LINUX SHELL PROGRAMMING

Operating System Sessional : CSE 308

Shell Program

- A shell program is nothing but a series of commands.
- Instead of specifying one job at a time, we give the shell a to-do list – a program – that carries out an entire procedure.
- Such programs are known as 'Shell Script'.
- The shell scripts offer new horizons to computing prowess, combining the collective power of various commands and the versatility of programming language.

Shell Program (cont.)

- The shell programming language incorporates most of the features that most modern day programming language offer.
- For example, it has local and global variables, control instructions, functions etc.
- If we are to execute a shell program we don't need a separate compiler.
- The shell it self interprets the commands in the shell program and executes them.

When to use Shell Scripts?

- **Customizing work environment.**
  - For example, every time you log in if you want to see the current date, a welcome message and the list of users who have logged in you can write a shell script for the same.
- **Automating daily tasks.**
  - For example, you may want to back up all your programs at the end of the day. This can be done using a shell script.
- **Automating repetitive tasks.**
  - For example, the repetitive task of compiling a C program, linking it with some libraries and executing the executable code can be assigned to a shell script.

When to use Shell Scripts? (cont.)

- **Executing important system procedures.**
  - For Example, shutting down the system, formatting a disk, creating a file system on it, mounting the file system, letting the users use the floppy and finally unmounting the disk.

- **Performing same operation on many files.**
  - For example, you may want to replace a string ***printf*** with a string ***myprintf*** in all the C programs present in a directory.

When you should not use Shell Program?

- If the task is too complex, such as writing an entire billing system.
- If the task requires a high degree of efficiency.
- If the task requires a variety of software tools.

A Simple Shell Script

- To execute the script -> ./SS1
- To change permission -> chmod 744 SS1

Interactive Shell Scripts

- To accept input: ***read***
- To display output: ***echo***

SS2

```
echo what is your name\?
read name
echo Hello $name. Good Morning
```

- The question mark '?' must be preceded by a backslash '\' to convey to the shell that here the '?' is to be treated as an ordinary character.

Shell Variables

- Shell variables provide the ability to store and manipulate information within a shell program.
- You can create and destroy any number of variables as needed to solve the problem.
- The rules for building shell variables are as follows:
  - A variable name is any combination of alphabets, digits and an underscore.
  - No commas or blanks are allowed within a variable name.
  - The first character of a variable name must either be an alphabet or an underscore.
  - Variable names should be of any reasonable length.
  - Variable names are case sensitive.

Shell Keywords

echo	if	until	trap
read	else	case	wait
set	fi	esac	eval
unset	while	break	exec
readonly	do	continue	ulimit
shift	done	exit	unmask
export	for	return	

Assigning values to variables

- Values can be assigned to shell variables using a simple assignment operator.

SS2

```
name=sunny
age=20
dirname=/usr/aa5
echo $name $age $dirname
```

- While assigning values to variables using the assignment operator '=', there should be no spaces on either side of '='.
- Otherwise the shell will try to interpret the value being assigned as a command to be executed.

User-defined Variables

- The variable length can be of any reasonable length and may constitute alphabets, digits and underscore.
- The first character must be an alphabet or an underscore.

SS2

```
a=20
echo $a
echo a
```

- The $ causes the value of a to get displayed.
- Omitting the $ would simply treat **a** as a character to be echoed.

Tips and Traps

- All shell variables are string variables. In the statement a=20, the '20' stored in a is treated not as a number, but as a string of characters 2 and 0.
- A variable may contain more than one word. In such cases, the assignment must be made using double quotes.
  - C="two words"
- We can carry out more than one assignment in a line. We can echo more than one variable's value at a time.
  - Name=Jony age=10
  - Echo $name $age
- All variables defined inside a shell script die the moment the execution of the script is over.

Arithmetic in Shell Script

SS4

```
a=20
b=10
echo `expr $a + $b`
echo `expr $a - $b`
echo `expr $a \* $b`
echo `expr $a / $b`
echo `expr $a % $b`
```

Arithmetic in Shell Script (cont.)

