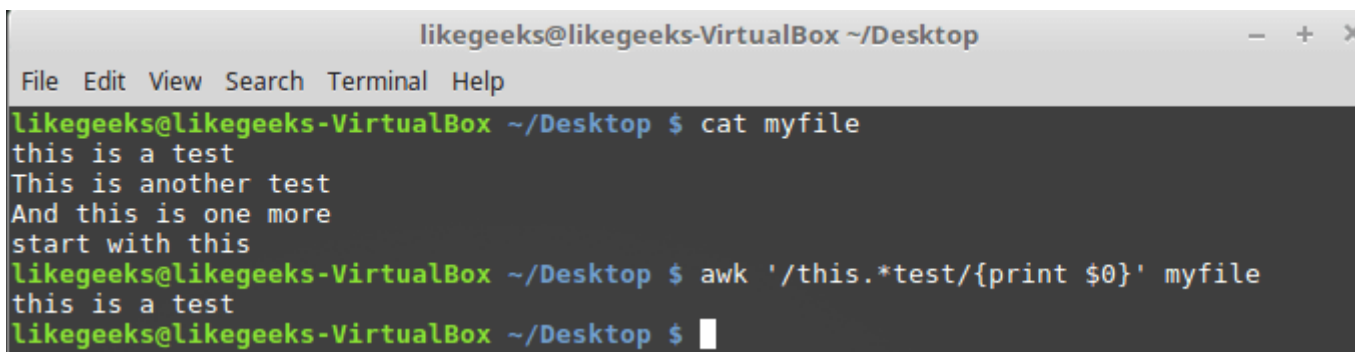
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "I like green color" | awk '/colou*r/'
I like green color
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "I like green colour " | awk '/colou*'
I like green colour
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Поиск слова, имеющего разные варианты написания

В этом примере одно и то же регулярное выражение реагирует и на слово «color», и на слово «colour». Это так благодаря тому, что символ «u», после которого стоит звёздочка, может либо отсутствовать, либо встречаться несколько раз подряд.

Ещё одна полезная возможность, вытекающая из особенностей символа звёздочки, заключается в комбинировании его с точкой. Такая комбинация позволяет регулярному выражению реагировать на любое количество любых символов:

```
$ awk '/this.*test/{print $0}' myfile
```



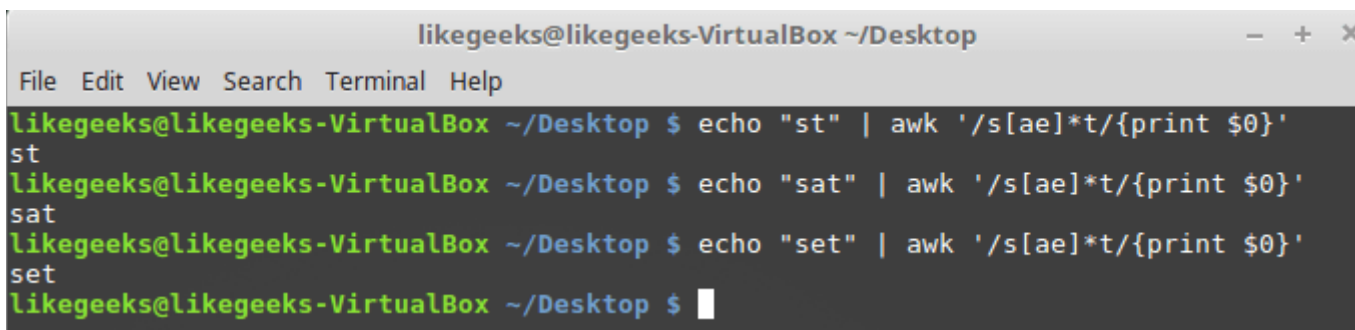
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/this.*test/{print $0}' myfile
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Шаблон, реагирующий на любое количество любых символов*

В данном случае неважно сколько и каких символов находится между словами «this» и «test».

