UNIX Shell Programming

Dr. Manmath N. Sahoo
Dept. of CSE, NIT Rourkela

The Shell

The shell acts as an interface between the user and the UNIX system, works as a command interpreter. Analogous to command.com in DOS.

Commonly used shell in most variant of UNIX are:

- Bourne Shell (sh), first shell developed for UNIX
- Bourne Again Shell (bash), written by programmers of Free Software Foundation, open source shell from GNU
- Korn Shell (ksh), written by David Korn, superset of Bourne shell, not widely distributed.
- C Shell (csh), written by Bill Joy, the author of vi, shared much of the C language structure.
- Terminal Based C Shell (tcsh), enhanced version of the UNIX C shell csh

Shell scripts

- A shell script is a text file with Unix commands in it.
- Shell scripts usually begin with a #! and a shell name (complete pathname of shell).
  - Pathname of the current shell can be found using the ***echo $SHELL or echo $0*** command at the shell prompt
  - The shell name is the shell that will execute this script.
    - e.g: #!/bin/bash
- If no shell is specified in the script file, the default is chosen to be the currently executing shell (login shell).

Shell scripts

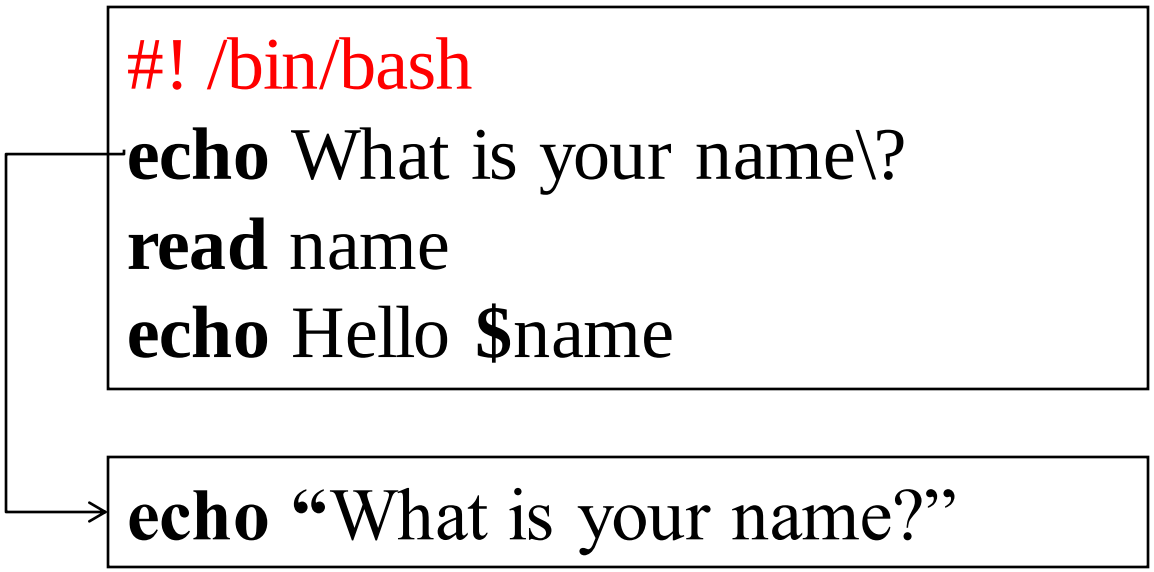
- Write the following code into ***myscript.sh***

```
#! /bin/bash
pwd
ls
```

- To run the script:
  - $ bash myscript.sh

Interactive shell scripts

- read – accept input
- echo – display output



Shell scripts

- Shell variables:
  - Case sensitive
  - Declared by:

    varname=varvalue
  - There should be no spaces on either side of =
  - If the variable does not exist then it will be created automatically during assignment.
  - All shell variables are string variables.

```
#! /bin/bash
a=20
echo $a
```

Shell scripts

- Assigning the output of a command to a variable:
  - Using accent graves, we can assign the output of a command to a variable:

```
#! /bin/bash
filelist=`ls`
echo $filelist
```

Shell scripts

- Variable containing more than one word:
  - a=“Two Words”
- More than one assignment in a line:
  - name=Johny age=25
  - echo $name $age
- Declared only variable or null variable:
  - **d=“”** *or* **d=‘’** *or* **d=**
- Constant variable:

```
a=20
readonly a
```

Positional parameters & command line arguments

- The shell defines 9 positional parameters, **$1,$2,…,$9,** to accept command line args.
- $0 is for the program/script name.
- echo $# gives the no of arguments passed
- e.g. **$bash myscript.sh A smiling face is always beautiful**
- $1=A, $2=smiling, $3=face, $4=is, $5=always, $6=beautiful

Positional parameters & command line arguments

- $bash myscript.sh You have the capacity to learn from mistakes. You will learn a lot in your life

  echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11

```
You have the capacity to learn from
mistakes. You You0 You1
```

echo $1 $2 $3 $4 $5 $6 $7 $8 $9

Shift 2

echo $8 $9 → will learn

Set positional parameters in script

set dept of cs

echo $1 $2 $3 → dept of cs

Shell scripts

- The **expr** command:
  - Calculates the value of an expression.
  - e.g:

```
#!/bin/bash
a=20   b=10
echo `expr $a + $b`
echo `expr $a - $b`
echo `expr $a \* $b`
echo `expr $a / $b`
echo `expr $a % $b`
echo `expr $a+$b` →20+10
echo `expr $a$b` →2010
```

```
count=5
count=`expr $count + 1`
echo $count
```

```
a=20.5 b=10
echo `expr $a + $b` →error
```

- Terms should be separated by spaces
- Performs integer arithmetic only.