SS4

```
a=10.5
b=3.5
c=`echo $a + $b | bc`
d=`echo $a - $b | bc`
e=`echo $a \* $b | bc`
f=`echo $a / $b | bc`
echo $c $d $e $f
```

Taking Decisions

- The Bourne shell offers four decision making instructions. They are:
  - *if-then-fi* statement
  - *if-then-else-fi* statement
  - *if-then-elif-then-fi* statement
  - *case-esac* statement

if-then-fi statement

SS1

```
echo Enter Source and Target File name
read source target
if cp $source $target
then
        echo file copied
fi
```

if-then-else-fi statement

SS1

```
echo Enter Source and Target File name
read source target
if cp $source $target
then
        echo file copied
else
        echo file not copied
fi
```

The *test* command

SS1

```
echo Enter a number from 1 to 10
read num
if test $num –lt 6
then
        echo Number is less than 6
else
        echo Number is greater than 5
fi
```

The *test* command (cont.)

- The test command can carry out several types if tests. These are:
  - Numerical Tests
  - String Tests
  - Files Tests

The *test* command (cont.)

- Numerical Tests

Arithmetic Comparison	Result
expression1 -eq expression2	True if the expressions are equal
expression1 -ne expression2	True if the expressions are not equal
expression1 -gt expression2	True if the expression1 is greater than expression2
expression1 -ge expression2	True if the expression1 is greater than or equal to expression2
expression1 -lt expression2	True if the expression1 is less than expression2
expression1 -le expression2	True if the expression1 is less than or equal to expression2

The *test* command (cont.)

SS1

```
echo “Insert Two Numbers:”
read num1
read num2
if  test  $num1 -ge $num2
then
        echo “Maximum is $num1”
else
        echo “Maximum is $num2”
fi
```

Nested *if-else*

SS1

```
echo Enter either 1 or 2
read i
if [ $i –eq 1 ]
then
        echo Odd number
else
        if [ $i –eq 2 ]
        then
                echo even number
        else
                echo go to hell
        fi
fi
```

Use of logical operators

- Shell allows usage of three logical operators while performing a test. These are:
  - -a (read as AND)
  - -o (read as OR)
  - ! (read as NOT)

Hierarchy of Operartors

- The higher an operator is in the table, the higher is its priority.

Operators	Type
!	Logical Not
-lt, -gt, -le, -ge, -eq, -ne	Relational
-a	Logical AND
-o	Logical OR

Case control Structure

SS1

```
echo “is it morning?”
read timeofday
case “$timeofday” in
        yes) echo “Good Morning”;;
        no ) echo “Good Afternoon”;;
        y   ) echo “Good Morning”;;
        n   ) echo “Good Afternoon”;;
        *   ) echo “Unrecognized”;;
esac
exit 0
```

Case control Structure (cont.)

SS1

```
echo “is it morning?”
read timeofday
case “$timeofday” in
        yes | y) echo “Good Morning”;;
        no | n ) echo “Good Afternoon”;;
        *   ) echo “Unrecognized”;;
esac
exit 0
```

The Loop Control Structure

- There are three methods by way of which we can repeat a part of a program. They are:
  - ***for*** statement
  - ***while*** statement
  - ***until*** statement

The *while* Loop

Syntax

```
while control command
do
   statement 1
   statement 2
done
```

The *while* Loop (cont.)

SS1

```
count=1
while [ $count  –le  3 ]
do
        echo $count
        count=`expr $count + 1`
done
```

The *until* Loop

Syntax:

```
until control command
do
    statement
    statement
done
```

The *while* Loop (cont.)

SS1 – prints number from 1 to 10

```
i=1
until [ $i  -gt  10 ]
do
        echo $i
        i=‘expr $i + 1’
done
```

The *for* Loop

Syntax:

```
for (( ____;_____;_____ ))
do
   statement
   statement
done
```

Nesting for loop

SS1

```
r=1
while [ $r  -le  3 ]
do
        c=1
        while [ $c  -le  2  ]
        do
                sum=`expr $r + $c`
                echo r=$r   c=$c  sum=$sum
                c=`expr $c + 1`
        done
        r=`expr $r +  1`
done
```

The *break* statement

SS1

```
r=1
while [ $r  -le  3 ]
do
        c=1
        if [ $c  -le  2  ]
        then
                break
        fi
        c=`expr $c + 1`
        r=`expr $r +  1`
done
```

The *continue* statement

SS1

```
r=1
while [ $r  -le  3 ]
do
        c=1
        if [ $c  -le  2  ]
        then
                continue
        fi
        c=`expr $c + 1`
        r=`expr $r +  1`
done
```