

Лабораторная работа

Основы программирования в Bash. Сценарии

Цели работы:

- изучение основных команд Bash
- изучение основных правил программирования на языке Bash
- приобретение навыка работы в консольном текстовом редакторе
- создание простейших сценария для интерпретатора Bash

Теоретические сведения Сценарии командной оболочки

Bash-скрипт – текстовый файл, содержащий последовательность команд для интерпретатора Bash. Для его выполнения необходимо указать путь к интерпретатору в первой строке через shebang:

```
#!/usr/bin/env bash
```

Использование `env` обеспечивает переносимость, так как `env` ищет `bash` в переменной `PATH`, что может быть полезно, если интерпретатор установлен не в директории `/bin/bash`. Для создания скриптов могут быть применены консольные текстовые редакторы `vim` и `nano`.

Основные элементы синтаксиса языка Bash

1. Комментарии

Строки, начинающиеся с символа `#` (кроме shebang) игнорируются интерпретатором. Комментарии могут быть оформлены следующим образом:

```
# Эта строка – комментарий  
echo "Hello" # Комментарий после команды
```

2. Переменные

Имена переменных в Bash чувствительны к регистру и могут состоять из буквы, цифр и символа подчеркивания. Bash не поддерживает строгую типизацию: все переменные хранятся как строки, интерпретируются в зависимости от контекста.

Пример объявления переменной и присвоения ей значения:

```
variable_name="value" # Знак присваивания без  
# пробелов вокруг него
```

В примере создается глобальная переменная, доступная в любой точке скрипта. Для объявления локальной (ограниченной функцией или блоком кода) переменной необходимо использовать ключевое слово `local`.

Bash чувствителен к пробелам, если использовать символ пробела, например, перед `"value"`, вместо присвоения значения интерпретатор будет рассматривать `"value"` как команду.

Для получения доступа к текущему значению переменной используется символ `$`. Например, для вывода значения переменной `variable_name`, можно использовать команду: `echo "Значение = $variable_name"`

Помимо переменных, определяемых в самом скрипте, могут быть использованы переменные, не требующие объявления – зарезервированные

переменные. Они доступны для скрипта, так как управляются непосредственно интерпретатором в рамках текущего сеанса работы оболочки или выполнения скрипта.

Пример часто используемых зарезервированных переменных приведен в таблице 2.

Таблица 2. Зарезервированные переменные Bash

Переменная	Описание значения
\$0	Имя скрипта
\$1, \$2, ...	Аргументы командной строки
\$#	Число переданных аргументов
\$@	Все аргументы как отдельные строки
\$?	Код завершения последней команды
\$\$	PID текущего процесса

Помимо приведенных в таблице, полезными могут быть такие переменные, как \$HOME, \$PWD, \$USR и т.д. (с более полным списком переменных рекомендуется ознакомиться на странице руководства: `man bash`)

3. Специальные символы

Для экранирования последующих символов используется символ \. Одинарные кавычки ' ' отключают интерпретацию всех заключенных в них спецсимволов. Двойные кавычки позволяют выполнить подстановку переменных и разрешают выполнение команд.

Пример:

```
str="время"
echo "Текущее $str $(date +%T)" # Выведет текущее время
```

4. Арифметические операции

Для выполнения арифметических операций в Bash нужно указать, что вычисления будут производиться в математическом контексте `$(( ))`. Рассмотрим пример выполнения операции целочисленного деления:

```
echo $((10 / 3)) # Вывод: 3
```

Доступны основные арифметические операции: +, -, *, /, ** (возведение в степень), % (взятие остатка от деления). Данные операции поддерживаются для работы с целыми числами, для работы с дробными числами нужно пользоваться дополнительно утилитами `bc` или `awk`.