Звёздочку можно использовать и с классами символов:

```
$ echo "st" | awk '/s[ae]*t/{print $0}'
$ echo "sat" | awk '/s[ae]*t/{print $0}'
$ echo "set" | awk '/s[ae]*t/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "st" | awk '/s[ae]*t/{print $0}'
st
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "sat" | awk '/s[ae]*t/{print $0}'
sat
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "set" | awk '/s[ae]*t/{print $0}'
set
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Использование звёздочки с классами символов*

Во всех трёх примерах регулярное выражение срабатывает, так как звёздочка после класса символов означает, что если будет найдено любое количество символов «a» или «e», а также если их найти не удастся, строка будет соответствовать заданному шаблону.

## Регулярные выражения POSIX ERE

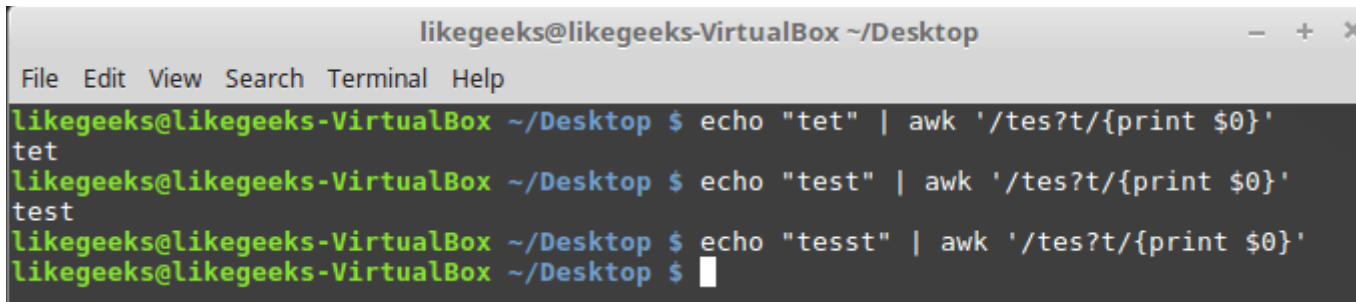
Шаблоны стандарта POSIX ERE, которые поддерживают некоторые утилиты Linux, могут содержать дополнительные символы. Как уже было сказано, awk поддерживает этот стандарт, а вот sed — нет.

Тут мы рассмотрим наиболее часто используемые в ERE-шаблонах символы, которые пригодятся вам при создании собственных регулярных выражений.

## Вопросительный знак

Вопросительный знак указывает на то, что предшествующий символ может встретиться в тексте один раз или не встретиться вовсе. Этот символ — один из метасимволов повторений. Вот несколько примеров:

```
$ echo "tet" | awk '/tes?t/{print $0}'
tet
$ echo "test" | awk '/tes?t/{print $0}'
test
$ echo "tesst" | awk '/tes?t/{print $0}'
```



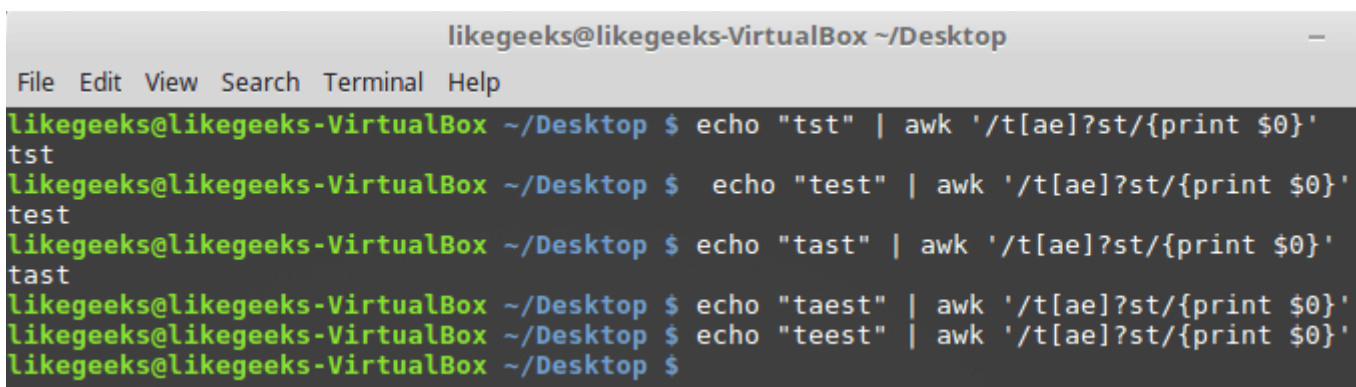
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tet" | awk '/tes?t/{print $0}'
tet
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/tes?t/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tesst" | awk '/tes?t/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Вопросительный знак в регулярных выражениях*

Как видно, в третьем случае буква «s» встречается дважды, поэтому на слово «tesst» регулярное выражение не реагирует.

Вопросительный знак можно использовать и с классами символов:

```
$ echo "tst" | awk '/t[ae]?st/{print $0}'
tst
$ echo "test" | awk '/t[ae]?st/{print $0}'
test
$ echo "tast" | awk '/t[ae]?st/{print $0}'
tast
$ echo "taest" | awk '/t[ae]?st/{print $0}'
$ echo "teest" | awk '/t[ae]?st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/t[ae]?st/{print $0}'
tst
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/t[ae]?st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tast" | awk '/t[ae]?st/{print $0}'
tast
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "taest" | awk '/t[ae]?st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/t[ae]?st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

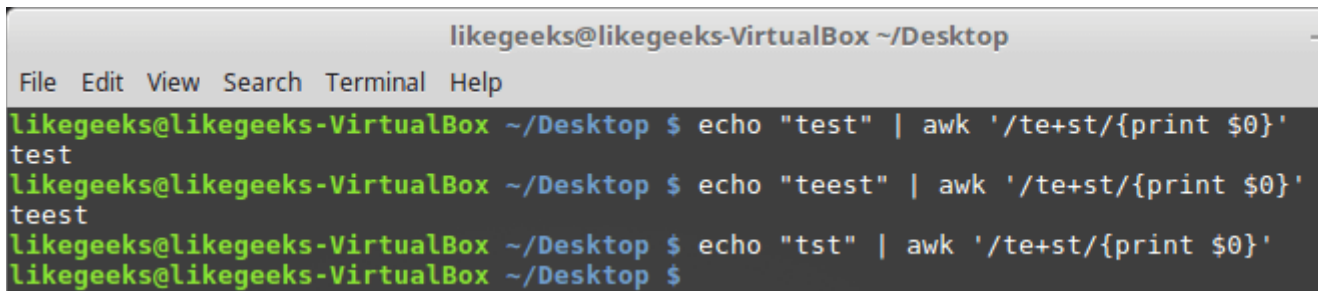
### *Вопросительный знак и классы символов*

Если символов из класса в строке нет, или один из них встречается один раз, регулярное выражение срабатывает, однако стоит в слове появиться двум символам и система уже не находит в тексте соответствия шаблону.

