

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 process, 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 itself interprets the commands in the shell program and executes them.



When to use Shell Scripts?

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**

chmod +x SS1



Interactive Shell Scripts

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

```
SS1
```

```
echo what is your name\?
read $name
echo Hello $name
```

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
unmask	exit	done	shift
export	for	return	

Assigning values to variables

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

```
name=sunny  
age=50  
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.

```
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.



Array Variable

Array_name[indexnumber] = value

Array_name=(value1 value2 ... valueN)



Array Variable(Cont..)

```
for((i=0;i<=5;i++))  
do  
a[i]=$(expr $i + 1)  
done  
echo "${a[@]}"      // Display all the array value  
echo "${!a[@]}"     // Display all the array index  
echo "${#a[@]}"     // Display total number of element.
```

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

```
a=20  
b=10  
echo $(( a + b ))  
echo $(( a - b ))  
echo $(( a * b ))  
echo $(( a / b ))  
echo $(( a % b ))
```

```
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.)

```
a=20.5
```

```
b=3.25
```

```
echo "$a + $b" | bc  
echo "$a - $b" | bc  
echo "$a * $b" | bc  
echo "$a / $b" | bc  
echo "$a % $b" | bc
```

```
a=20.5
```

```
b=3.25
```

```
echo "$a + $b" | bc  
echo "$a - $b" | bc  
echo "$a * $b" | bc  
echo "scale=2; $a / $b" | bc  
echo "$a % $b" | bc
```



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

Comparison Operator



Integer Comparison

-eq	is equal to
-ne	is not equal to
-gt	is greater than
-ge	is greater than or equal to
-lt	is less than

-le	is less than or equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to



String Comparison

=	is equal to
==	is equal to
!=	is not equal to
<	is less than, in ASCII alphabetical order
>	is greater than, in ASCII alphabetical order
-z	string is null. That is, has zero length



if-then-fi statement

```
if [ <some test> ]
```

```
then
```

```
<commands>
```

```
fi
```

```
if [ $count -eq 10 ]
```

```
then
```

```
echo Count is equal to 10
```

```
fi
```



if-then-else-fi statement

```
if [ <some test> ]
```

```
then
```

```
<commands>
```

```
else
```

```
<commands>
```

```
fi
```

Case control Structure



```
case expr in
first_case )
    body of first case
    ;;
Second_case )
    body of first case
    ;;
*
)
    body of first case
    ;;
esac
```

Case control Structure(Cont..)

```
read count
case $count in
8 )
echo Greater than 10 ;;
2 )
echo Less than 10 ;;
* )
echo This is default ;;
esac
```

```
read count
case $count in
"yes" | "y")
echo This is yes;;
"No" | "n")
echo This is no;;
* )
echo This is default;;
esac
```



The test command

```
count=10
```

```
if test $count -eq 10
```

```
then
```

```
echo Count is equal to 10
```

```
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)



Use of logical operators(Cont..)

```
balance=25
if [ $balance -gt 20 -a $balance -lt 30 ]
then
    echo valid balance
else
    echo invalid
fi
```



Use of logical operators(Cont..)

```
read -p "Enter Your Number:" i
```

```
if [ $i -ge 10 -a $i -le 20 -o $i -ge 100 -a $i -le 200 ]
then
    echo "OK"
else
    echo "Not OK"
fi
```



Hierarchy of Operators

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



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..)

```
a=0
```

```
while [ $a -lt 10 ]
do
    echo $a
    a=$((expr $a + 1))
done
```



The until Loop

Syntax

until control command

do

Statement 1

Statement 2

done



The until Loop(Cont..)

```
a=0
```

```
until [ $a -gt 10 ]
do
    echo $a
    a=$((expr $a + 1))
done
```



The for Loop

Syntax

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



The for Loop(Cont..)

```
max=10
```

```
for (( i=2; i <= $max; ++i ))
```

```
do
```

```
    echo "$i"
```

```
done
```



*Thank
You*