

Отчёт по лабораторной работе №10

Программирование в командном процессоре ОС UNIX. Командные файлы

Митичкина Екатерина Павловна

Содержание

Цель работы	2
Задача	2
Теоретическое введение:	2
Командные процессоры (оболочки)	2
Переменные в языке программирования bash	3
Использование арифметических вычислений. Операторы let и read	4
Метасимволы и их экранирование	7
Командные файлы и функции	8
Передача параметров в командные файлы и специальные переменные	9
Использование команды getopts	10
Управление последовательностью действий в командных файлах	11
Оператор цикла for	12
Оператор выбора case	13
Условный оператор if	14
Операторы цикла while и until	15
Прерывание циклов	16
Выполнение лабораторной работы	17
Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор zip, bzip2 или tar. Способ использования команд архивации необходимо узнать, изучив справку.	17
Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.	21
Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном	

каталоге и выводил информацию о возможностях доступа к файлам этого каталога.	23
Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки.	24
Выводы	25
Ответы на контрольные вопросы	25

Цель работы

Изучить основы программирования в оболочке ОС UNIX/Linux. Научиться писать небольшие командные файлы.

Задача

1. Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор zip, bzip2 или tar. Способ использования команд архивации необходимо узнать, изучив справку.
2. Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.
3. Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.
4. Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки.

Теоретическое введение:

Командные процессоры (оболочки)

Командный процессор (командная оболочка, интерпретатор команд shell) — это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера. В операционных системах типа UNIX/Linux наиболее часто

используются следующие реализации командных оболочек: - оболочка Борна (Bourne shell или sh) — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций; - C-оболочка (или csh) — надстройка на оболочке Борна, использующая C-подобный синтаксис команд с возможностью сохранения истории выполнения команд; - оболочка Корна (или ksh) — напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна; - BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation). POSIX (Portable Operating System Interface for Computer Environments) — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ.

Стандарты POSIX разработаны комитетом IEEE (Institute of Electrical and Electronics Engineers) для обеспечения совместимости различных UNIX/Linux-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. POSIX-совместимые оболочки разработаны на базе оболочки Корна.

Рассмотрим основные элементы программирования в оболочке bash. В других оболочках большинство команд будет совпадать с описанными ниже.

Переменные в языке программирования bash

Командный процессор bash обеспечивает возможность использования переменных типа строка символов. Имена переменных могут быть выбраны пользователем. Пользователь имеет возможность присвоить переменной значение некоторой строки символов.

Например, команда

```
mark=/usr/andy/bin
```

присваивает значение строки символов /usr/andy/bin переменной mark типа строка символов.

Значение, присвоенное некоторой переменной, может быть впоследствии использовано. Для этого в соответствующем месте командной строки должно быть употреблено имя этой переменной, которому предшествует метасимвол \$.

Например, команда

```
mv afile ${mark}
```

переместит файл afile из текущего каталога в каталог с абсолютным полным именем /usr/andy/bin.

Использование значения, присвоенного некоторой переменной, называется подстановкой. Для того чтобы имя переменной не сливалось с символами, которые могут следовать за ним в командной строке, при подстановке в общем случае используется следующая форма записи:

```
${имя переменной}
```

Например, использование команд

```
b=/tmp/andy-  
ls -l myfile > ${b}lsudo apt-get install texlive-luatex
```

приведёт к переназначению стандартного вывода команды `ls` с терминала на файл `/tmp/andy-ls`, а использование команды `ls -l>$bls` приведёт к подстановке в командную строку значения переменной `bls`. Если переменной `bls` не было предварительно присвоено никакого значения, то её значением будет символ пробела.

Оболочка `bash` позволяет работать с массивами. Для создания массива используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделённых пробелами.

Например,

```
set -A states Delaware Michigan "New Jersey"
```

Далее можно сделать добавление в массив, например, `states[49]=Alaska`. Индексация массивов начинается с нулевого элемента.

Использование арифметических вычислений. Операторы `let` и `read`

Оболочка `bash` поддерживает встроенные арифметические функции. Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Простейшее выражение — это единичный терм (`term`), обычно целочисленный.

Целые числа можно записывать как последовательность цифр или в любом базовом формате типа `radix#number`, где `radix` (основание системы счисления) — любое число не более 26. Для большинства команд используются следующие основания систем исчисления: 2 (двоичная), 8 (восьмеричная) и 16 (шестнадцатеричная). Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (*), целочисленное деление (/) и целочисленный остаток от деления (%).

Команда `let` берет два операнда и присваивает их переменной. Положительным моментом команды `let` можно считать то, что для идентификации переменной ей не нужен знак доллара; вы можете писать команды типа `let sum=x+7`, и `let` будет искать переменную `x` и добавлять к ней 7.

Команда `let` также расширяет другие выражения `let`, если они заключены в двойные круглые скобки. Таким способом вы можете создавать довольно сложные выражения. Команда `let` не ограничена простыми арифметическими выражениями. Табл. показывает полный набор `let`-операций.

Подобно `C` оболочка `bash` может присваивать переменной любое значение, а произвольное выражение само имеет значение, которое может использоваться. При этом «ноль» воспринимается как «ложь», а любое другое значение выражения — как

«истина». Для облегчения программирования можно записывать условия оболочки `bash` в двойные скобки — `(())`.

Арифметические операторы оболочки `bash`

Оператор	Синтаксис	Результат
<code>!</code>	<code>!exp</code>	Если <code>exp</code> равно 0, то возвращает 1; иначе 0
<code>!=</code>	<code>exp1 !=exp2</code>	Если <code>exp1</code> не равно <code>exp2</code> , то возвращает 1; иначе 0
<code>%</code>	<code>exp1%exp2</code>	Возвращает остаток от деления <code>exp1</code> на <code>exp2</code>
<code>%=</code>	<code>var=%exp</code>	Присваивает остаток от деления <code>var</code> на <code>exp</code> переменной <code>var</code>
<code>&</code>	<code>exp1&exp2</code>	Возвращает побитовое AND выражений <code>exp1</code> и <code>exp2</code>
<code>&&</code>	<code>exp1&&exp2</code>	Если и <code>exp1</code> и <code>exp2</code> не равны нулю, то возвращает 1; иначе 0
<code>&=</code>	<code>var &= exp</code>	Присваивает переменной <code>var</code> побитовое AND <code>var</code> и <code>exp</code>
<code>*</code>	<code>exp1 * exp2</code>	Умножает <code>exp1</code> на <code>exp2</code>
<code>*=</code>	<code>var *= exp</code>	Умножает <code>exp</code> на значение переменной <code>var</code> и присваивает результат переменной <code>var</code>
<code>+</code>	<code>exp1 + exp2</code>	Складывает <code>exp1</code> и <code>exp2</code>
<code>+=</code>	<code>var += exp</code>	Складывает <code>exp</code> со значением переменной <code>var</code> и результат присваивает переменной <code>var</code>
<code>-</code>	<code>-exp</code>	Операция отрицания <code>exp</code> (унарный минус)
<code>-</code>	<code>exp1 - exp2</code>	Вычитает <code>exp2</code> из <code>exp1</code>
<code>-=</code>	<code>var -= exp</code>	Вычитает <code>exp</code> из значения переменной <code>var</code> и присваивает результат переменной <code>var</code>
<code>/</code>	<code>exp / exp2</code>	Делит <code>exp1</code> на <code>exp2</code>
<code>/=</code>	<code>var /= exp</code>	Делит значение переменной <code>var</code> на <code>exp</code> и присваивает результат переменной <code>var</code>
<code><</code>	<code>exp1 < exp2</code>	Если <code>exp1</code> меньше, чем <code>exp2</code> , то возвращает 1, иначе возвращает 0
<code><<</code>	<code>exp1 << exp2</code>	Сдвигает <code>exp1</code> влево на <code>exp2</code> бит
<code><<=</code>	<code>var <<= exp</code>	Побитовый сдвиг влево значения переменной <code>var</code> на <code>exp</code>
<code><=</code>	<code>exp1 <= exp2</code>	Если <code>exp1</code> меньше или равно <code>exp2</code> , то возвращает 1; иначе возвращает 0
<code>=</code>	<code>var = exp</code>	Присваивает значение <code>exp</code> переменной <code>var</code>
<code>==</code>	<code>exp1==exp2</code>	Если <code>exp1</code> равно <code>exp2</code> , то возвращает 1; иначе возвращает 0
<code>></code>	<code>exp1 > exp2</code>	1, если <code>exp1</code> больше, чем <code>exp2</code> ; иначе 0
<code>>=</code>	<code>exp1 >= exp2</code>	1, если <code>exp1</code> больше или равно <code>exp2</code> ; иначе 0

