Shell Basics

Shells

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

Shell Script

- A shell script is just a file containing shell commands, but with a few extras:
  - The first line of a shell script should be a comment of the following form:
    - `#!/bin/sh`
- for a Bourne shell script. Bourne shell scripts are the most common, since C Shell scripts have buggy features.
- A shell script must be readable and executable.
  - `chmod u+rx scriptname`
- As with any command, a shell script has to be “in your path” to be executed.
- If “.” is not in your PATH, you must specify “`./scriptname`” instead of just “`scriptname`”

Shell Script Example

- Here is a “hello world” shell script:
  - `$ ls -l`
  - `-rwxr-xr-x  1 <user>   48 Feb 08 19:50 hello*`
  - `$ cat hello`
  - `#!/bin/sh`
  - `# comment lines start with the # character`
  - `echo "Hello world"`
  - `$ hello`
  - `Hello world`
  - `$`

- The `echo` command functions like a print command in shell scripts.

Shell Variables

- The user variable name can be any sequence of letters, digits, and the underscore character, but the first character must be a letter.

- To assign a value to a variable:
  - `number=25`
  - `name="ABC XYZ"`

- There cannot be any space before or after the “=“

- Internally, all values are stored as strings.

Shell Variables

- To use a variable, precede the name with a "$":
  - `$ cat test1`
  - `#!/bin/sh`
  - `number=25`
  - `name="XYZ ABC"`
  - `echo "$number $name"`
  - `$ test1`
  - `25 XYZ ABC`
  - `$`

Shell Variables

- Use the `read` command to get and store input from the user.

```
$ cat test2
#!/bin/sh
echo "Enter name: "
read name
echo "Enter Course: "
read number
echo “Enter Year: ”
read y
echo "$name is studying in
$number $y year"
```

```
$ test2
Enter name:
XYZ
Enter Course:
B.Tech.
Enter Year
3rd
XYZ is studying in B.Tech. 3rd year
```

Variables in Shell

- In UNIX (Shell), there are 2 types of variables:
  - **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
  - **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in {(a-z)+(0-9)+_}* not starting with a digit.

Variables in Shell (cntd…)

Defining Variables

- variable_name=variable_value

- Variables of this type are called scalar variables. A **scalar** variable can hold only one value at a time.
- The shell enables us to store any value you want in a variable.
  - VAR1="Zara Ali"
  - VAR2=100

Variables in Shell (cntd…)

Accessing Variable Values

- To access the value stored in a variable, prefix its name with the dollar sign ($)
  - For example, following script would access the value of defined variable NAME and would print it on STDOUT:

```
#!/bin/sh
NAME="XYZ"
 echo $NAME
```

This would produce following value:

```
XYZ
```

Variables in Shell (cntd…)

Read-only Variables

- The shell provides a way to mark variables as read-only by using the readonly command.
- After a variable is marked read-only, its value cannot be changed.

```
#!/bin/sh
NAME="DEEKAY"
readonly NAME
NAME="DIVYA"
```

This would produce following result: /bin/sh: NAME: This variable is read only.

Variables in Shell (cntd…)

Unsetting Variables

- Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks.
- Once you unset a variable, you would not be able to access stored value in the variable.

```
#!/bin/sh
NAME="XYZ"
unset NAME
echo $NAME
```

Above example would not print anything.

>>You cannot use the unset command to **unset** variables that are marked **readonly**.

Variables in Shell (cntd...)

Null Variables

- You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition)

  For e.g.
  $ vech=
  $ vech=""
  Try to print it's value by issuing following command:
  $ echo $vech
  Nothing will be shown because variable has no value i.e. NULL variable.

Variables in Shell (cntd…)

Special Variables

- That we can't use as normal variables.

Variables	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.

Variables in Shell (cntd…)

Special Variables

Variables	Description
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Variables in Shell (cntd…)

- **Special Variables**
  - The command-line arguments $1, $2, $3,…$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, …$9 as the arguments to the command.