5. Обработка ошибок и завершение

Для завершения работы скрипта может использоваться команда `exit` с передачей ей соответствующего значения кода возврата - целого числа от 0 до 255, например:

```
exit 0 # Успех (по умолчанию)
exit 1 # Ошибка (1-255)
```

При успешном завершении работы принято возвращать значение 0, если произошла ошибка - код, который конкретизирует этап возникновения ошибки и ее тип. Для проверки кода завершения используется значение зарезервированной переменной `$?`.

6. Ввод интерактивных пользовательских данных

Для передачи значений в скрипт непосредственно при запуске или во время выполнения можно использовать команду `read` или зарезервированные переменные `$1`, `$2`, Пример использования `read` с опцией `-p` для вывода подсказки, отображаемой в терминале:

```
# Введенное значение записывается в переменную var_to_read
read -p "Введите значение " var_to_read
```

7. Управляющие конструкции

К управляющим конструкциям традиционно относятся условный оператор, оператор выбора и операторы циклов. Базовый синтаксис условного оператора:

```
if [[ условие_1 ]]; then
    команды
elif [[ условие_2 ]]; then
    команды
else
    команды
fi
```

Условный оператор может быть записан как в полной форме, так и без блоков `elif` или `else`. Использование `[[ ]]` вместо `[ ]` считается более современным, т.к. позволяет, в том числе, работать с пустыми переменными и использовать в условиях логические переменные, записанные в привычной форме. Пример сравнения вариантов синтаксиса:

```
# Синтаксис с [ ]
if [ "$a" -gt 5 -a "$a" -lt 10 ]; then
    echo "значение a между 5 и 10"
fi

# Синтаксис с [[ ]] (более современный)
if [[ "$a" > 5 && "$a" < 10 ]]; then
    echo "значение a между 5 и 10"
fi
```

Для работы со списком условий и избегания множества конструкций `if-elif` можно применить конструкцию `case` (оператор выбора) для сопоставления с шаблонами. Пример:

```
case "$var" in
    "start")                # Значение var = "start"
        echo "Запуск процесса"
        ;;
    "stop" | "pause")       # var = "stop" или "pause"
        echo "Остановка процесса"
        ;;
    *)                      # выполняется в других случаях
        echo "Неизвестная команда"
        ;;
esac
```

В Bash существуют три основных типа циклов: `for`, `while`, `until`.

Цикл `for` можно использовать для перебора элементов списка строк, что удобно, например, для просмотра всех файлов в директории или перебора переданных в скрипт аргументов. Пример поиска файлов в домашней директории:

```
for file in $HOME/*; do
    echo "Найден файл: $file"
done
```

Также существует вариант записи цикла `for` для перебора чисел в диапазоне, пример:

```
for ((i=0; i<n; i++)); do
    echo $i
done
```

Цикл `while` выполняется пока условие остается истинным. Пример использования цикла `while` для строчного чтения данных из файла `file.txt`:

```
while read line; do
    echo "Строка: $line"
done < file.txt
```

Цикл `until` выполняется до тех пор, пока условие ложно, что можно использовать, например, для проверки доступности IP-адреса:

```
# Пока 8.8.8.8 не доступен, цикл продолжает работу
until ping -c1 8.8.8.8 &> /dev/null; do
    echo "Адрес 8.8.8.8 не доступен"
    sleep 5
done
echo "Продолжаем работу"
```

Для завершения текущей итерации цикла и перехода к следующей можно использовать оператор `continue`, а для прерывания цикла и передачи управления за него - `break`.

8. Функции Bash

Функции Bash - это именованные блоки кода, которые можно повторно использовать в скриптах.

Объявление функции в Bash:

```
имя_функции () { команды
}
или
имя_функции { команды
}
```

Вызов функции:

```
имя_функции
```

Функция должна быть объявлена до ее первого использования.