Оператор	Синтаксис	Результат
>>	exp >> exp2	Сдвигает exp1 вправо на exp2 бит
>>=	var >>=exp	Побитовый сдвиг вправо значения переменной var на exp
^	exp1 ^ exp2	Исключающее OR выражений exp1 и exp2
^=	var ^= exp	Присваивает переменной var побитовое XOR var и exp
	exp1 exp2	Побитовое OR выражений exp1 и exp2
=	var = exp	Присваивает переменной var результат операции XOR var и exp
	exp1 exp2	1, если или exp1 или exp2 являются ненулевыми значениями; иначе 0
~	~exp	Побитовое дополнение до exp

Можно присваивать результаты условных выражений переменным, также как и использовать результаты арифметических вычислений в качестве условий. Хорошим примером сказанного является выполнение некоторого действия, одновременно декрементируя некоторое значение. например:

```
$ let x=5
$ while
> (( x--=1 ))
> do
> something
> done
```

Этот пример показывает выполнение некоторого действия с начальным значением 5, которое декрементирует до тех пор, пока оно не будет равно нулю. При каждой итерации выполняется функция something.

Наиболее распространённым является сокращение, избавляющееся от слова let в программах оболочек. Если объявить переменные целыми значениями, то любое присвоение автоматически будет трактоваться как арифметическое действие. Если использовать typeset -i для объявления и присвоения переменной, то при последующем её применении она станет целой. Также можно использовать ключевое слово integer (псевдоним для typeset -i) и объявлять таким образом переменные целыми. Выражения типа x=u+z будет восприниматься в это случае как арифметические.

Команда read позволяет читать значения переменных со стандартного ввода:

```
echo "Please enter Month and Day of Birth ?" read mon day trash
```

В переменные mon и day будут считаны соответствующие значения, введённые с клавиатуры, а переменная trash нужна для того, чтобы отобрать всю избыточно введённую информацию и игнорировать её.

Изъять переменную из программы можно с помощью команды unset.

Имена некоторых переменных имеют для командного процессора специальный смысл. Значением переменной PATH (т.е. \$PATH) является список каталогов, в которых командный процессор осуществляет поиск программы или команды, указанной в командной строке, в том случае, если указанное имя программы или команды не содержит ни одного символа /. Если имя команды содержит хотя бы один символ /, то последовательность поиска, предписываемая значением переменной PATH, нарушается. В этом случае в зависимости от того, является имя команды абсолютным или относительным, поиск начинается соответственно от корневого или текущего каталога.

Если Вы сами явно не присвоите переменной PATH какое-либо значение, то стандартной (по умолчанию) последовательностью поиска файла является следующая: текущий каталог, каталог /bin, каталог /usr/bin. Именно в такой последовательности командный процессор ищет файлы, содержащие программы, которые обеспечивают выполнение таких, например, команд, как echo, ls и cat.

В списке каталогов, являющемся значением переменной PATH, имена каталогов отделяются друг от друга с помощью символа двоеточия. В качестве примера приведём команду:

```
PATH=~:/bin:/usr/local/bin/~/bin:/usr/bin
```

Переменные PS1 и PS2 предназначены для отображения промптера командного процессора. PS1 — это промптер командного процессора, по умолчанию его значение равно символу \$ или #. Если какая-то интерактивная программа, запущенная командным процессором, требует ввода, то используется промптер PS2. Он по умолчанию имеет значение символа >.

Другие стандартные переменные: - HOME — имя домашнего каталога пользователя. Если команда cd вводится без аргументов, то происходит переход в каталог, указанный в этой переменной. - IFS — последовательность символов, являющихся разделителями в командной строке, например, пробел, табуляция и перевод строки (new line). - MAIL — командный процессор каждый раз перед выводом на экран промптера проверяет содержимое файла, имя которого указано в этой переменной, и если содержимое этого файла изменилось с момента последнего ввода из него, то перед тем как вывести на терминал промптер, командный процессор выводит на терминал сообщение You have mail (у Вас есть почта). - TERM — тип используемого терминала. - LOGNAME — содержит регистрационное имя пользователя, которое устанавливается автоматически при входе в систему.

В командном процессоре Си имеется ещё несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды set.

Метасимволы и их экранирование

При перечислении имён файлов текущего каталога можно использовать следующие символы: - * — соответствует произвольной, в том числе и пустой строке; - ? —

соответствует любому одинарному символу; - [c1-c1] — соответствует любому символу, лексикографически находящемуся между символами c1 и c2.

Например, - `echo *` — выведет имена всех файлов текущего каталога, что представляет собой простейший аналог команды `ls`; `-ls *.c` — выведет все файлы с последними двумя символами, совпадающими с `.c`. `-echo prog.?` — выведет все файлы, состоящие из пяти или шести символов, первыми пятью символами которых являются `prog.` - `[a-z]*` — соответствует произвольному имени файла в текущем каталоге, начинающемуся с любой строчной буквы латинского алфавита.

Такие символы, как ' < > * ? | \ " &, являются метасимволами и имеют для командного процессора специальный смысл. Снятие специального смысла с метасимвола называется экранированием метасимвола. Экранирование может быть осуществлено с помощью предшествующего метасимволу символа \, который, в свою очередь, является метасимволом.

Для экранирования группы метасимволов нужно заключить её в одинарные кавычки. Строка, заключённая в двойные кавычки, экранирует все метасимволы, кроме \$, ' , \, ". Например, - `echo *` выведет на экран символ *, - `echo ab'*\|*'cd` выведет на экран строку `ab*\|*cd`.

Командные файлы и функции

Последовательность команд может быть помещена в текстовый файл. Такой файл называется командным. Далее этот файл можно выполнить по команде:

```
bash командный_файл [аргументы]
```

Чтобы не вводить каждый раз последовательности символов `bash`, необходимо изменить код защиты этого командного файла, обеспечив доступ к этому файлу по выполнению. Это может быть сделано с помощью команды

```
chmod +x имя_файла
```

Теперь можно вызывать свой командный файл на выполнение, просто вводя его имя с терминала так, как-будто он является выполняемой программой. Командный процессор распознает, что в Вашем файле на самом деле хранится не выполняемая программа, а программа, написанная на языке программирования оболочки, и осуществит её интерпретацию.

Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключённых в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`.

Команда `typeset` имеет четыре опции для работы с функциями: - `-f` — перечисляет определённые на текущий момент функции; - `-ft` — при последующем вызове функции иницирует её трассировку; - `-fx` — экспортирует все перечисленные функции в любые дочерние программы оболочки; - `-fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции

хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FPATH`, отыскивая файл с одноимёнными именами функций, загружает его и вызывает эти функции.