## Символ «плюс»

Символ «плюс» в шаблоне указывает на то, что регулярное выражение обнаружит искомое в том случае, если предшествующий символ встретится в тексте один или более раз. При этом на отсутствие символа такая конструкция реагировать не будет:

```
$ echo "test" | awk '/te+st/{print $0}'
$ echo "teest" | awk '/te+st/{print $0}'
$ echo "tst" | awk '/te+st/{print $0}'
```

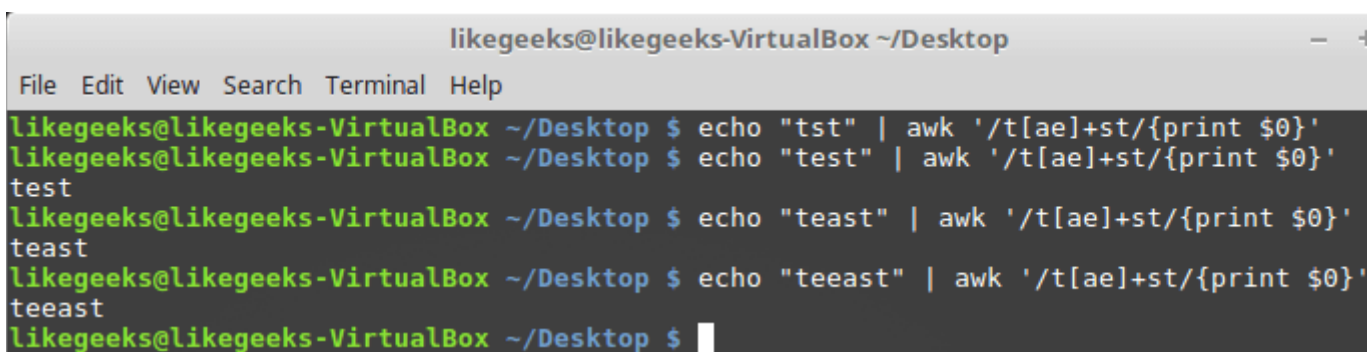


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/te+st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/te+st/{print $0}'
teest
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/te+st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Символ «плюс» в регулярных выражениях

В данном примере, если символа «е» в слове нет, движок регулярных выражений не найдёт в тексте соответствий шаблону. Символ «плюс» работает и с классами символов — этим он похож на звёздочку и вопросительный знак:

```
$ echo "tst" | awk '/t[ae]+st/{print $0}'
$ echo "test" | awk '/t[ae]+st/{print $0}'
$ echo "teast" | awk '/t[ae]+st/{print $0}'
$ echo "teeast" | awk '/t[ae]+st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/t[ae]+st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/t[ae]+st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teast" | awk '/t[ae]+st/{print $0}'
teast
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teeast" | awk '/t[ae]+st/{print $0}'
teeast
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Знак «плюс» и классы символов

В данном случае если в строке имеется любой символ из класса, текст будет сочтён соответствующим шаблону.

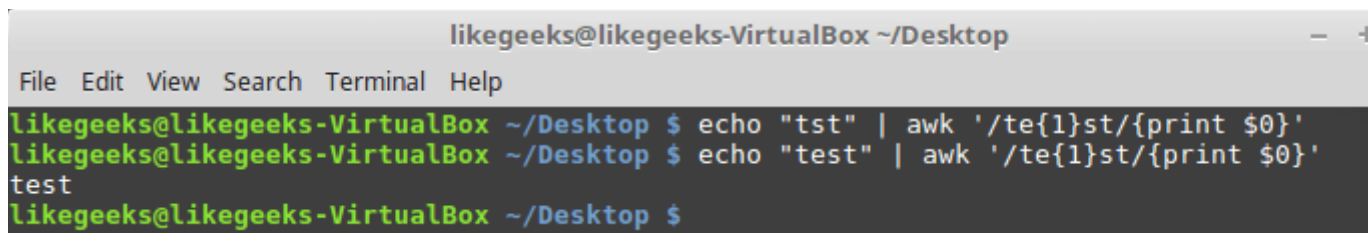
## Фигурные скобки

Фигурные скобки, которыми можно пользоваться в ERE-шаблонах, похожи на символы, рассмотренные выше, но они позволяют точнее задавать необходимое число вхождений предшествующего им символа. Указывать ограничение можно в двух форматах:

- $n$  — число, задающее точное число искомых вхождений
- $n, m$  — два числа, которые трактуются так: «как минимум  $n$  раз, но не больше чем  $m$ ».

Вот примеры первого варианта:

```
$ echo "tst" | awk '/te{1}st/{print $0}'
$ echo "test" | awk '/te{1}st/{print $0}'
```

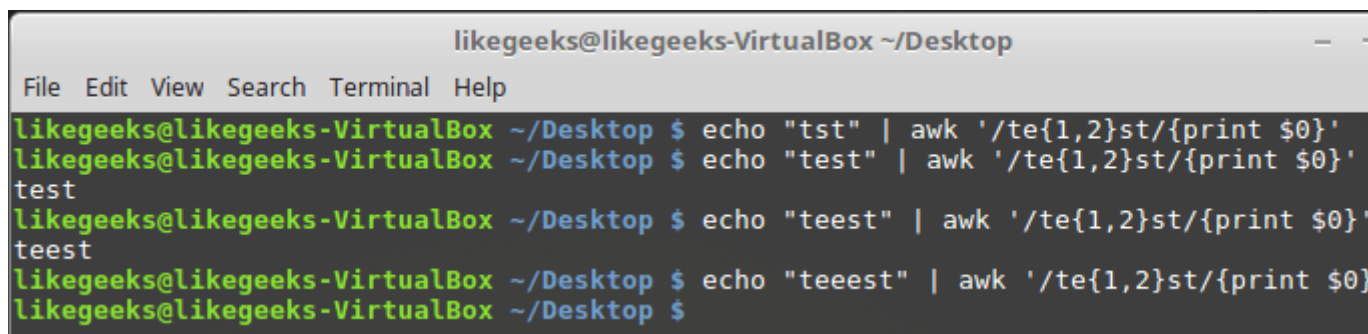


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/te{1}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/te{1}st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Фигурные скобки в шаблонах, поиск точного числа вхождений*

В старых версиях `awk` нужно было использовать ключ командной строки `--re-interval` для того, чтобы программа распознавала интервалы в регулярных выражениях, но в новых версиях этого делать не нужно.

```
$ echo "tst" | awk '/te{1,2}st/{print $0}'
$ echo "test" | awk '/te{1,2}st/{print $0}'
$ echo "teest" | awk '/te{1,2}st/{print $0}'
$ echo "teeest" | awk '/te{1,2}st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/te{1,2}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/te{1,2}st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/te{1,2}st/{print $0}'
teest
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teeest" | awk '/te{1,2}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

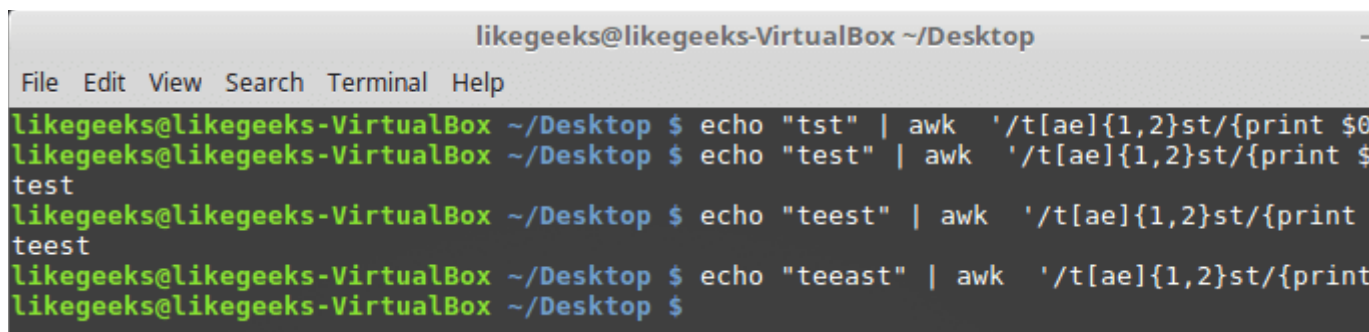
*Интервал, заданный в фигурных скобках*

В данном примере символ «е» должен встретиться в строке 1 или 2 раза, тогда регулярное выражение отреагирует на текст.



Фигурные скобки можно применять и с классами символов. Тут действуют уже знакомые вам принципы:

```
$ echo "tst" | awk '/t[ae]{1,2}st/{print $0}'
$ echo "test" | awk '/t[ae]{1,2}st/{print $0}'
$ echo "teest" | awk '/t[ae]{1,2}st/{print $0}'
$ echo "teeast" | awk '/t[ae]{1,2}st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/t[ae]{1,2}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/t[ae]{1,2}st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/t[ae]{1,2}st/{print $0}'
teest
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teeast" | awk '/t[ae]{1,2}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

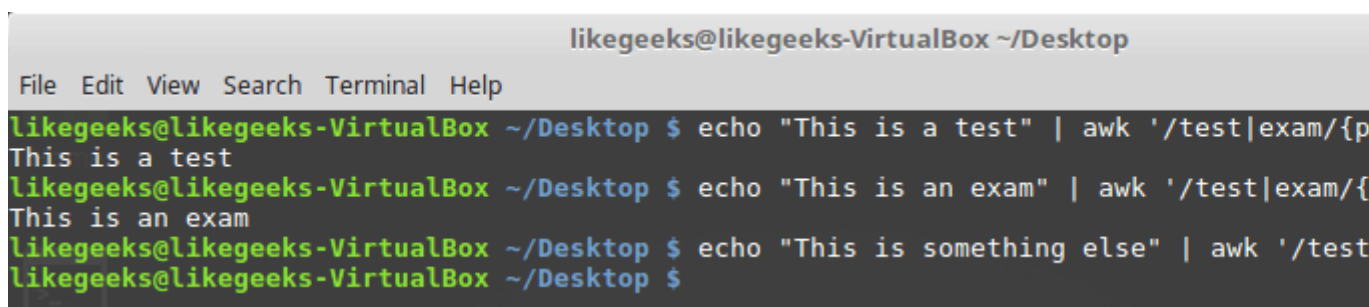
### Фигурные скобки и классы символов

Шаблон отреагирует на текст в том случае, если в нём один или два раза встретится символ «а» или символ «е».

## Символ логического «или»

Символ | — вертикальная черта, означает в регулярных выражениях логическое «или». Обработывая регулярное выражение, содержащее несколько фрагментов, разделённых таким знаком, движок сочтёт анализируемый текст подходящим в том случае, если он будет соответствовать любому из фрагментов. Вот пример:

```
$ echo "This is a test" | awk '/test|exam/{print $0}'
$ echo "This is an exam" | awk '/test|exam/{print $0}'
$ echo "This is something else" | awk '/test|exam/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/test|exam/{print $0}'
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is an exam" | awk '/test|exam/{print $0}'
This is an exam
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is something else" | awk '/test|exam/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

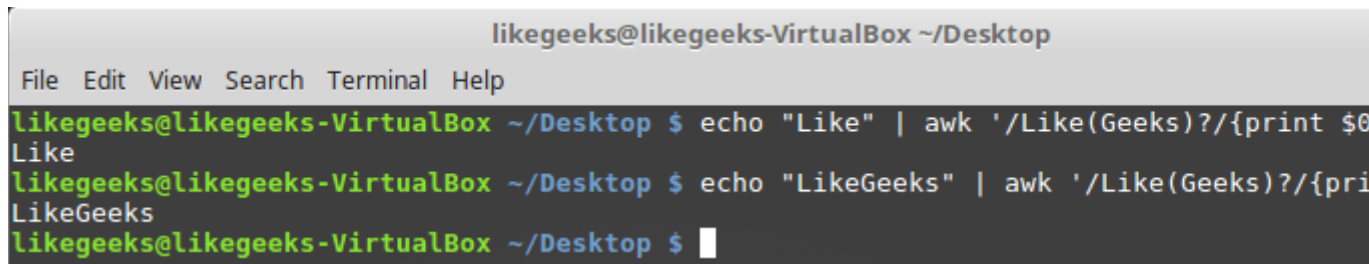
### Логическое «или» в регулярных выражениях

В данном примере регулярное выражение настроено на поиск в тексте слов «test» или «exam». Обратите внимание на то, что между фрагментами шаблона и разделяющим их символом | не должно быть пробелов.

## Группировка фрагментов регулярных выражений

Фрагменты регулярных выражений можно группировать, пользуясь круглыми скобками. Если сгруппировать некую последовательность символов, она будет восприниматься системой как обычный символ. То есть, например, к ней можно будет применить метасимволы повторений. Вот как это выглядит:

```
$ echo "Like" | awk '/Like(Geeks)?/{print $0}'
$ echo "LikeGeeks" | awk '/Like(Geeks)?/{print $0}'
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows two commands being executed. The first command is 'echo "Like" | awk "/Like(Geeks)?/{print \$0}"', which outputs 'Like'. The second command is 'echo "LikeGeeks" | awk "/Like(Geeks)?/{print \$0}"', which outputs 'LikeGeeks'. The prompt '\$' is visible at the end of each line.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "Like" | awk '/Like(Geeks)?/{print $0}'
Like
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "LikeGeeks" | awk '/Like(Geeks)?/{print $0}'
LikeGeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Группировка фрагментов регулярных выражений*

В данных примерах слово «Geeks» заключено в круглые скобки, после этой конструкции идёт знак вопроса. Напомним, что вопросительный знак означает «0 или 1 повторение», в результате регулярное выражение отреагирует и на строку «Like», и на строку «LikeGeeks».

## Практические примеры

После того, как мы разобрали основы регулярных выражений, пришло время сделать с их помощью что-нибудь полезное.

### Подсчёт количества файлов

Напишем bash-скрипт, который подсчитывает файлы, находящиеся в директориях, которые записаны в переменную окружения PATH. Для того, чтобы это сделать, понадобится, для начала, сформировать список путей к директориям. Сделаем это с помощью sed, заменив двоеточия на пробелы:

```
$ echo $PATH | sed 's:/:/g'
```

Команда замены поддерживает регулярные выражения в качестве шаблонов для поиска текста. В данном случае всё предельно просто, ищем мы символ двоеточия, но никто не мешает использовать здесь и что-нибудь другое — всё зависит от конкретной задачи.

Теперь надо пройти по полученному списку в цикле и выполнить там необходимые для подсчёта количества файлов действия. Общая схема скрипта будет такой:

```
mypath=$(echo $PATH | sed 's:/:/g')
```

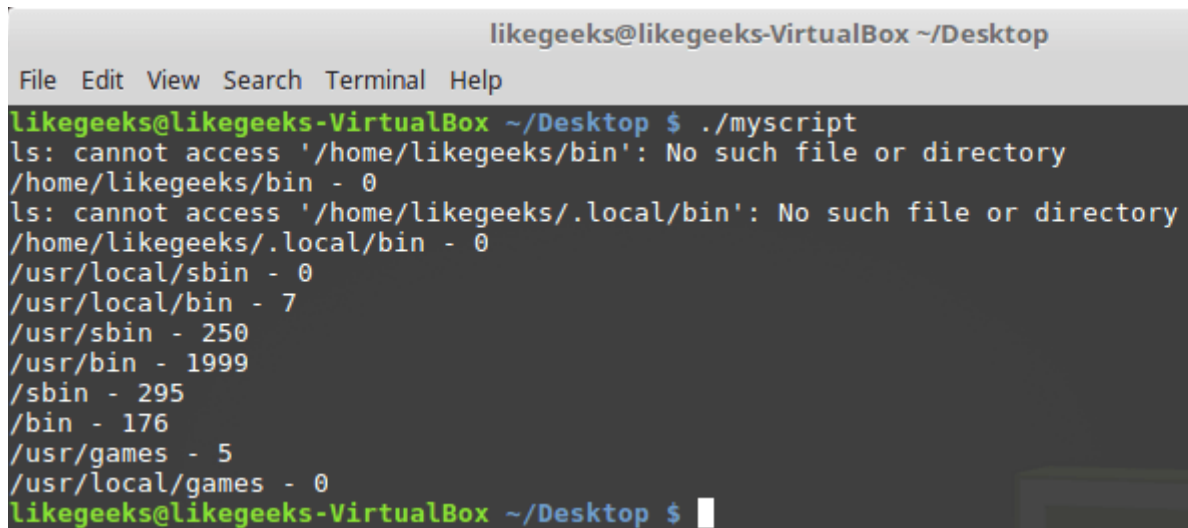


```
for directory in $mypath
do
done
```

Теперь напишем полный текст скрипта, воспользовавшись командой `ls` для получения сведений о количестве файлов в каждой из директорий:

```
#!/bin/bash
mypath=$(echo $PATH | sed 's:/:/g')
count=0
for directory in $mypath
do
check=$(ls $directory)
for item in $check
do
count=$((count + 1))
done
echo "$directory - $count"
count=0
done
```

При запуске скрипта может оказаться, что некоторых директорий из `PATH` не существует, однако, это не мешает ему посчитать файлы в существующих директориях.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named './myscript'. The script iterates through the directories in the PATH variable and counts the number of files in each. The output is as follows:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
ls: cannot access '/home/likegeeks/bin': No such file or directory
/home/likegeeks/bin - 0
ls: cannot access '/home/likegeeks/.local/bin': No such file or directory
/home/likegeeks/.local/bin - 0
/usr/local/sbin - 0
/usr/local/bin - 7
/usr/sbin - 250
/usr/bin - 1999
/sbin - 295
/bin - 176
/usr/games - 5
/usr/local/games - 0
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Подсчёт файлов*

Главная ценность этого примера заключается в том, что пользуясь тем же подходом, можно решать и куда более сложные задачи. Какие именно — зависит от ваших потребностей.

## **Проверка адресов электронной почты**

Существуют веб-сайты с огромными коллекциями регулярных выражений, которые позволяют проверять адреса электронной почты, телефонные

номера, и так далее. Однако, одно дело — взять готовое, и совсем другое — создать что-то самому. Поэтому напишем регулярное выражение для проверки адресов электронной почты. Начнём с анализа исходных данных. Вот, например, некий адрес:

```
username@hostname.com
```

Имя пользователя, `username`, может состоять из алфавитно-цифровых и некоторых других символов. А именно, это точка, тире, символ подчёркивания, знак «плюс». За именем пользователя следует знак `@`.

Вооружившись этими знаниями, начнём сборку регулярного выражения с его левой части, которая служит для проверки имени пользователя. Вот что у нас получилось:

```
^([a-zA-Z0-9_-\.\+])@
```

Это регулярное выражение можно прочитать так: «В начале строки должен быть как минимум один символ из тех, которые имеются в группе, заданной в квадратных скобках, а после этого должен идти знак `@`».

Теперь — очередь имени хоста — `hostname`. Тут применимы те же правила, что и для имени пользователя, поэтому шаблон для него будет выглядеть так:

```
([a-zA-Z0-9_-\.\+])
```

Имя домена верхнего уровня подчиняется особым правилам. Тут могут быть лишь алфавитные символы, которых должно быть не меньше двух (например, такие домены обычно содержат код страны), и не больше пяти. Всё это значит, что шаблон для проверки последней части адреса будет таким:

```
\.([a-zA-Z]{2,5})$
```

Прочитать его можно так: «Сначала должна быть точка, потом — от 2 до 5 алфавитных символов, а после этого строка заканчивается».

Подготовив шаблоны для отдельных частей регулярного выражения, соберём их вместе:

```
^([a-zA-Z0-9_-\.\+])@([a-zA-Z0-9_-\.\+])\.[a-zA-Z]{2,5}$
```

Теперь осталось лишь протестировать то, что получилось:

```
$ echo "name@host.com" | awk '/^([a-zA-Z0-9_-\.\+])@([a-zA-Z0-9_-\.\+])\.[a-zA-Z]{2,5})$/ {print $0}'
$ echo "name@host.com.us" | awk '/^([a-zA-Z0-9_-\.\+])@([a-zA-Z0-9_-\.\+])\.[a-zA-Z]{2,5})$/ {print $0}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "name@host.com" | awk '/^([a-zA-Z0-9_+)]\.[a-zA-Z]{2,5})$/ {print $0}'
name@host.com
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "name@host.com.us" | awk '/^([a-zA-Z0-9_+)]\.[a-zA-Z]{2,5})$/ {print $0}'
name@host.com.us
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Проверка адреса электронной почты с помощью регулярных выражений*

То, что переданный awk текст выводится на экран, означает, что система распознала в нём адрес электронной почты.

