Writing a Simple Shell Script

A *shell script* is a file that holds commands that the shell can execute. The commands in a shell script can be any commands you can enter in response to a shell prompt. In addition to the commands you would ordinarily use on the command line, *control flow* commands (also called *control structures*) find most of their use in shell scripts.

There are a lot of different shells available for Linux but usually the bash (bourne again shell) is used for shell programming as it is available for free and is easy to use. For writing our shell programs we use any kind of text editor, e.g. vi .as with other programming languages. The program must start with the following line (it must be the first line in the file):

```
    #!/bin/sh
```

The #! characters tell the system that the first argument that follows on the line is the program to be used to execute this file. In this case /bin/sh is shell we use.

When you have written your script and saved it you have to make it executable to be able to use it.

To make a script executable type

chmod +x filename

Then you can start your script by typing: `./filename`

Comments

Comments in shell programming start with # and go until the end of the line.

Variables

In shell programming all variables have the datatype string and you do not need to declare them.

To assign a value to a variable you write:

```
varname=value
```

To get the value back you just put a dollar sign in front of the variable:

```
#!/bin/sh
# assign a value:
a="hello world"

# now print the content of "a":
echo "A is:"
echo $a
```

Type this lines into your text editor and save it e.g. as first. Then make the script executable by typing chmod +x first in the shell and then start it by typing ./first

The script will just print:

```
A is:
hello world
```

Sometimes it is possible to confuse variable names with the rest of the text:

```
num=2
echo "this is the $numnd"
```

This will not print "this is the 2nd" but "this is the " because the shell searches for a variable called numnd which has no value. To tell the shell that we mean the variable num we have to use curly braces:

```
num=2
echo "this is the ${num}nd"
```