Передача параметров в командные файлы и специальные переменные

При вызове командного файла на выполнение параметры ему могут быть переданы точно таким же образом, как и выполняемой программе. С точки зрения командного файла эти параметры являются позиционными. Символ `$` является метасимволом командного процессора. Он используется, в частности, для ссылки на параметры, точнее, для получения их значений в командном файле. В командный файл можно передать до девяти параметров. При использовании где-либо в командном файле комбинации символов `$i`, где $0 < i < 10$, вместо неё будет осуществлена подстановка значения параметра с порядковым номером i , т.е. аргумента командного файла с порядковым номером i . Использование комбинации символов `$0` приводит к подстановке вместо неё имени данного командного файла. Рассмотрим это на примере.

Пусть к командному файлу `where` имеется доступ по выполнению и этот командный файл содержит следующий конвейер:

```
who | grep $1.
```

Если Вы введёте с терминала команду `where andy`, то в случае, если пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, то на терминал будет выведена строка, содержащая номер терминала, используемого указанным пользователем. Если же в данный момент этот пользователь не работает в ОС UNIX, то на терминал ничего не будет выведено.

Команда `grep` производит контекстный поиск в тексте, поступающем со стандартного ввода, для нахождения в этом тексте строк, содержащих последовательности символов, переданные ей в качестве аргументов, и выводит результаты своей работы на стандартный вывод. В примере команда `grep` используется как фильтр, обеспечивающий ввод со стандартного ввода и вывод всех строк, содержащих последовательность символов `andy`, на стандартный вывод.

В ходе интерпретации файла командным процессором вместо комбинации символов `$1` осуществляется подстановка значения первого и единственного параметра `andy`. Если предположить, что пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, то на терминале Вы увидите примерно следующее:

```
$ where andy
andy ttyG Jan 14    09:12
$
```

Определим функцию, которая изменяет каталог и печатает список файлов:

```
$ function clist {  
>   cd $1  
>   ls  
> }
```

Теперь при вызове команды `clist` будет изменён каталог и выведено его содержимое.

Команда `shift` позволяет удалять первый параметр и сдвигает все остальные на места предыдущих.

При использовании в командном файле комбинации символов `$#` вместо неё будет осуществлена подстановка числа параметров, указанных в командной строке при вызове данного командного файла на выполнение.

Вот ещё несколько специальных переменных, используемых в командных файлах: - `$*` — отображается вся командная строка или параметры оболочки; - `$?` — код завершения последней выполненной команды; - `$$` — уникальный идентификатор процесса, в рамках которого выполняется командный процессор; - `$!` — номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда; - `$-` — значение флагов командного процессора; - `${#*}` — возвращает целое число — количество слов, которые были результатом `$*`; - `${#name}` — возвращает целое значение длины строки в переменной `name`; - `${name[n]}` — обращение к `n`-му элементу массива; - `${name[*]}` — перечисляет все элементы массива, разделённые пробелом; - `${name[@]}` — то же самое, но позволяет учитывать символы пробелы в самих переменных; - `${name:-value}` — если значение переменной `name` не определено, то оно будет заменено на указанное `value`; - `${name:value}` — проверяется факт существования переменной; - `${name=value}` — если `name` не определено, то ему присваивается значение `value`; - `${name?value}` — останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке; - `${name+value}` — это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`; - `${name#pattern}` — представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`); - `${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.

Использование команды `getopts`

Весьма необходимой при программировании является команда `getopts`, которая осуществляет синтаксический анализ командной строки, выделяя флаги, и используется для объявления переменных. Синтаксис команды следующий:

```
getopts option-string variable [arg ... ]
```

Флаги — это опции командной строки, обычно помеченные знаком минус; Например, для команды `ls` флагом может являться `-F`. Иногда флаги имеют аргументы, связанные с ними. Программы интерпретируют флаги, соответствующим образом изменяя своё поведение.

Строка опций option-string — это список возможных букв и чисел соответствующего флага. Если ожидается, что некоторый флаг будет сопровождаться некоторым аргументом, то за символом, обозначающим этот флаг, должно следовать двоеточие. Соответствующей переменной присваивается буква данной опции. Если команда getoptс может распознать аргумент, то она возвращает истину. Принято включать getoptс в цикл while и анализировать введенные данные с помощью оператора case.

Предположим, необходимо распознать командную строку следующего формата:

```
testprog -ifile_in.txt -ofile_out.doc -L -t -r
```

Вот как выглядит использование оператора getoptс в этом случае:

```
while getoptс o:i:Ltr optletter
do case $optletter in
o) oflag=1; oval=$OPTARG;;
i) iflag=1; ival=$OPTARG;;
L) Lflag=1;;
t) tflag=1;;
r) rflag=1;;
*) echo Illegal option $optletter esac
done
```

Функция getoptс включает две специальные переменные среды — OPTARG и OPTIND. Если ожидается дополнительное значение, то OPTARG устанавливается в значение этого аргумента (будет равна file_in.txt для опции i и file_out.doc для опции o. OPTIND является числовым индексом на упомянутый аргумент.

Функция getoptс также понимает переменные типа массив, следовательно, можно использовать её в функции не только для синтаксического анализа аргументов функций, но и для анализа введенных пользователем данных.

Управление последовательностью действий в командных файлах

Часто бывает необходимо обеспечить проведение каких-либо действий циклически и управление дальнейшими действиями в зависимости от результатов проверки некоторого условия. Для решения подобных задач язык программирования bash предоставляет возможность использовать такие управляющие конструкции, как for, case, if и while. С точки зрения командного процессора эти управляющие конструкции являются обычными командами и могут использоваться как при создании командных файлов, так и при работе в интерактивном режиме. Команды, реализующие подобные конструкции, по сути, являются операторами языка программирования bash. Поэтому при описании языка программирования bash термин оператор будет использоваться наравне с термином команда.

Команды ОС UNIX возвращают код завершения, значение которого может быть использовано для принятия решения о дальнейших действиях. Команда test, например, создана специально для использования в командных файлах.

Единственная функция этой команды заключается в выработке кода завершения. Так например, команда

```
test -f file
```

возвращает нулевой код завершения (истина), если файл file существует, и ненулевой код завершения (ложь) в противном случае: - test s — истина, если аргумент s имеет значение истина; - test -f file — истина, если файл file существует; - test -i file — истина, если файл file доступен по чтению; - test -w file — истина, если файл file доступен по записи; - test -e file — истина, если файл file — исполняемая программа; - test -d file — истина, если файл file является каталогом.

Оператор цикла for

В обобщённой форме оператор цикла for выглядит следующим образом:

```
for имя [in список-значений]
do список-команд
done
```

При каждом следующем выполнении оператора цикла for переменная имя принимает следующее значение из списка значений, задаваемых списком список-значений. Вообще говоря, список-значений является необязательным. При его отсутствии оператор цикла for выполняется для всех позиционных параметров или, иначе говоря, аргументов. Таким образом, оператор for i эквивалентен оператору for i in *. Выполнение оператора цикла for завершается, когда список-значений будет исчерпан. Последовательность команд (операторов), задаваемая списком список-команд, состоит из одной или более команд оболочки, отделённых друг от друга с помощью символов newline или ;.

Рассмотрим примеры использования оператора цикла for.

В результате выполнения оператора

```
for A in alpha beta gamma
do echo A
done
```

на терминал будет выведено следующее:

```
alpha
beta
gamma
```

Предположим, что Вы хотите найти во всех файлах текущего каталога, содержащих исходные тексты программ, написанных на языке программирования Си, все вхождения функции с некоторым именем. Это можно сделать с помощью такой последовательности команд:

```
for i
do
grep $i *.c
done
```

Поместив эту последовательность команд в файл findref, после возможно, используя команду

```
findref 'hash(' 'insert(' 'symbol(',
```

вывести на терминал все строки из всех файлов текущего каталога, имена которых оканчиваются символами .c, содержащие ссылки на функции hash(), insert() и symbol(). Использование символов ' в вышеприведённом примере необходимо для снятия специального смысла с символа (.