Shell scripts

- basic calculator (bc)

```
a=20.5 b=10
echo “$a+$b” | bc →30.5
echo “$a-$b” | bc →10.5
echo “$a*$b” | bc →205
echo “$a/$b” | bc →2
echo “$a%$b” | bc →0.5
echo “scale=2;$a/$b” | bc →2.05
echo “scale=2;$a%$b” | bc →0
echo “scale=1;$a/$b” | bc →2.0
echo “scale=1;$a%$b” | bc →0.5
```

- Terms need not be separated by spaces
- Performs integer and floating point arithmetic.

Control statements

- The two most common types of control statements:
  - conditionals: if/then/else, case, ...
  - loop statements: while, for, until, ...

for loops

- *for* loops allow the repetition of a command for a specific set of values.
- Syntax:

```
for var in value1 value2 ...
do
   command_set
done
```

  - command_set is executed with each value of var (value1, value2, ...) in sequence

for loops

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo $i
done
```

```
1
2
3
4
5
```

```
#!/bin/bash
for i in 1 two 3 4 5
do
  echo $i
done
```

```
1
two
3
4
5
```

Notes on *for*

- Example: Listing all files in a directory.

```
#! /bin/bash
for i in *
do
  echo $i
done
```

NOTE: * is a wild card that stands for all files in the current directory, and *for* will go through each value in *, which is all the files and $i has the filename.

Conditionals

- Conditionals are used to “test” something.
  - In Java or C, they test whether an expression is true or false.
  - In a Bourne shell script, the only thing you can test is whether or not a command is “successful”.

Conditionals

- Every valid unix command returns back a return code
  - 0 if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)
  - This is opposite to ‘C’.

The *if* command

```
if decision_command_1
then
      command_set_1
fi
```

Example

grep returns 0 if it finds the specified pattern
returns non-zero otherwise
If successful it also prints the lines containing the string **there**

```
if  grep there a.sh  > result_grep
then
  echo "It's there"
else
  echo "It's not there"
fi
```

redirect to result_grep so that intermediate results do not get printed

Using *elif* with *if*

```
#! /bin/bash
if grep “UNIX” myfile > result_grep
  then
    echo  UNIX occurs in myfile
elif grep “DOS” myfile > result_grep
  then
    echo DOS appears in myfile not UNIX
  else
  echo  neither UNIX nor DOS appears in myfile
fi
```

Do nothing operation

- Sometimes, we don't want a statement to do anything
  - In that case, use a colon '**:**'

```
if grep UNIX myfile > result_grep
then
:
fi
```

Does not do anything when UNIX is found in *myfile* .

The *test* command

- Used to check validity.
- Three ways of using *test*:
  - Check on files.
  - Check on strings.
  - Check on integers

Testing on files

Note space after [and before]

- if test –e file: does file exist? → if [-e file]
- if test –f file: does file exist and is a file?
- if test -d file: does file exist and is a directory?
- if test –r file: does file exist and is readable?
- if test –w file: does file exist and is writeable?
- if test –x file: does file exist and is executable?
- if test –s file: file is not 0 size?
- if test f1 -nt f2: file f1 is newer than f2?
- if test f1 -ot f2: file f1 is older than f2?
- ! Reverses the sense of the tests
- if test ! –e file: successful if file doesn't exist

Example

```
#!/bin/bash
count=0
for i in *
do
        if test –x $i          # if [ -x $i ]
        then
                count=`expr $count + 1`
        fi
done
echo Total of $count files executable
```

Testing on strings

a=hello b=HELLO

- if [$a = $b] :is equal to
- if [$a != $b] :is not equal to
- if [$a \< $b] :less than
- if [$a \> $b] :greater than
- if [-z $a] :string has zero length?
- if [-n $a] :string has NON zero length?

Testing on integers

a=20 b=10

- test $a –eq b : is a equal to b ?
- test $a –ne b : is a not equal to b?
- test $a –lt b : is a less than to b?
- test $a –gt b : is a greater than to b?
- test $a –le b : is a less than or equal to b?
- test $a –ge b : is a greater than or equal to b?

Example

```
#!/bin/bash
i=10
j=20
if test $i -lt $j
then
  echo $i
else
  echo $j
fi
```

[$i -lt $j]

The *while* loop

- While loop repeats statements as long as the next Unix command is successful.

```
#! /bin/bash
i=1
sum=0
while [ $i -le 100 ]
do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

The *until* loop

- Until loop repeats statements until the next Unix command is successful.

```
#! /bin/bash
x=1 sum=0
until [ $x -gt 3 ]
do
  sum=`expr $sum + $x`
  x=`expr $x + 1`
done
```

case…esac statement

```
#!/bin/bash
echo "Enter a number between 1 and 4. "
read NUM
case $NUM in
1)    echo "one" ;;
2)    echo "two" ;;
3)    echo "three" ;;
4)    echo "four”;;
*)    echo "INVALID NUMBER!" ;;
esac
```

Array

```
arr=( 1 two 3 four 5)
for i in 0 1 2 3 4
do
        echo ${arr[$i]}
done
```

Output:

1
two
3
four
5

Array

```
arr=( 1 two 3 four 5)
echo ${arr[*]}
for i in ${arr[@]}   # ${arr[*]}
do
        echo $i
done
```

Output:

```
1 two 3 four 5
1
two
3
four
5
```

Array

```
echo “Enter array elements  separated by space”
read –a arr
for i in ${arr[@]}
do
        echo $i
done
```

Output:

Enter array elements separated by space

1 two 3 four 5

1

two

3

four

5