### Часть 3. Bash-скрипты

Данная часть работы посвящена написанию bash-скриптов.

По сути своей Bash-скрипты представляют из себя ни что иное как последовательность команд командной строки, объединенных в один файл для решения какой-либо задачи

Любой скрипт начитается с так называемой шебанг строки, которая указывает на расположение исполняемого файла той оболочки, для которой мы хотим написать скрипт. В нашем случае это bash, следовательно данная строка будет выглядеть следующим образом:

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

После чего уже следует непосредственно сам скрипт.

Символом решетки в bash-скриптах обозначаются комментарии.

Напишем самый базовый скрипт, который просто будет выводить текущий каталог. Создадим файл **myscript** и запишем в него следующий код.

```
#!/bin/bash  
# This is a comment  
pwd
```

Запуск скриптов из командной строки осуществляется следующим образом. В директории с файлом скрипта необходимо написать следующую команду **./myscript**.

Однако просто так этого сделать не получится, система выдаст ошибку, необходимо выдать этому файлу права на выполнение.

```
chmod +x ./myscript
```

После чего скрипт может быть запущен.

Для вывода текста в скриптах можно воспользоваться уже известной командой **echo**.

## Переменные

Как и любой другой язык `bash` обладает возможностью создания переменных.

Переменные бывают как пользовательскими, создаваемыми при работе скрипта, так и переменными среды, созданными для хранения параметров ОС.

Для обращения к переменным среды следует использовать следующую конструкцию: **\$ИМЯ\_ПЕРЕМЕННОЙ**.

К примеру, выведем в сообщении путь к домашней директории текущего пользователя.

```
#!/bin/bash
# display user home
echo "Home for the current user is: $HOME"
```

Заметьте, что строковый литерал — двойные кавычки — никак не повлиял на распознавание переменной среды.

Пользовательские переменные в `bash`-скриптах имеют динамический тип. Вот пример их задания с последующим выводом.

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

В данном случае было создано 2 переменные: **grade** и **person**.

## Подстановка команд

Одна из самых полезных возможностей `bash`-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами.

- С помощью значка обратного апострофа «`»
- С помощью конструкции `$()`

Используя первый подход, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка:

```
mydir=`pwd`
```

При втором подходе то же самое записывают так:

```
mydir=$(pwd)
```

А скрипт, в итоге, может выглядеть так:

```
#!/bin/bash  
mydir=$(pwd)  
echo $mydir
```

В ходе его работы вывод команды `pwd` будет сохранён в переменной `mydir`, содержимое которой, с помощью команды `echo`, попадёт в консоль.

### Математические операции

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида `$(( a + b ))`:

```
#!/bin/bash  
var1=$(( 5 + 5 ))  
echo $var1  
var2=$(( $var1 * 2 ))  
echo $var2  
var3=$(( $var2 / 4 ))  
echo $var3  
var4=$(( $var3 % 5 ))  
echo $var4  
var5=$(( $var4 ** 2 ))  
echo $var5
```

### Управляющая конструкция if-then-else

Для управления потоком исполнения команд в `bash`-скриптах можно использовать управляющую конструкцию `if-then-else`. Работает она как во всех языках программирования, однако имеет свои особенности. Вот её подробный синтаксис:

```
if команда1
then
команды
elif команда2
then
команды
else
команды
fi
```

В отличие от большинства известных вам на данный момент языков bash-скрипты обладают возможностью задать дополнительные условия в конструкции **if-else** при помощи ключевого слова **elif**. Пример скрипта с **if-then-else**.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
else
echo "The user $user doesn't exist"
fi
```

Пример скрипта с **elif**.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
elif ls /home
then echo "The user doesn't exist but anyway there
is a directory under /home"
fi
```

Команда `grep` используется для поиска информации о пользователе, чье имя записано в переменной **user**, в файле `/etc/passwd`, который содержит информацию обо всех пользователях в системе.

### Сравнение чисел

В отличие от привычных операций сравнения в языках программирования bash-скрипты обладают специфическим синтаксисом

операций сравнения. Например, для чисел применяются следующие операции:

- `n1 -eq n2` – Возвращает истинное значение, если `n1` равно `n2`.
- `n1 -ge n2` – Возвращает истинное значение, если `n1` больше или равно `n2`.
- `n1 -gt n2` – Возвращает истинное значение, если `n1` больше `n2`.
- `n1 -le n2` – Возвращает истинное значение, если `n1` меньше или равно `n2`.
- `n1 -lt n2` – Возвращает истинное значение, если `n1` меньше `n2`.
- `n1 -ne n2` – Возвращает истинное значение, если `n1` не равно `n2`.