Оператор выбора case

Оператор выбора case реализует возможность ветвления на произвольное число ветвей. Эта возможность обеспечивается в большинстве современных языков программирования, предполагающих использование структурного подхода.

В обобщённой форме оператор выбора case выглядит следующим образом:

```
case имя in
шаблон1) список-команд;; шаблон2) список-команд;;
...
esac
```

Выполнение оператора выбора case сводится к тому, что выполняется последовательность команд (операторов), задаваемая списком список-команд, в строке, для которой значение переменной имя совпадает с шаблоном. Поскольку метасимвол * соответствует произвольной, в том числе и пустой, последовательности символов, то его можно использовать в качестве шаблона в последней строке перед служебным словом esac. В этом случае реализуются все действия, которые необходимо произвести, если значение переменной имя не совпадает ни с одним из шаблонов, заданных в предшествующих строках.

Рассмотрим примеры использования оператора выбора case.

В результате выполнения оператора

```
for A in alpha beta gamma
do case $A in
alpha) V=a;; beta) V=c;; gamma) V=e
esac
echo $V
done
```

на терминал будет выведено следующее:

a
c
e

Условный оператор if

В обобщённой форме условный оператор if выглядит следующим образом:

```
if список-команд
then список-команд
{elif список-команд then список-команд}
[else список-команд]
fi
```

Выполнение условного оператора if сводится к тому, что сначала выполняется последовательность команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово if. Затем, если последняя выполненная команда из этой последовательности команд возвращает нулевой код завершения (истина), то будет выполнена последовательность команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово then. Фраза elif проверяется в том случае, когда предыдущая проверка была ложной. Строка, содержащая служебное слово else, является необязательной. Если она присутствует, то последовательность команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово else, будет выполнена только при условии, что последняя выполненная команда из последовательности команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово if или elif, возвращает ненулевой код завершения (ложь).

Рассмотрим следующий пример:

```
for A in *
do if test -d $A
    then echo $A: is a directory
    else echo -n $A: is a file and
        if test -w $A
            then echo writeable elif test -r $A then echo readable
            else echo neither readable nor writeable
        fi
    fi
done
```

Первая строка в приведённом выше примере обеспечивает выполнение всех последующих действий в цикле для всех имён файлов из текущего каталога. При этом переменная A на каждом шаге последовательно принимает значения, равные именам этих файлов. Первая содержащая служебное слово if строка проверяет, является ли файл, имя которого представляет собой текущее значение переменной A, каталогом. Если этот файл является каталогом, то на стандартный вывод выводятся имя этого файла и сообщение о том, что файл с указанным именем

является каталогом. Эти действия в приведённом выше примере обеспечиваются в результате выполнения третьей строки.

Оставшиеся строки выполняются только в том случае, если проверка того, является ли файл, имя которого представляет собой текущее значение переменной A, каталогом, даёт отрицательный ответ. Это означает, что файл, имя которого представляет собой текущее значение переменной A, является обычным файлом. Если этот файл является обычным файлом, то на стандартный вывод выводятся имя этого файла и сообщение о том, что файл с указанным именем является обычным файлом. Эти действия в приведённом выше примере обеспечиваются в результате выполнения четвертой строки. Особенностью использования команды echo в этой строке является использование флага -n, благодаря чему выводимая командой echo строка не будет дополнена символом newline (перевод строки), что позволяет впоследствии дополнить эту строку, как это, например, показано в приведённом выше примере.

Вторая строка, содержащая служебное слово if, проверяет, доступен ли по записи файл, имя которого представляет собой текущее значение переменной A. Если этот файл доступен по записи, то строка дополняется соответствующим сообщением. Если же этот файл недоступен по записи, то проверяется, доступен ли этот файл по чтению. Эти действия в приведённом выше примере обеспечиваются в результате выполнения седьмой строки. Если этот файл доступен по чтению, то строка дополняется соответствующим сообщением. Если же этот файл недоступен ни по записи, ни по чтению, то строка также дополняется соответствующим сообщением. Эти действия в приведённом выше примере обеспечиваются в результате выполнения девятой строки.

Операторы цикла `while` и `until`

В обобщённой форме оператор цикла `while` выглядит следующим образом:

```
while список-команд
do список-команд
done
```

Выполнение оператора цикла `while` сводится к тому, что сначала выполняется последовательность команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово `while`, а затем, если последняя выполненная команда из этой последовательности команд возвращает нулевой код завершения (истина), выполняется последовательность команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово `do`, после чего осуществляется безусловный переход на начало оператора цикла `while`. Выход из цикла будет осуществлён тогда, когда последняя выполненная команда из последовательности команд (операторов), которую задаёт список-команд в строке, содержащей служебное слово `while`, возвратит ненулевой код завершения (ложь).

Приведённый ниже фрагмент командного файла иллюстрирует использование оператора цикла `while`. В нем реализуется ожидание события, состоящего в удалении

файла с определённым именем, и только после наступления этого события производятся дальнейшие действия:

```
while test -f lockfile
do sleep 30
    echo waiting for semaphore
done
```

:create the semaphore file

```
echo > lockfile
:further commands and after them delete the semaphore file rm lockfile
```

Командный файл, продемонстрированный в приведённом примере, по сути, является простейшей реализацией механизма синхронизации взаимодействующих процессов на основе семафоров.

При замене в операторе цикла while служебного слова while на until условие, при выполнении которого осуществляется выход из цикла, меняется на противоположное. В остальном оператор цикла while и оператор цикла until идентичны. В обобщённой форме оператор цикла until выглядит следующим образом:

```
until список-команд
do список-команд
done
```

Следующие две команды ОС UNIX используются только совместно с управляющими конструкциями языка программирования bash: это команда true, которая всегда возвращает код завершения, равный нулю (т.е. истина), и команда false, которая всегда возвращает код завершения, не равный нулю (т. е. ложь).

Ниже приведены два примера, иллюстрирующие бесконечные циклы, которые будут выполняться до тех пор, пока ЭВМ не сломается или не будет выключена (ну, по крайней мере, до тех пор, пока Вы не нажмёте клавишу, соответствующую специальному символу INTERRUPT):

```
while true
do echo hello andy
done
```

```
until false
do echo hello mike
done
```

Прерывание циклов

Два несложных способа позволяют вам прерывать циклы в оболочке bash. Команда break завершает выполнение цикла, а команда continue завершает данную итерацию блока операторов.

Команда `break` полезна для завершения цикла `while` в ситуациях, когда условие перестает быть правильным. Пример бесконечного цикла `while` с прерыванием в момент, когда файл перестает существовать:

```
while true
do
    if [! -f $file]
    then
        break
    fi
    sleep 10
done
```

Команда `continue` используется в ситуациях, когда больше нет необходимости выполнять блок операторов, но вы можете захотеть продолжить проверять данный блок на других условных выражениях. Пример предназначен для игнорирования файла `/dev/null` в произвольном списке:

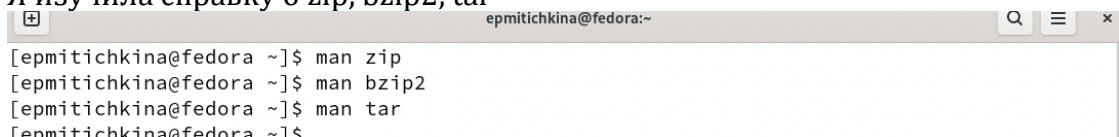
```
while file=$filelist[$i]
    (( $i < ${#filelist[*]} ))
do
    if
        [ "$file" == "dev/null" ]
    then
        continue
    fi
    action
done
```

Эта программа пропускает нужное значение, но продолжает тестировать остальные.

Выполнение лабораторной работы

Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию `backup` в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор `zip`, `bzip2` или `tar`. Способ использования команд архивации необходимо узнать, изучив справку.