Variables in Shell (cntd…)

```
#!/bin/sh
echo "File Name: $0"
echo "First Parameter : $1"
echo “Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*”
echo "Total Number of Paramers : $#”
```

OUTPUT

```
./test.sh divya kumar
File Name : ./test.sh
First Parameter : divya
Second Parameter : kumar
Quoted Values: divya kumar
Quoted Values: divya kumar
Total Number of Paramers : 2
```

Variables in Shell (cntd…)

- **Special Parameters $* & $@**
  - The "$*" special parameter takes the entire list as one argument with spaces between.
  - The "$@" special parameter takes the entire list and separates it into separate arguments.

Variables in Shell (cntd…)

- **Special Parameters $* & $@**

```
#!/bin/sh
for TOKEN in $*
do
echo $TOKEN
done
```

OUTPUT

```
./test.sh divya kumar 25 Years Old
divya
kumar
25
Years
Old
```

EXIT Status

- The **$?** variable represents the exit status of the previous command.
- Exit status is a numerical value returned by every command upon its completion.
- As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Special Characters & Quoting

- List of Special Characters and what they mean

<RETURN> Execute command

\# Start a comment

<SPACE> Argument separator

\` Command substitution

\" Weak Quotes

\' Strong Quotes

\\ Single Character Quote

Special Characters & Quoting (cntd…)

- List of Special Characters and what they mean

&	Run program in background
?	Match one character
*	Match any number of characters
;	Command separator
;;	End of Case statement

Special Characters & Quoting (cntd…)

- List of Special Characters and what they mean

~ Home Directory

~user User's Home Directory

! History of Commands (csh only)

$# Number of arguments to script

$* Arguments to script

$@ Original arguments to script

Special Characters & Quoting (cntd…)

- List of Special Characters and what they mean

$?	Status of previous command
$$	Process identification number
$!	PID of last background job
&&	Short-circuit AND
||	Short-circuit OR
[ ]	Match range of characters OR Test

Special Characters & Quoting (cntd…)

- List of Special Characters and what they mean

(cmd;cmd)	Runs cmd;cmd as a sub-shell
{cmd;cmd }	Runs cmd;cmd without subshell
>file	Output to
>>file	Append output to
<file	Input from

Special Characters & Quoting (cntd…)

- **Quoting** is used to remove the special meaning of certain characters or words to the shell.
- Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Special Characters & Quoting (cntd…)

- **Backslash (\)**

  Any character immediately following the backslash loses its special meaning.

Special Characters & Quoting (cntd…)

- **Single quote (‘)**

  All special characters between these quotes lose their special meaning.

```
echo <-$1500.**>; (update?) [y|n]
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
echo '<-$1500.**>; (update?) [y|n]'
```

Special Characters & Quoting (cntd…)

- **Single quote (‘)**

>> HOW TO CORRECTLY WRITE…??

echo divya’s book

Special Characters & Quoting (cntd…)

- **Double quote (“)**

  Enclosing characters in **double quotes** preserves the literal value of all characters within the quotes, with the exception of

  $, `, and \.

Special Characters & Quoting (cntd…)

- **Double quote (“)**

  $ for parameter substitution.

  Backquotes for command substitution.

  \$ to enable literal dollar signs.

  \` to enable literal backquotes.

  \" to enable embedded double quotes.

  \\ to enable embedded backslashes.

Special Characters & Quoting (cntd…)

```
VAR=BOB
echo '$VAR owes -$15; [ on (`date +%m/%d`)]'
>>>>>$VAR owes -$15; [ on (`date +%m/%d`)]

echo “$VAR owes -$15; [ on (`date +%m/%d`)]”
>>>>> BOB owes -$15; [ on (03/05) ]
```

Operators in Shell

- There are following operators which we are going to discuss:
- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

Operators in Shell (cntd…)

- **Arithmetic Operators**
  - Assume variable a holds 10 and variable b holds 20 then:
- + `expr $a + $b` will give 30
- - `expr $a - $b` will give -10
- * `expr $a * $b` will give 200
- / `expr $b / $a` will give 2
- %`expr $b % $a` will give 0 (modulus or reminder)
- = a=$b will assign value of b into a (Assignment)
- == [$a == $b] would return false (equality)
- != [$a != $b] would return true (inequality)

  - Mind the space for the last 2

Operators in Shell (cntd…)

- **Relational Operators**
  - Assume variable a holds 10 and variable b holds 20 then:
- -eq [$a -eq $b] is not true.
- -ne [$a -ne $b] is true.
- -gt [$a -gt $b] is not true.
- -lt [$a -lt $b] is true.
- -ge [$a -ge $b] is not true.
- -le [$a -le $b] is true.