При использовании подобных операций выражения необходимо заключать в квадратные скобки, например:

```
#!/bin/bash
val1=6
if [ $val1 -gt 5 ]
then
echo "The test value $val1 is greater than 5"
else
echo "The test value $val1 is not greater than 5"
fi
```

## Сравнение строк

Для сравнения строк используются чуть более привычные операторы, однако и они не лишены новшеств.

- `str1 = str2` – Проверяет строки на равенство, возвращает истину, если строки идентичны.
- `str1 != str2` – Возвращает истину, если строки не идентичны.
- `str1 < str2` – Возвращает истину, если `str1` меньше, чем `str2`.
- `str1 > str2` – Возвращает истину, если `str1` больше, чем `str2`.
- `-n str1` – Возвращает истину, если длина `str1` больше нуля.
- `-z str1` – Возвращает истину, если длина `str1` равна нулю.

Пример сравнения строк

```
#!/bin/bash
user="likegeeks"
if [$user = $USER]
then
echo "The user $user is the current logged in user"
fi
```



При работе с операторами < и > есть несколько особенностей, которые стоит учитывать. Во-первых, их необходимо экранировать при помощи символа \. Во-вторых, необходимо заключать имена переменных в двойные кавычки "\$val2". Приведем пример работы с подобным оператором.

```
#!/bin/bash
val1=text
val2="another text"
if [ "$val1" \> "$val2" ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

## Проверки файлов

Наиболее используемыми командами для bash скриптов являются команды проверки файлов.

-d file – Проверяет, существует ли файл, и является ли он директорией.

-e file – Проверяет, существует ли файл.

-f file – Проверяет, существует ли файл, и является ли он файлом.

-r file – Проверяет, существует ли файл, и доступен ли он для чтения.

-s file – Проверяет, существует ли файл, и не является ли он пустым.

-w file – Проверяет, существует ли файл, и доступен ли он для записи.

-x file – Проверяет, существует ли файл, и является ли он исполняемым.

file1 -nt file2 – Проверяет, новее ли file1, чем file2.

file1 -ot file2 – Проверяет, старше ли file1, чем file2.

-O file – Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.

-G file – Проверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Пример скрипта, в котором используются эти команды.

```
#!/bin/bash
mydir=/home/likegeeks
if [ -d $mydir ]
then
```

```
echo "The $mydir directory exists"
cd $mydir
ls
else
echo "The $mydir directory does not exist"
fi
```

## Операторы цикла

Bash-скрипты поддерживают несколько вариантов циклов для перебора последовательностей значений. В число этих циклов входят:

- for;
- while.

Для начала рассмотрим цикл **for**. Базовая структура такого цикла выглядит следующим образом.

```
for var in list
do
команды
done
```

Самый простой пример – перебор списка простых значений.

```
#!/bin/bash
for var in first second third fourth fifth
do
echo The $var item
done
```

Для перебора сложных значений необходимо заключать эти значения в строковые литералы – двойные кавычки, к примеру:

```
#!/bin/bash
for var in first "the second" "the third" "I'll do it"
do
echo "This is: $var"
done
```

Список для цикла `for` может быть получен из результата выполнения какой-либо команды, полученного при помощи рассмотренной ранее подстановки команд. Как пример – перебор вывода команды `cat`.

```
#!/bin/bash
file="myfile"
for var in $(cat $file)
do
echo " $var"
done
```

Однако проблема такого вывода в том, что файл будет обрабатываться по словам, а не по строкам, как того хотелось бы.

Это связано с тем, что оболочка `bash` считает разделителем строки следующий набор символов:

- Пробел;
- Знак табуляции;
- Знак перевода строки.

Для того, чтобы задать разделитель необходимо использовать переменную окружения `IFS` (Internal Field Separator). Пример того, как можно выполнить итерацию лишь по строкам файла.

```
#!/bin/bash
file="/etc/passwd"
IFS=$'\n'
for var in $(cat $file)
do
echo " $var"
done
```

По мере работы скрипта возможно изменение переменной `IFS`, так что она может задаваться в зависимости от контекста.

Для обхода файлов, находящихся в директориях, также используется цикл `for`. Вот пример такого обхода с выводом списка файлов и директорий.

```
#!/bin/bash
for file in /home/<username>/*
do
if [ -d "$file" ]
```

```
then
echo "$file is a directory"
elif [ -f "$file" ]
then
echo "$file is a file"
fi
done
```

Как можно видеть из примера для обхода файлов можно использовать следующий путь, который подразумевает сбор всех файлов и директорий - **/home/<username>/\***

Вторая разновидность циклов – цикл while.

```
while команда проверки условия
do
другие команды
done
```

Пример скрипта с таким циклом.

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
do
echo $var1 var1=$(( $var1 - 1 ))
done
```

Для управления циклами также могут применяться привычные команды операторы **break** и **continue**.

Результат работы цикла также могут быть выведены в файл при помощи перенаправления вывода. Вот пример такого скрипта.

```
#!/bin/bash
for (( a = 1; a < 10; a++ ))
do
echo "Number is $a"
done > myfile.txt
echo "finished."
```