1. Я изучила справку о `zip`, `bzip2`, `tar`



```
epmitichkina@fedora:~$ man zip
epmitichkina@fedora ~$ man bzip2
epmitichkina@fedora ~$ man tar
epmitichkina@fedora ~$
```

```
epmitichkina@fedora:~ — man zip
ZIP(1L)
ZIP(1L)

NAME
    zip - package and compress (archive) files

SYNOPSIS
    zip [-aABcdDeEfFghjklLmoqrRSTuvVwXyz!@&$] [--longoption ...] [-b path]
        [-n suffixes] [-t date] [-tt date] [zipfile [file ...]] [-xi list]

    zipcloak (see separate man page)

    zipnote (see separate man page)

    zipsplit (see separate man page)

Note: Command line processing in zip has been changed to support long
options and handle all options and arguments more consistently. Some
old command lines that depend on command line inconsistencies may no
longer work.

DESCRIPTION
    zip is a compression and file packaging utility for Unix, VMS, MSDOS,
    OS/2, Windows 9x/NT/XP, Minix, Atari, Macintosh, Amiga, and Acorn RISC
    OS. It is analogous to a combination of the Unix commands tar(1) and
    compress(1) and is compatible with PKZIP (Phil Katz's ZIP for MSDOS
    systems).

    A companion program (unzip(1L)) unpacks zip archives. The zip and un-
zip(1L) programs can work with archives produced by PKZIP (supporting
    most PKZIP features up to PKZIP version 4.6), and PKZIP and PKUNZIP can
    work with archives produced by zip (with some exceptions, notably
    streamed archives, but recent changes in the zip file standard may fa-
    cilitate better compatibility). zip version 3.0 is compatible with
    PKZIP 2.04 and also supports the Zip64 extensions of PKZIP 4.5 which
    allow archives as well as files to exceed the previous 2 GB limit (4 GB
    in some cases). zip also now supports bzip2 compression if the bzip2
    library is included when zip is compiled. Note that PKUNZIP 1.10 can-
    not extract files produced by PKZIP 2.04 or zip 3.0. You must use PKUN-
    ZIP 2.04g or unzip 5.0p1 (or later versions) to extract them.
```

Manual page zip(1) line 1 (press h for help or q to quit)

```
epmitichkina@fedora:~ — man bzip2
bzip2(1)                                General Commands Manual                                bzip2(1)

NAME
    bzip2, bunzip2 - a block-sorting file compressor, v1.0.8
    bzcat - decompresses files to stdout
    bzip2recover - recovers data from damaged bzip2 files

SYNOPSIS
    bzip2 [ -cdfkqstvzVL123456789 ] [ filenames ... ]
    bunzip2 [ -fkvsVL ] [ filenames ... ]
    bzcat [ -s ] [ filenames ... ]
    bzip2recover filename

DESCRIPTION
    bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors.

    The command-line options are deliberately very similar to those of GNU gzip, but they are not identical.

    bzip2 expects a list of file names to accompany the command-line flags. Each file is replaced by a compressed version of itself, with the name "original_name.bz2". Each compressed file has the same modification date, permissions, and, when possible, ownership as the corresponding original, so that these properties can be correctly restored at decompression time. File name handling is naive in the sense that there is no mechanism for preserving original file names, permissions, ownerships or dates in filesystems which lack these concepts, or have serious file name length restrictions, such as MS-DOS.

    bzip2 and bunzip2 will by default not overwrite existing files. If you want this to happen, specify the -f flag.

    If no file names are specified, bzip2 compresses from standard input to standard output. In this case, bzip2 will decline to write compressed output to a terminal, as this would be entirely incomprehensible and therefore pointless.

    bunzip2 (or bzip2 -d) decompresses all specified files. Files which were not
```

```
epmitichkina@fedora:~ — man tar
TAR(1) GNU TAR Manual TAR(1)

NAME
    tar - an archiving utility

SYNOPSIS
    Traditional usage
        tar {A|c|d|r|t|u|x}[GnSkUW0mpsMBiajJzZhPlRvwo] [ARG...]

    UNIX-style usage
        tar -A [OPTIONS] ARCHIVE ARCHIVE

        tar -c [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -d [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -t [-f ARCHIVE] [OPTIONS] [MEMBER...]

        tar -r [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -u [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -x [-f ARCHIVE] [OPTIONS] [MEMBER...]

    GNU-style usage
        tar {--catenate|--concatenate} [OPTIONS] ARCHIVE ARCHIVE

        tar --create [--file ARCHIVE] [OPTIONS] [FILE...]

        tar {--diff|--compare} [--file ARCHIVE] [OPTIONS] [FILE...]

        tar --delete [--file ARCHIVE] [OPTIONS] [MEMBER...]

        tar --append [-f ARCHIVE] [OPTIONS] [FILE...]

        tar --list [-f ARCHIVE] [OPTIONS] [MEMBER...]

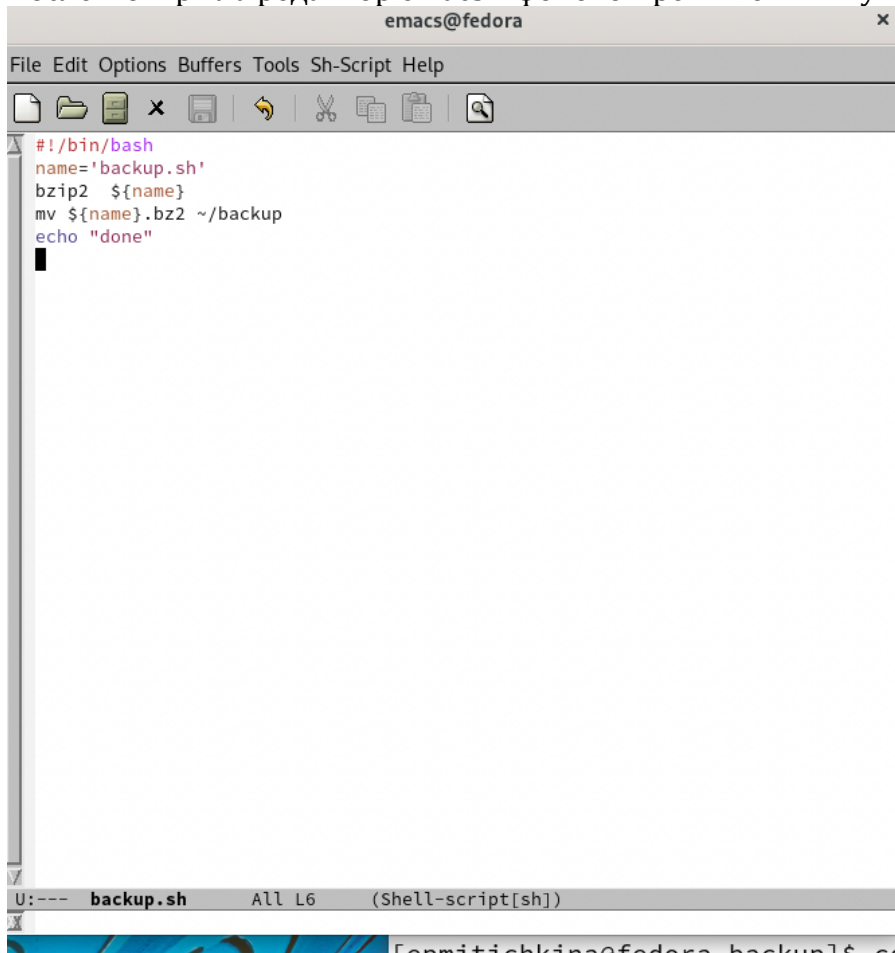
        tar --test-label [--file ARCHIVE] [OPTIONS] [LABEL...]

Manual page tar(1) line 1 (press h for help or q to quit)
```