This prints what you want: this is the 2nd

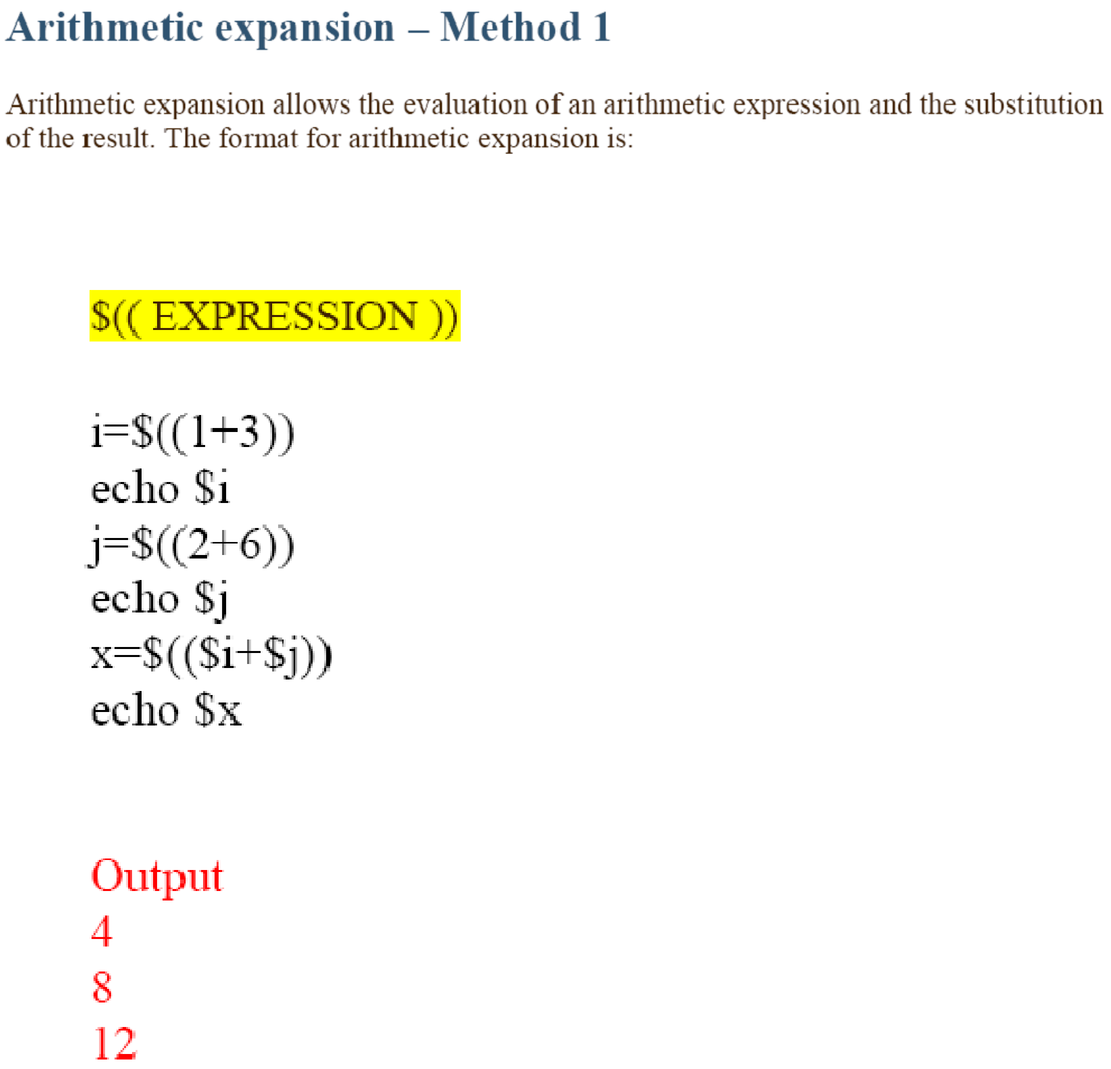
Reading User Input

Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

read variable1, variable2,...variableN

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.



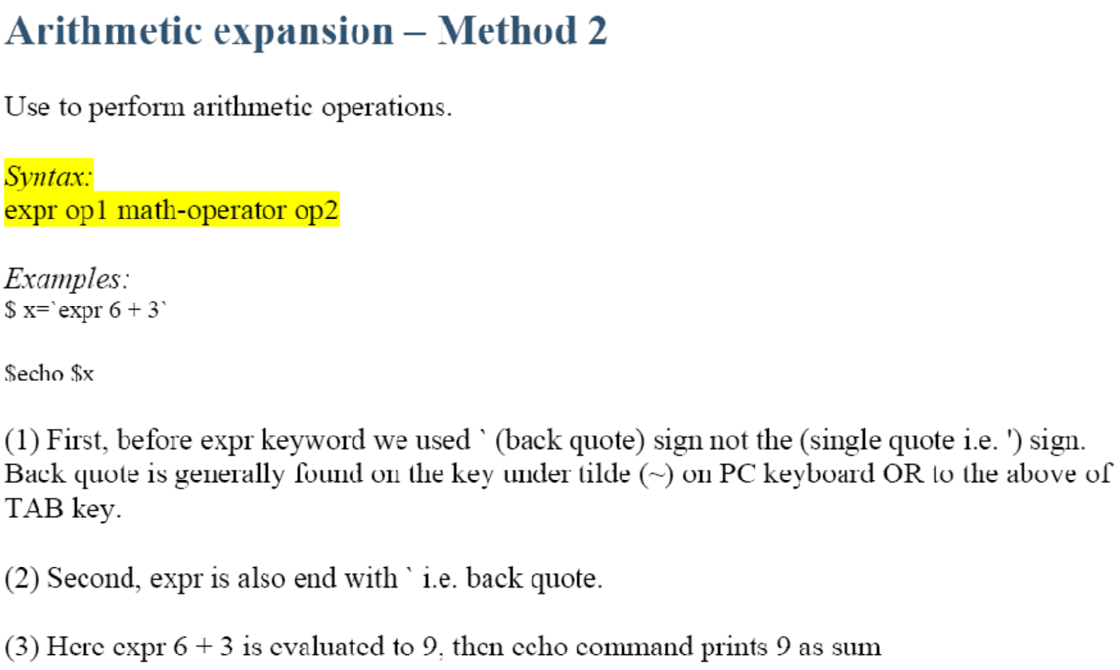
Arithmetic expansion – Method 1

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

```
$(( EXPRESSION ))

i=$((1+3))
echo $i
j=$((2+6))
echo $j
x=$(($i+$j))
echo $x
```

Output

```
4
8
12
```



Arithmetic expansion – Method 2

Use to perform arithmetic operations.

