ЛАБОРАТОРНАЯ РАБОТА 4.
ЯЗЫК КОМАНДНОГО ИНТЕРПРИТАТОРА SHELL

Цель работы: Освоение основ работы с языком командного интерпретатора Shell, изучение основных команд и возможностей Shell, а также развитие навыков написания простых скриптов для автоматизации задач в операционной системе Unix/Linux.

Основные теоретические сведения

Рассмотрим структуру скрипта Shell:

```
#!/bin/bash

# Комментарий: Начало скрипта

# Объявление переменных
VAR1="Hello"
VAR2="World"

# Основная часть скрипта
echo "$VAR1 $VAR2"   # Выводит "Hello World"

# Завершение скрипта
exit 0
```

- **Шебанг (Shebang)**: `#!/bin/bash` – это строка в начале скрипта, указывающая на то, какой интерпретатор использовать. В данном случае используется интерпретатор Bash.

- **Комментарии**: Комментарии помогают описать, что делает скрипт. Они начинаются с символа `#` и игнорируются интерпретатором.
- **Объявление переменных**: Здесь объявляются переменные `VAR1` и `VAR2`, которые содержат строки «Hello» и «World» соответственно.
- **Основная часть скрипта**: Это место, где выполняются основные действия. В данном случае, скрипт просто выводит содержимое переменных с помощью команды `echo`.
- **Завершение скрипта**: Команда `exit 0` указывает на успешное завершение скрипта. Код `0` обычно означает успешное выполнение, а другие значения могут указывать на различные ошибки или условия завершения.

Переменные в shell играют ключевую роль, позволяя хранить и использовать данные в скриптах. Переменные объявляются без использования ключевых слов, просто путем присваивания значения. Например:

```
VAR="value"
```

Для использования переменной в тексте скрипта нужно указать её имя, предваряя его символом доллара `$`. Например:

```
echo $VAR
```

В shell есть множество специальных переменных, таких как `$0`, `$1`, `$2`, и т. д., которые хранят аргументы командной строки. Например:

```
echo "Имя этого скрипта: $0"
```

Можно использовать команду `read`, чтобы прочитать значение из ввода и присвоить его переменной. Например:

```
read NAME
echo "Привет, $NAME!"
```

Для удаления переменной используется команда `unset`. Например:

```
unset VAR
```

Переменные могут быть интерполированы в строковых значениях. Например:

```
GREETING="Hello"
echo "$GREETING, world!"
```

Все переменные, которые вы экспортируете из shell, становятся переменными окружения для любых запущенных подкоманд или программ. Например:

```
export MY_VAR="value"
```

В bash и других расширениях shell можно использовать ассоциативные массивы для хранения данных. Например:

```
declare -A fruits
fruits[apple]="red"
fruits[banana]="yellow"
echo "Apple is ${fruits[apple]}"
```

Специальные переменные в shell представляют собой переменные, которые предопределены и предназначены для выполнения определенных задач.

- **$0**: имя исполняемого файла скрипта (имя команды).
- **$1, $2, ..., $N**: параметры командной строки. $1 содержит первый аргумент, $2 – второй, и так далее.
- **$@**: список всех аргументов командной строки.
- **$#**: количество аргументов командной строки.
- **$?**: код возврата (exit code) последней выполненной команды.
- **$$**: PID (идентификатор процесса) текущего shell.
- **$!**: PID последнего запущенного в фоновом режиме процесса.
- **$***: похоже на $@, но сохраняет аргументы как одну строку.
- **$IFS**: разделитель полей (Internal Field Separator). Это строка символов, которая определяет, где shell должен разбивать строки на поля, при чтении из ввода или из переменной.
- **$PWD**: текущий рабочий каталог (полный путь).
- **$OLDPWD**: предыдущий рабочий каталог.

Переменные окружения представляют собой переменные, которые могут быть установлены в системе операционной или виртуальной среде и доступны для всех процессов, запущенных в этой среде. Они могут содержать различную информацию о системе, пользователе, конфигурации и многое другое.