2. Создала директорию backup и файл backup.sh. Также проверила их наличие

```
[epmitichkina@fedora ~]$ mkdir backup
[epmitichkina@fedora ~]$ ls
01      conf.txt  text.txt  Документы  Музыка      Шаблоны
5       file.txt   work      Загрузки   Общедоступные
backup  montly      Видео     Изображения 'Рабочий стол'
[epmitichkina@fedora ~]$ 
[epmitichkina@fedora ~]$ touch backup.sh
[epmitichkina@fedora ~]$ ls
01      backup.sh  montly      Видео     Изображения 'Рабочий стол'
5       conf.txt  text.txt   Документы  Музыка      Шаблоны
backup  file.txt   work      Загрузки   Общедоступные
[epmitichkina@fedora ~]$
```

3. После я открыла редактор emacs в фоновом режиме и пишу код



```
emacs@fedora
File Edit Options Buffers Tools Sh-Script Help
#!/bin/bash
name='backup.sh'
bzip2 ${name}
mv ${name}.bz2 ~/backup
echo "done"
U:--- backup.sh All L6 (Shell-script[sh])
```

4. Предоставила право на выполнение и проверка файла

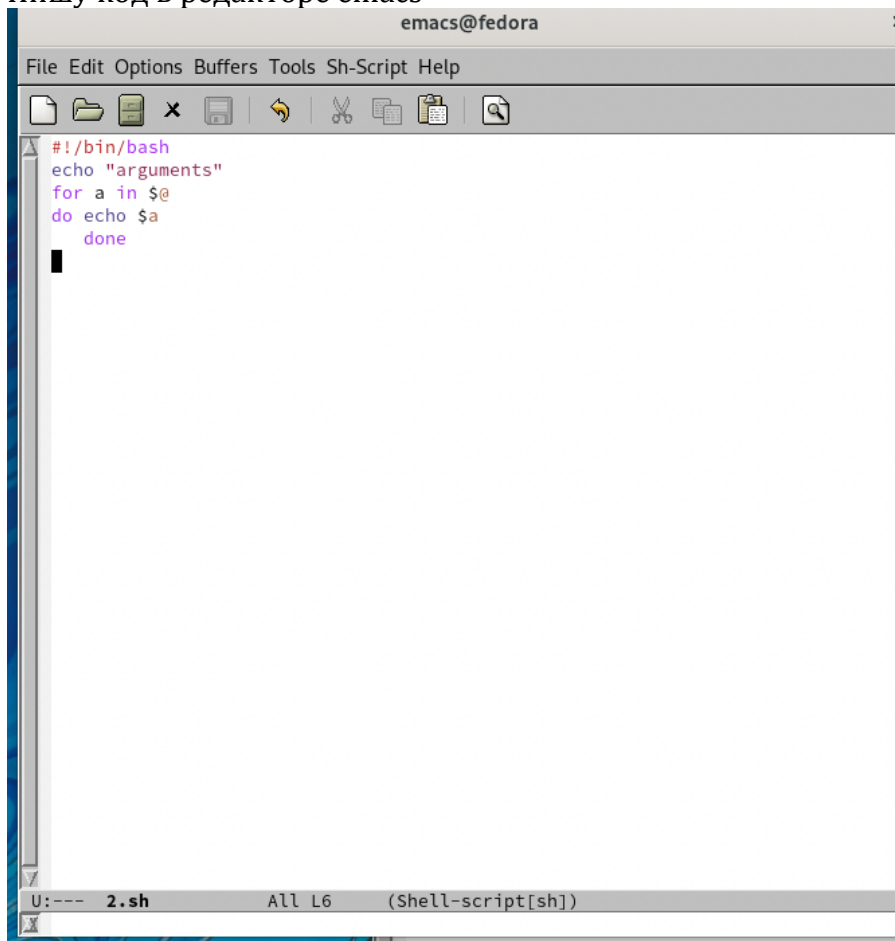
```
[epmitichkina@fedora ~]$ ./backup.sh
done
[epmitichkina@fedora ~]$ cd backup/
[epmitichkina@fedora backup]$ ls
backup.sh.bz2
[epmitichkina@fedora backup]$
```

Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.

1. Создаю файл 2.sh и проверяю

```
[epmitichkina@fedora ~]$ touch 2.sh
[epmitichkina@fedora ~]$ ls
01 backup file.txt work Загрузки Общедоступные
2.sh backup.sh~ montly Видео Изображения 'Рабочий стол'
5 conf.txt text.txt Документы Музыка Шаблоны
[epmitichkina@fedora ~]$
```

2. Пишу код в редакторе emacs

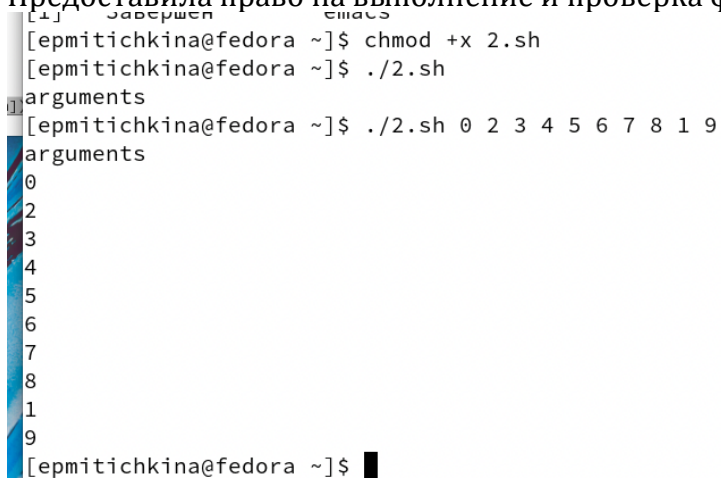


The screenshot shows the Emacs editor window titled "emacs@fedora". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "Sh-Script", and "Help". The toolbar contains icons for file operations and editing. The main text area contains a shell script:

```
#!/bin/bash
echo "arguments"
for a in $@
do echo $a
done
```

The status bar at the bottom indicates the current buffer is "2.sh", line 6, column 1, in a shell script mode.

3. Предоставила право на выполнение и проверка файла



The screenshot shows a terminal window with the following commands and output:

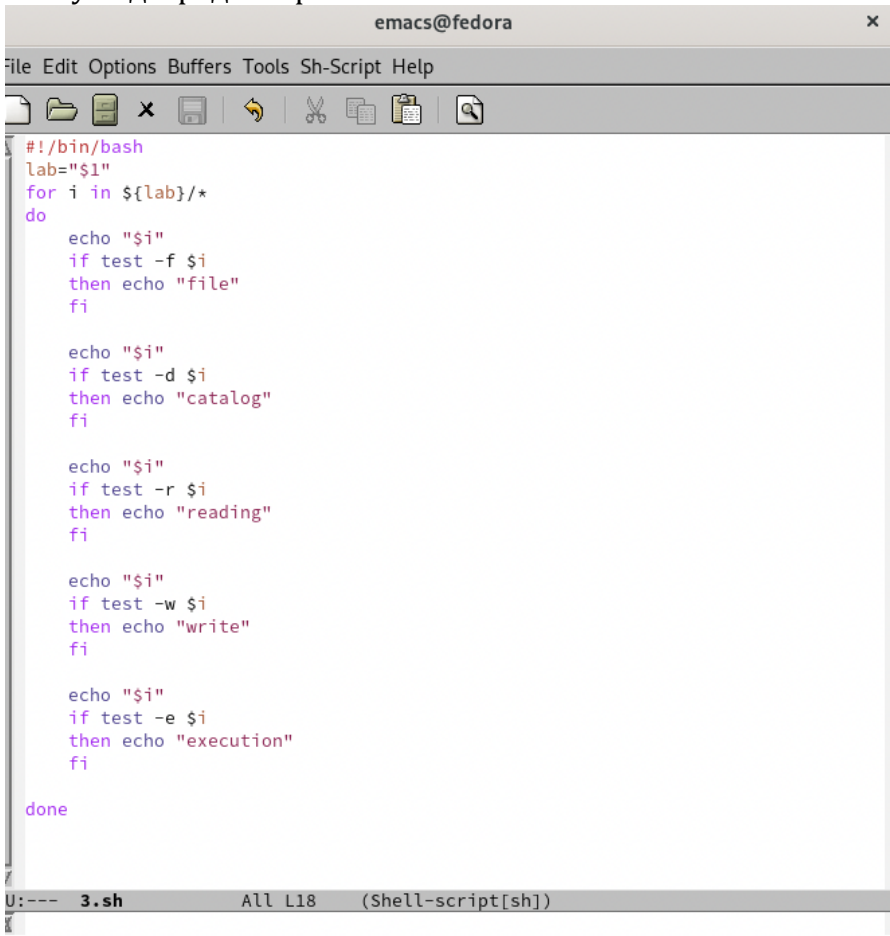
```
[epmitichkina@fedora ~]$ chmod +x 2.sh
[epmitichkina@fedora ~]$ ./2.sh
arguments
[epmitichkina@fedora ~]$ ./2.sh 0 2 3 4 5 6 7 8 1 9
arguments
0
2
3
4
5
6
7
8
1
9
[epmitichkina@fedora ~]$
```

Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.

1. Создаю файл 3.sh и проверяю

```
[epmitichkina@fedora ~]$ touch 3.sh
[epmitichkina@fedora ~]$ ls
01      3.sh      backup.sh~  montly  Видео      Изображения  'Рабочий сто
2.sh    5          conf.txt   text.txt  Документы  Музыка        Шаблоны
2.sh~   backup    file.txt   work     Загрузки   Общедоступные
[epmitichkina@fedora ~]$
```

2. Пишу код в редакторе emacs



```
emacs@fedora
File Edit Options Buffers Tools Sh-Script Help
#!/bin/bash
lab="$1"
for i in ${lab}/*
do
    echo "$i"
    if test -f $i
    then echo "file"
    fi

    echo "$i"
    if test -d $i
    then echo "catalog"
    fi

    echo "$i"
    if test -r $i
    then echo "reading"
    fi

    echo "$i"
    if test -w $i
    then echo "write"
    fi

    echo "$i"
    if test -e $i
    then echo "execution"
    fi
done

U:--- 3.sh      All L18      (Shell-script[sh])
```


3. Предоставила право на выполнение и проверка файла

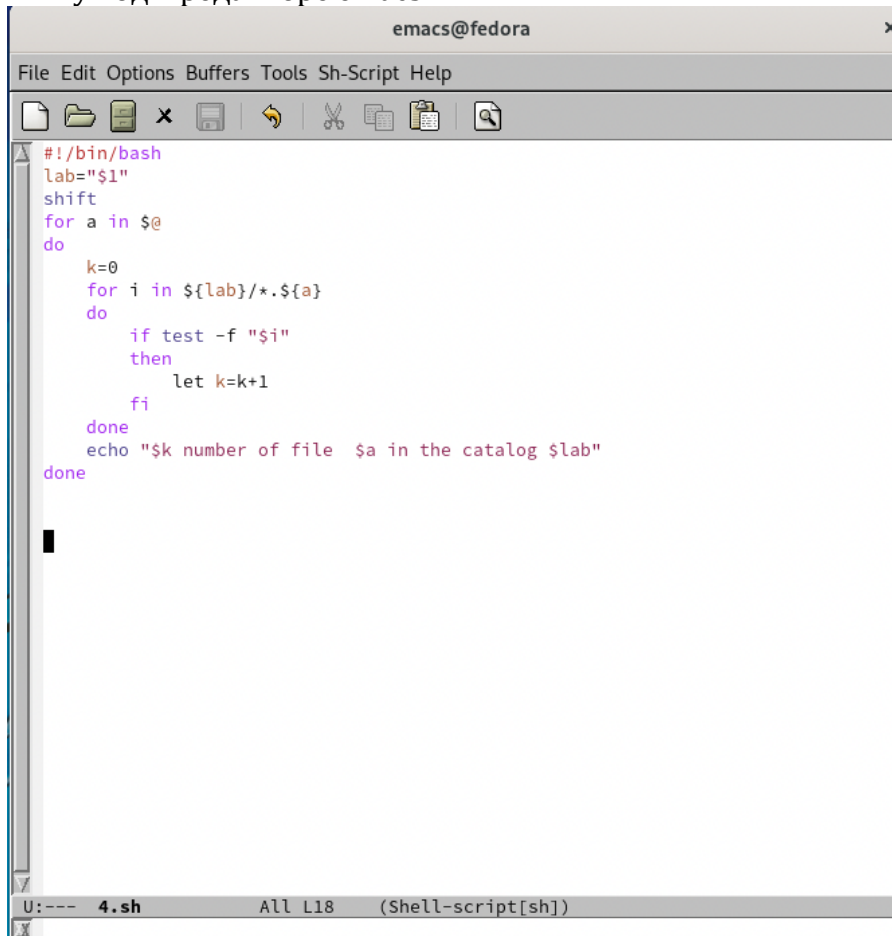
```
[epmitichkina@fedora ~]$ chmod +x 3.sh
[epmitichkina@fedora ~]$ ./3.sh ~
/home/epmitichkina/01
/home/epmitichkina/01
catalog
/home/epmitichkina/01
reading
/home/epmitichkina/01
write
/home/epmitichkina/01
execution
/home/epmitichkina/2.sh
file
/home/epmitichkina/2.sh
/home/epmitichkina/2.sh
reading
/home/epmitichkina/2.sh
write
/home/epmitichkina/2.sh
execution
/home/epmitichkina/2.sh~
file
/home/epmitichkina/2.sh~
/home/epmitichkina/2.sh~
reading
/home/epmitichkina/2.sh~
write
/home/epmitichkina/2.sh~
```

Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки.

1. Создаю файл 4.sh и проверяю

```
[epmitichkina@fedora ~]$ touch 4.sh
[2]- Завершён      emacs
[3]+ Завершён      emacs
[epmitichkina@fedora ~]$ ls
01      3.sh    5      conf.txt  text.txt  Документы  Музыка      Шаблоны
2.sh    3.sh~   backup  file.txt  work      Загрузки   Общедоступные
2.sh~   4.sh    backup.sh~  montly    Видео     Изображения 'Рабочий стол'
```


2. Пишу код в редакторе emacs



```
emacs@fedora
File Edit Options Buffers Tools Sh-Script Help
#!/bin/bash
lab="$1"
shift
for a in $@
do
    k=0
    for i in ${lab}/*.${a}
    do
        if test -f "$i"
        then
            let k=k+1
        fi
    done
    echo "$k number of file  $a in the catalog $lab"
done
```

3. Предоставила право на выполнение и проверка файла

```
[epmitichkina@fedora ~]$ chmod +x 4.sh
[epmitichkina@fedora ~]$ ./4.sh ~ sh
3 number of file  sh in the catalog /home/epmitichkina
[epmitichkina@fedora ~]$
```

Выводы

В результате работы изучила основы программирования в оболочке ОС UNIX/Linux. Научилась писать небольшие командные файлы.

Ответы на контрольные вопросы

1. Объясните понятие командной оболочки. Приведите примеры командных оболочек. Чем они отличаются?

Командный процессор (командная оболочка, интерпретатор команд shell) – это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера.

