Shell Programming-II

Special Characters

#	starts a comment.
$	indicates the name of a variable.
\	escape character to display next character literally.
{ }	used to enclose name of variable.
;	Command separator [semicolon]. Permits putting two or more commands on the same line.
;;	Terminator in a case option [double semicolon].
.	"dot" command [period]. Equivalent to source. This is a bash builtin.
$?	exit status variable.
$$	process ID variable.
[ ]	test expression
[[ ]]	test expression, more flexible than [ ]
$[ ], (( ))	integer expansion
||, &&, !	Logical OR, AND and NOT

Quotations

- Double Quotation " "
  - Enclosed string is expanded (”$”, ”/” and ”‘”)
  - Example: `echo "$myvar"` prints the value of `myvar`
- Single Quotation ’ ’
  - Enclosed string is read literally
  - Example: `echo ’$myvar’` prints `$myvar`
- Back Quotation ‘ ‘
  - Used for command substitution
  - Enclosed string is executed as a command
  - Example: `echo ‘pwd‘` prints the output of the `pwd` command i.e. print working directory
  - In **bash**, you can also use `$(···)` instead of ‘···‘
    e.g. `$(pwd)` and ‘`pwd`‘ are the same

Arithmetic Operations

- You can carry out numeric operations on integer variables

Operation	Operator	
Addition	+	
Subtraction	-	
Multiplication	*	
Division	/	
Exponentiation	**	(**bash** only)
Modulo	%	

- Arithmetic operations in **bash** can be done within the `$((···))` or `$[···]` commands
  - ★ Add two numbers: `$((1+2))`
  - ★ Multiply two numbers: `$[$a*$b]`
  - ★ You can also use the `let` command: `let c=$a-$b`
  - ★ or use the `expr` command: `c=‘expr $a - $b‘`

Comparison Operators

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Logical Operator

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if str is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Selective Flow Control

- An **if/then** construct tests whether the exit status of a list of commands is 0, and if so, executes one or more commands.

```
if [ condition1 ]; then
  some commands
elif [ condition2 ]; then
  some commands
else
  some commands
fi
```

- Note the space between *condition* and "[" "]"
- **bash** is very strict about spaces.

Selective Flow Control (contd...)

```
read a
if [ "$a" -gt 0 ]; then
  if [ "$a" -lt 5 ]; then
    echo "The value of \"a\" lies somewhere between 0
          and 5"
  fi
fi
```

```
read a
if [[ "$a" -gt 0 &&  "$a" -lt 5 ]]; then
  echo "The value of $a lies somewhere between 0 and
        5"
fi
OR
if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]; then
  echo "The value of $a lies somewhere between 0 and
        5"
fi
```

Loop Construct

- A *loop* is a block of code that iterates a list of commands as long as the *loop control condition* is true.
- Loop constructs available in

`for`, `while` and `until`

Loop Construct / for construct

- The `for` loop is the basic looping construct in **bash**

```
for arg in list
do
  some commands
done
```

- the `for` and `do` lines can be written on the same line: `for` *arg* in *list*; `do`
- `for` loops can also use C style syntax

```
for (( EXP1; EXP2; EXP3 )); do
  some commands
done
```

Loop Construct / while construct

- The **while** construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).
- In contrast to a **for** loop, a **while** loop finds use in situations where the number of loop repetitions is not known beforehand.

```
while [ condition ]
do
  some commands
done
```

factorial.sh

```
#!/bin/bash

echo -n "Enter a number less than 10: "
read counter
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

Loop Construct / until construct

- The `until` construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of `while` loop).

```
until [ condition is true ]
do
  some commands
done
```

```
i=1
until (($i > 3))
do
        echo 'UPES'
        i=$[ $i + 1 ]
done
```

Switching or Branching Construct

- The `case` and `select` constructs are technically not loops, since they do not iterate the execution of a code block.
- Like loops, however, they direct program flow according to conditions at the top or bottom of the block.

case construct

```
case variable in
  "condition1")
    some command
    ;;
  "condition2")
    some other command
    ;;
esac
```

Arrays in Shell Programming

- Array elements may be initialized with the `variable[xx]` notation

  `variable[xx]=1`

- Initialize an array during declaration

  `name=(firstname 'last name')`

  - reference an element `i` of an array `name`

    `${name[i]}`

  - print the whole array

    `${name[@]}`

  - print length of array

    `${#name[@]}`

Arrays in Shell Programming

- print length of element `i` of array `name`

  `${#name[i]}`

In **bash** `${#name}` prints the length of the first element of the array

- Add an element to an existing array

Functions in Shell Programming

- Like "real" programming languages, **bash** has functions.
- A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task.
- Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

```
function function_name {
  command
}
OR
function_name () {
  command
}
```