Пример создания функции и ее вызова с передачей одного аргумента:

```
#!/usr/bin/env bash
function my_func {
    echo "Эта строка напечатана из функции $1 раз"
```

```

}
count=1
while [[ $count < 3 ]]
do
my_func $count
count=$(( $count + 1 ))
done
echo "Цикл закончен, еще один вызов функции:"
my_func $count
echo "Скрипт завершен"

```

9. Проверка скриптов

Скрипт, содержащий ошибки, может представлять серьезную опасность для системы.

ShellCheck – бесплатный инструмент статического анализа с открытым исходным кодом, который можно использовать для проверки и улучшения скриптов. Он способен обнаруживать как общие, так и граничные ошибки и предлагать соответствующие исправления.

ShellCheck может использоваться как онлайн-овая или системная утилита, а также может быть интегрирован в качестве линтера в различные текстовые редакторы.

Установка ShellCheck в Debian/Ubuntu Linux:

```
sudo apt install shellcheck
```

Порядок выполнения работы

В директории `Ваша_фамилия` (латиницей) создайте подкаталог `practice_3` в котором создавайте все скрипты этой работы.

Задание 1. Напишите скрипт, принимающий от 2 до 5 параметров и выводящий сумму всех переданных значений. При вызове скрипта с неправильным количеством параметров, например, с 1 или с 6, или с параметром(-и), не являющимся(-ися) числовыми значениями, необходимо предусмотреть обработку ошибок и возвращение различных кодов ошибок.

Задание 2. Напишите скрипт, принимающий число `N`, и выводящий все нечетные числа от 1 до `N`. Требуется предусмотреть обработку ошибок.

Задание 3. Напишите скрипт, проверяющий существование файла, переданного в качестве первого аргумента. При отсутствии файла предложите пользователю выбор, создать ли указанный файл, и создайте его в случае положительного ответа.

Задание 4. Напишите скрипт, выводящий меню, предлагающее получить значения некоторых зарезервированных переменных, хранящих системную информацию (например, имя пользователя, идентификатор процесса скрипта и т.п.). Количество пунктов меню - от 6. Предусмотрите пункт завершения работы со скриптом.

Задание 5. Напишите интерактивный калькулятор: скрипт должен запрашивать у пользователя два числа, затем – выполняемую операцию (+,

-, *, /), вычислять и выводить результат в терминал. Требуется предусмотреть обработку ошибок.

Проверьте все написанные скрипты с помощью утилиты ShellCheck.

В отчете по работе приведите скриншоты, демонстрирующие результаты выполнения всех пунктов задания, код разработанных сценариев, скриншоты проверок ShellCheck.

Дополнительное задание

Напишите скрипт для подсчета среднего размера файла в директории:

- путь к директории должен передаваться параметром
- скрипт проверяет, что указанная директория существует, если нет – выводит сообщение об ошибке и завершается с кодом 2
- подсчитывает и выводит на экран средний размер файла в ней;
- при подсчете не нужно учитывать поддиректории и символичные ссылки, считать только средний размер файлов в заданной директории.

Подсказка. Примерный алгоритм решения:

1. В скрипте объявите функцию. Поместите аргумент в переменную. Проверьте, существует ли заданная в аргументе директория с помощью if.
2. Внутри функции объявите вспомогательные переменные. Одна переменная будет служить для сохранения суммы размеров всех файлов в директории, а другая будет содержать количество файлов.
3. Для перебора файлов внутри директории используйте цикл for.
4. Внутри цикла с помощью if проверяйте, является ли данный объект файлом.
5. Если это файл, то к переменной суммы добавляйте размер файла и увеличивайте на один переменную с количеством файлов (получить размер одного файла можно с помощью stat -с "%s" filename).
6. После выхода из цикла for (перебора всех файлов в директории), разделите значение переменной с суммой размеров файлов на количество этих файлов и выведите результат на экран.
7. В конце скрипта, вызовите только что написанную функцию с аргументом, например, avgfile \$1
8. Запустите скрипт, передав в качестве аргумента \$HOME, и проанализируйте результат.