В операционных системах типа UNIX/Linux наиболее часто используются следующие реализации командных оболочек: - оболочка Борна (Bourne shell или sh) — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций; - C-оболочка (или csh) — надстройка на оболочке Борна, использующая Сподобный синтаксис команд с возможностью сохранения истории выполнения команд; - оболочка Корна (или ksh) — напоминает оболочку С, но операторы управления программой совместимы с операторами оболочки Борна; - BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек С и Корна (разработка компании Free Software Foundation).

2. Что такое POSIX?

POSIX (Portable Operating System Interface for Computer Environments) — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ. Стандарты POSIX разработаны комитетом IEEE (Institute of Electrical and Electronics Engineers) для обеспечения совместимости различных UNIX/Linux-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. POSIX-совместимые оболочки разработаны на базе оболочки Корна.

3. Как определяются переменные и массивы в языке программирования bash?

Командный процессор bash обеспечивает возможность использования переменных типа строка символов. Имена переменных могут быть выбраны пользователем.

Пользователь имеет возможность присвоить переменной значение некоторой строки символов. Например, команда «mark=/usr/andy/bin» присваивает значение строки символов /usr/andy/bin переменной mark типа строка символов. Значение, присвоенное некоторой переменной, может быть впоследствии использовано. Для этого в соответствующем месте командной строки должно быть употреблено имя этой переменной, которому предшествует метасимвол

. Например, команда «mv afile{mark}» переместит файл afile из текущего каталога в каталог с абсолютным полным именем /usr/andy/bin. Оболочка bash позволяет работать с массивами. Для создания массива используется команда setc флагом -A. За флагом следует имя переменной, а затем список значений, разделённых пробелами. Например, «set -A states Delaware Michigan "New Jersey"». Далее можно сделать добавление в массив, например, states[49]=Alaska. Индексация массивов начинается с нулевого элемента.

4. Каково назначение операторов let и read?

Команда let является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Простейшее выражение – это единичный терм (term), обычно целочисленный. Команда let берет два операнда и присваивает их переменной. Команда read позволяет читать значения переменных со стандартного ввода: «echo "Please enter Month and Day of Birth ?"» «read mon day trash». В переменные mon day будут считаны соответствующие значения,

введённые с клавиатуры, а переменная `trash` нужна для того, чтобы отобразить всю избыточно введённую информацию и игнорировать её.

5. Какие арифметические операции можно применять в языке программирования `bash`?

В языке программирования `bash` можно применять такие арифметические операции как сложение (+), вычитание (-), умножение (), целочисленное деление (/) и целочисленный остаток от деления (%).

6. Что означает операция (())?

В (()) записывают условия оболочки `bash`, а также внутри двойных скобок можно вычислять арифметические выражения и возвращать результат.

7. Какие стандартные имена переменных Вам известны?

Стандартные переменные: - `PATH`: значением данной переменной является список каталогов, в которых командный процессор осуществляет поиск программы или команды, указанной в командной строке, в том случае, если указанное имя программы или команды не содержит ни одного символа /. Если имя команды содержит хотя бы один символ /, то последовательность поиска, предписываемая значением переменной `PATH`, нарушается. В этом случае в зависимости от того, является имя команды абсолютным или относительным, поиск начинается соответственно от корневого или текущего каталога. - `PS1` и `PS2`: эти переменные предназначены для отображения промптера командного процессора. `PS1` – это промптер командного процессора, по умолчанию его значение равно символу \$ или #. Если какая-то интерактивная программа, запущенная командным процессором, требует ввода, то используется промптер `PS2`. Он по умолчанию имеет значение символа >. - `HOME` — имя домашнего каталога пользователя. Если команда `cd` вводится без аргументов, то происходит переход в каталог, указанный в этой переменной. - `IFS` — последовательность символов, являющихся разделителями в командной строке, например, пробел, табуляция и перевод строки (new line). - `MAIL` — командный процессор каждый раз перед выводом на экран промптера проверяет содержимое файла, имя которого указано в этой переменной, и если содержимое этого файла изменилось с момента последнего ввода из него, то перед тем как вывести на терминал промптер, командный процессор выводит на терминал сообщение `You have mail` (у Вас есть почта). - `TERM` — тип используемого терминала. - `LOGNAME` — содержит регистрационное имя пользователя, которое устанавливается автоматически при входе в систему.

8. Что такое метасимволы?

Такие символы, как ' < > ? | " &, являются метасимволами и имеют для командного процессора специальный смысл.

9. Как экранировать метасимволы?

Снятие специального смысла с метасимвола называется экранированием мета символа. Экранирование может быть осуществлено с помощью предшествующего мета символу символа `\`, который, в свою очередь, является мета символом. Для экранирования группы метасимволов нужно заключить её в одинарные кавычки. Строка, заключённая в двойные кавычки, экранирует все метасимволы, кроме `$`, `'`, `,` и `"`. Например, `-echo*` выведет на экран символ `*`, `-echoab'|'cd` выведет на экран строку `ab|cd`.

10. Как создавать и запускать командные файлы?

Последовательность команд может быть помещена в текстовый файл. Такой файл называется командным. Далее этот файл можно выполнить по команде: `bash командный_файл [аргументы]`. Чтобы не вводить каждый раз последовательности символов `bash`, необходимо изменить код защиты этого командного файла, обеспечив доступ к этому файлу по выполнению. Это может быть сделано с помощью команды `chmod +x имя_файла`. Теперь можно вызывать свой командный файл на выполнение, просто вводя его имя с терминала так, как будто он является выполняемой программой. Командный процессор распознает, что в Вашем файле на самом деле хранится не выполняемая программа, а программа, написанная на языке программирования оболочки, и осуществить её интерпретацию.

11. Как определяются функции в языке программирования bash?

Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключённых в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`.

12. Каким образом можно выяснить, является файл каталогом или обычным файлом?

Чтобы выяснить, является ли файл каталогом или обычным файлом, необходимо воспользоваться командами `test -f [путь до файла]` (для проверки, является ли обычным файлом) и `test -d [путь до файла]` (для проверки, является ли каталогом).

13. Каково назначение команд `set`, `typeset` и `unset`?

«`set`» можно использовать для вывода списка переменных окружения. В системах Ubuntu и Debian команда «`set`» также выведет список функций командной оболочки после списка переменных командной оболочки. Поэтому для ознакомления со всеми элементами списка переменных окружения при работе с данными системами рекомендуется использовать команду «`set | more`». Команда «`typeset`» предназначена для наложения ограничений на переменные. Команду «`unset`» следует использовать для удаления переменной из окружения командной оболочки.

14. Как передаются параметры в командные файлы?

При вызове командного файла на выполнение параметры ему могут быть переданы точно таким же образом, как и выполняемой программе. С точки зрения командного

файла эти параметры являются позиционными. Символ \$ является метасимволом командного процессора.

15. Назовите специальные переменные языка bash и их назначение.

- `$*` — отображается вся командная строка или параметры оболочки;
- `$?` — код завершения последней выполненной команды;
- `$$` — уникальный идентификатор процесса, в рамках которого выполняется командный процессор;
- `#!` — номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда;
- `$-` — значение флагов командного процессора;
- `${#*}` — возвращает целое число — количество слов, которые были результатом `$*`;
- `${#name}` — возвращает целое значение длины строки в переменной `name`;
- `${name[n]}` — обращение к `n`-му элементу массива;
- `${name[*]}` — перечисляет все элементы массива, разделённые пробелом;
- `${name[@]}` — то же самое, но позволяет учитывать символы пробелы в самих переменных;
- `${name:-value}` — если значение переменной `name` не определено, то оно будет заменено на указанное `value`;
- `${name:value}` — проверяется факт существования переменной;
- `${name=value}` — если `name` не определено, то ему присваивается значение `value`;
- `${name?value}` — останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке;
- `${name+value}` — это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`;
- `${name#pattern}` — представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`);
- `${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.