- **PATH**: содержит список каталогов, в которых операционная система ищет исполняемые файлы.
- **HOME**: путь к домашнему каталогу текущего пользователя.
- **USER**: имя текущего пользователя.
- **LANG**: определяет язык, используемый для отображения сообщений и форматирования даты и времени.
- **TERM**: определяет тип терминала, используемого для взаимодействия с пользователем.
- **PWD**: текущий рабочий каталог.
- **SHELL**: путь к оболочке, используемой текущим пользователем.
- **DISPLAY**: определяет, на каком дисплее должны отображаться графические приложения.
- **TMP, TEMP**: пути к временным каталогам.
- **EDITOR, VISUAL**: определяют текстовый редактор, используемый по умолчанию.
- **TZ**: определяет часовой пояс системы.
- **HOSTNAME**: имя хоста (имя компьютера).
- **PS1, PS2**: переменные, используемые для настройки приглашения командной строки.
- **LD_LIBRARY_PATH**: список каталогов, в которых операционная система ищет библиотеки, используемые программами.

- **JAVA_HOME**: путь к установленной JDK (Java Development Kit).

Эти переменные можно использовать в shell скриптах для доступа к информации о среде выполнения и для настройки поведения программы в соответствии с окружением. Для просмотра всех переменных окружения в вашей текущей среде выполнения вы можете использовать команду `env` в большинстве Unix-подобных систем.

Операторы ветвления в shell используются для принятия решений на основе условий и выполнения различных действий в зависимости от результатов этих условий.

- **if-then-else**:

```
if условие; then
    команды_если_условие_истинно
else
    команды_если_условие_ложно
fi
```

Пример:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
else
    echo "x не больше 10"
fi
```

- **if-then** (без else):

```
if условие; then
    команды_если_условие_истинно
fi
```

Пример:

```
if [ -e $FILE ]; then
    echo "$FILE существует"
fi
```

- **if-elif-else:**

```
if условие1; then
    команды_если_условие1_истинно
elif условие2; then
    команды_если_условие2_истинно
else
    команды_если_все_условия_ложны
fi
```

Пример:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
elif [ $x -eq 10 ]; then
    echo "x равно 10"
else
    echo "x меньше 10"
fi
```

- **case:**

```
case выражение in
    паттерн1)
        команды_если_паттерн1_соответствует
        ;;
    паттерн2)
        команды_если_паттерн2_соответствует
        ;;
    *)
        команды_если_нет_соответствия_ни_одному_паттерну
        ;;
esac
```

Пример:

```
case $VAR in
    "value1")
        echo "Переменная равна value1"
        ;;
    "value2")
        echo "Переменная равна value2"
        ;;
    *)
        echo "Переменная не равна ни value1, ни value2"
        ;;
esac
```

Эти операторы ветвления позволяют создавать гибкие скрипты, которые могут выполнять различные действия в зависимости от условий, и обеспечивают управление потоком выполнения программы в shell.

В контексте программирования «*предикат*» обычно означает выражение, которое оценивается как истинное или ложное. В языках программирования, включая shell, предикаты используются в операторах ветвления (например, в операторах `if`, `while`, `case` и других) для принятия решений на основе условий.

- **Сравнение чисел**:
  - `-eq`: равно
  - `-ne`: не равно
  - `-gt`: больше
  - `-lt`: меньше
  - `-ge`: больше или равно
  - `-le`: меньше или равно

Например:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
fi
```

- **Сравнение строк**:
  - `=`: равно
  - `!=`: не равно
  - `-z`: пустая строка

- `-n`: не пустая строка

Например:

```
if [ "$var" = "value" ]; then
    echo "Переменная равна value"
fi
```

- **Файловые предикаты**:
  - `-e`: файл существует
  - `-f`: обычный файл
  - `-d`: каталог
  - `-r`: доступен для чтения
  - `-w`: доступен для записи
  - `-x`: доступен для выполнения

Например:

```
if [ -e $FILE ]; then
    echo "$FILE существует"
fi
```

- **Логические операции**:
  - `-a`: логическое «и»
  - `-o`: логическое «или»
  - `!`: логическое «не»

Например:

```
if [ "$var" = "value1" -o "$var" = "value2" ]; then
    echo "Переменная равна value1 или value2"
fi
```

Эти предикаты позволяют создавать условия в операторах ветвления, которые позволяют программам принимать решения и выполнять различные действия на основе этих условий.

В shell скриптах часто используются циклы для повторения определенных действий определенное количество раз или до выполнения определенного условия.

- **Цикл while**:

```
while условие; do
    команды
done
```

Например:

```
counter=0
while [ $counter -lt 10 ]; do
    echo $counter
    ((counter++))
done
```

- **Цикл until**:

```
until условие; do
    команды
done
```

Например:

```
counter=0
until [ $counter -eq 10 ]; do
    echo $counter
    ((counter++))
done
```

- **Цикл for** (для перебора элементов в списке):

```
for переменная in список; do
    команды
done
```

Например:

```
for i in {1..5}; do
    echo $i
done
```

- **Цикл for** (для перебора элементов в массиве или списке):

```
for переменная in элемент1 элемент2 ... элементN; do
    команды
done
```

Например:

```
for color in red green blue; do
    echo "Color: $color"
done
```

- **Цикл select** (для создания интерактивного меню):

```
select переменная in список; do
    команды
done
```

Например:

```
select fruit in apple banana orange; do
    echo "Вы выбрали: $fruit"
    break
done
```

Эти циклы могут использоваться для автоматизации повторяющихся задач, обработки данных, итерации по спискам и многих других вещей в shell скриптах.

Вызов сторонних программ из shell скриптов – обычная практика, которая позволяет скриптам взаимодействовать с другими программами и выполнять различные задачи. Для вызова сторонней программы используются команды, которые могут быть исполнены внутри скрипта.

- **Вызов программы по её имени**:

```
программа_или_команда аргументы
```

Например:

```
ls -l
```

- **Использование переменных в качестве аргументов**:

```
переменная="аргументы"
программа_или_команда $переменная
```

Например:

```
directory="/path/to/directory"
ls $directory
```

- **Подстановка вывода другой команды**:

```
программа_или_команда $(другая_команда)
```

Например:

```
files_count=$(ls | wc -l)
echo "В текущем каталоге $files_count файлов"
```

- **Использование обратных кавычек для подстановки вывода команды** (устаревший метод):

```
программа_или_команда `другая_команда`
```

Например:

```
files_count=`ls | wc -l`
echo "В текущем каталоге $files_count файлов"
```

- **Пайплайны (|) для передачи вывода одной команды в качестве входных данных другой**:

```
команда1 | команда2
```

Например:

```
ls | grep ".txt"
```

- **Перенаправление вывода (stdout и stderr):**

```
программа_или_команда > файл
программа_или_команда >> файл (для добавления в конец файла)
```

Например:

```
ls > file_list.txt
```

- **Перенаправление ввода:**

```
программа_или_команда < файл
```

Например:

```
sort < unsorted_file.txt
```

Это основные способы вызова сторонних программ из shell скриптов. Они позволяют взаимодействовать со стандартными командами операционной системы, а также с любыми другими программами, доступными в вашей среде выполнения.

В shell скриптах можно выполнить математические вычисления несколькими способами.

- **Арифметическое выражение с помощью встроенной команды let:**

```
let "результат = выражение"
```

Например:

```
let "x = 10 + 5"
echo $x   # Выведет 15
```

- **Арифметическое выражение с помощью двойных круглых скобок ((...)):**

```
результат=$((выражение))
```

Например:

```
x=$((10 + 5))
echo $x   # Выведет 15
```

- **Использование команды expr:**

```
результат=$(expr выражение)
```

Например:

```
x=$(expr 10 + 5)
echo $x   # Выведет 15
```

- **Использование конструкции $((...)):**

```
результат=$((выражение))
```

Например:

```
x=$((10 + 5))
echo $x   # Выведет 15
```

- **Использование встроенной команды bc для вычисления выражений с плавающей запятой:**

```
результат=$(echo "выражение" | bc)
```

Например:

```
x=$(echo "10.5 * 3" | bc)
echo $x   # Выведет 31.5
```

Каждый из этих методов имеет свои особенности и подходит для разных типов вычислений. Например, `let` и `((...))` работают только с целыми числами, в то время как `bc` поддерживает вычисления с плавающей запятой.

В shell скриптах вы можете определять пользовательские функции для организации и структурирования вашего кода. Вот пример определения и вызова пользовательской функции:

```
# Определение функции
function my_function {
    echo "Привет, это моя пользовательская функция!"
}
```

```
# Вызов функции
my_function
```

Этот пример определяет функцию с именем `my_function`, которая выводит приветственное сообщение. После определения функции она вызывается с помощью `my_function`.

Ниже приведены некоторые дополнительные концепции, которые могут быть полезны при работе с пользовательскими функциями:

1. **Передача аргументов в функцию**:

```
function my_function {
    echo "Привет, $1!"
}

my_function "Мир"
```

Этот пример передает строку «Мир» в функцию `my_function`, которая выводит «Привет, Мир!».

2. **Возврат значений из функции:**

```
function add_numbers {
    local result=$(( $1 + $2 ))
    echo $result
}

sum=$(add_numbers 5 3)
echo "Сумма: $sum"
```

Этот пример определяет функцию `add_numbers`, которая принимает два аргумента, складывает их и возвращает результат. Результат сохраняется в переменной `sum`.

3. **Локальные переменные в функции:**

```
function my_function {
    local name="John"
    echo "Привет, $name!"
}

my_function
echo "Имя: $name"  # Это вызовет ошибку, потому что переменная name является локальной для функции
```

В этом примере переменная `name` является локальной для функции `my_function` и недоступна за её пределами.

4. **Рекурсивные функции:**

```
function factorial {
    if [ $1 -le 1 ]; then
        echo 1
    else
        local prev=$(factorial $(( $1 - 1 )))
        echo $(( $1 * $prev ))
    fi
```

```
}

result=$(factorial 5)
echo "Факториал 5: $result"
```

В этом примере определена рекурсивная функция для вычисления факториала числа. Функция вызывает саму себя до тех пор, пока не достигнет базового случая.

Порядок выполнения работы

1. Переменные Shell:

○ Создать скрипт, который запрашивает у пользователя его имя и приветствует его с использованием этого имени.

○ Написать скрипт, который запрашивает у пользователя два числа и выводит их сумму.

2. Переменные окружения:

○ Написать скрипт, который выводит значения некоторых системных переменных окружения, таких как `$HOME`, `$PATH`, `$USER`, и т. д.

○ Создать скрипт, который выводит список всех переменных окружения.

3. Операторы ветвления:

○ Написать скрипт, который проверяет, является ли введенное пользователем число четным или нечетным, и выводит соответствующее сообщение.

○ Создать скрипт, который запрашивает у пользователя его возраст и сообщает, является ли он совершеннолетним или нет.

4. Оператор выбора:

○ Написать скрипт, который запрашивает у пользователя его любимый сезон и выводит сообщение о том, какие виды активностей связаны с этим сезоном.

○ Создатть скрипт, который проверяет введенную пользователем строку на наличие ключевого слова и выводит соответствующее сообщение.

5. Операторы циклов:

○ Написать скрипт, который выводит числа от 1 до 10 с использованием цикла `for`.

○ Создать скрипт, который запрашивает у пользователя число и выводит таблицу умножения этого числа.

6. Создание и использование пользовательских функций:

○ Создать функцию `greet`, которая принимает имя в качестве аргумента и выводит приветствие.

○ Вызвать функцию `greet` со своим именем.

7. Использование команд для работы с файлами:

○ Создать текстовый файл `example.txt`.

○ Записать в него строку «Пример текста».

○ Использовать команду `cat` для чтения содержимого файла `example.txt`.

Контрольные вопросы

1. *Основы работы с командной строкой:*
   - Какие команды используются для навигации по файловой системе в Shell?
   - Как узнать текущий рабочий каталог в командной строке?
   - Какие команды используются для управления файлами и каталогами?
2. *Перенаправление ввода/вывода и конвейеры:*
   - Какие символы используются для перенаправления вывода команды в файл?
   - Чем отличаются символы > и >> при перенаправлении вывода?
   - Что такое конвейер (pipe) в Shell, и как он используется?
3. *Написание скриптов на Shell:*
   - Как объявить переменную в Shell скрипте?
   - Как передать аргументы в скрипт при его вызове?
   - Какие операторы управления используются в Shell скриптах, и как они работают?
4. *Управление процессами:*
   - Как узнать список запущенных процессов в системе?
   - Как отправить сигнал процессу для его завершения?
   - Как перевести процесс в фоновый режим или вернуть его в передний план?
5. *Безопасность и права доступа:*
   - Как изменить права доступа к файлу или каталогу в Shell?
   - Как изменить владельца и группу файла или каталога?
   - Какие команды используются для работы с пользователями и группами в Unix/Linux?
6. *Практические навыки:*
   - Как написать скрипт, который будет искать все файлы в заданном каталоге с расширением «.txt» и выводить их список?
   - Как создать скрипт, который будет резервировать копию заданного каталога и его содержимого?
   - Как можно использовать командный интерпретатор для автоматизации рутинных задач на вашем компьютере?