Operators in Shell (cntd…)

- **Relational Operators**

Example:

```
#!/bin/sh
a=10 b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b“
fi
```

Operators in Shell (cntd…)

- **Boolean Operators**

  –Assume variable a holds 10 and variable b holds 20 then:

  ! This is logical negation.

  [! false] is true.

  -o This is logical OR.

  [$a -lt 20 -o $b -gt 100] is true.

  -a This is logical AND.

  [$a -lt 20 -a $b -gt 100] is false.

Operators in Shell (cntd…)

- **String Operators (=, !=)**

Example:

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
# vice versa for !=
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi
```

Operators in Shell (cntd…)

```
if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
Fi
```

Operators in Shell (cntd…)

- **File Test Operators**
- Example

```
#!/bin/sh

file="somedir/test.sh"
if [ -r $file ]
then
   echo "File has read access"
else
   echo "File does not have read access"
fi
```

Operators in Shell (cntd…)

```
if [ -w $file ]
#similar for execute x
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi
```

Operators in Shell (cntd…)

```
if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file“
fi
```

Operators in Shell (cntd…)

---

d for directory;

s for size gt zero;

e for file exists;

b for block file;

c for character file;

Extra thoughts

- More operators in csh:

  << >>

  & |

  && ||

  ++

  --

  -o file (if USER owns the file)

  <op>=

  History lists

Arrays

- We can use a single array to store all the above mentioned names. This is expressed as follows:

NAME[0]="ABC"

NAME[1]="DEF"

NAME[2]="ghi"

NAME[3]="Jkl"

NAME[4]="XYZ"

Arrays (cntd…)

Another Syntax for defining arrays

array_name=(value1 … valuen)

Example:

array=(zero one two three four five)

Element 0 1 2 3 4 5

Arrays (cntd...)

Yet another Syntax for defining arrays

```
array=( [0]="first element" [1]="second
    element" [3]="fourth element" )
```

Arrays (cntd…)

Fetching the values

Expression	Meaning
${array[0]}	Value of first element
${array:1}	Parameter extension from first character
${#array[0]} or ${#array}	Length of first element
${#array[*]}	Number of elements in array
${#array[@]}	

Arrays (cntd…)

Fetching the values

Expression	Meaning
${array[0]}	Value of first element
${array:1}	Parameter extension from first character
${#array[0]} or ${#array}	Length of first element
${#array[*]}	Number of elements in array
${#array[@]}	

Arrays (cntd…)

String operations on arrays

```
#!/bin/bash
arrayZ=( one two three four five five )

# Trailing Substring Extraction
echo ${arrayZ[@]:0}
# one two three four five five # All elements.
echo ${arrayZ[@]:1}
# two three four five five # All elements following element[0].
echo ${arrayZ[@]:1:2}
# two three # Only the two elements after element[0].
```

Arrays (cntd…)

String operations on arrays

#String Removal

echo ${arrayZ[@]#f*r}

Removes shortest match from front of string(s).

echo ${arrayZ[@]##f*r}

Removes longest match from front of string(s).

echo ${arrayZ[@]%t*e}

% Removes shortest match from back of string(s).

echo ${arrayZ[@]%%t*e}

%% Removes shortest match from back of string(s).

Arrays (cntd…)

String operations on arrays

Substring Replacement

echo ${arrayZ[@]/five/WXYZ}

Replace first occurrence of substring with replacement.

echo ${arrayZ[@]//five/YYYY}

Replace all occurrences of substring.

Then >>>> **echo ${arrayZ[@]//five/} will do??**

Arrays (cntd…)

Example to load the contents of a file into array

$cat sample_file
1 a b c
2 d e f g

Arrays (cntd…)

Example to load the contents of a file into array

```
#!/bin/bash
filename=sample_file
declare -a array1
array1=( `cat "$filename"`)
echo ${array1[@]}
element_count=${#array1[*]}
echo $element_count
```

```
1 a b c 2 d e fg
8
```