Syntax:
expr op1 math-operator op2

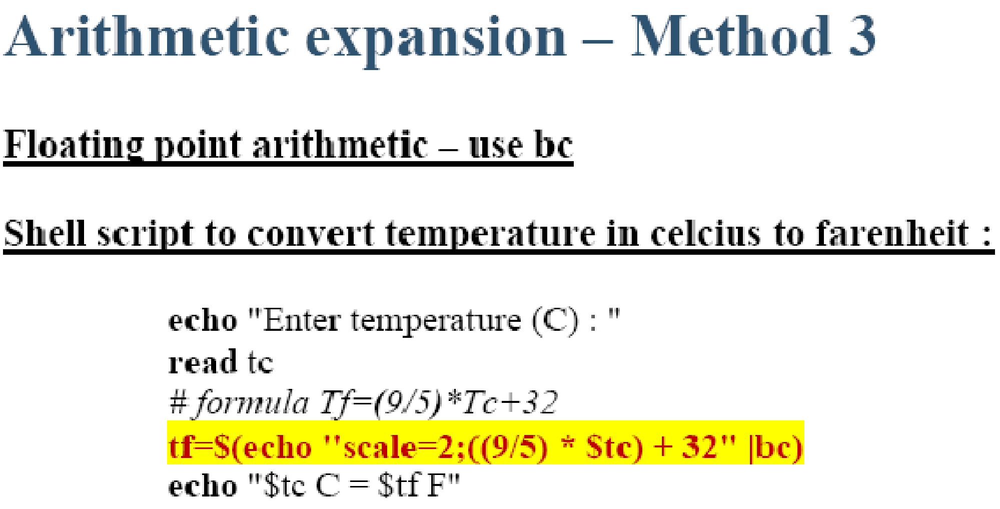
Examples:

```
$ x=`expr 6 + 3`

$echo $x
```

(1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboard OR to the above of TAB key.

(2) Second, expr is also end with ` i.e. back quote.

(3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum



Arithmetic expansion – Method 3

Floating point arithmetic – use bc

Shell script to convert temperature in celcius to farenheit :

```
echo "Enter temperature (C) : "
read tc
# formula Tf=(9/5)*Tc+32
tf=$(echo "scale=2;((9/5) * $tc) + 32" |bc)
echo "$tc C = $tf F"
```

BASIC OPERATORS

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

Here is simple example to add two numbers:

```
val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result:

```
Total value : 4
```

Arithmetic Operators:

There are following arithmatic operators supported by Bourne Shell.

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

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a * $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

Relational Operators:

Bourne Shell supports following relational operators which are specific to numberic values. These operators would not work for string values unless their value is numerics.

For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

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

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

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

Example

```
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

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

This would produce following result:

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

Boolean Operators:

There are following boolean operators supported by Boume Shell.

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

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 condition would be true.	[$a -lt 20 -o $b -gt 100] is true
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

Example

```
a=10
b=20

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

```
if [ $a -lt 100 -a $b -gt 15 ]
then
  echo "$a -lt 100 -a $b -gt 15 : returns true"
else
  echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
  echo "$a -lt 100 -o $b -gt 100 : returns true"
else
  echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
  echo "$a -lt 100 -o $b -gt 100 : returns true"
else
  echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

This would produce following result:

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

String Operators:

There are following string operators supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then:

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then 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 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 non-zero length then it returns true.	[-z $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

Example

```
a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

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

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

This would produce following result:

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

File Test Operators:

There are following operators to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then	[-f $file] is true.

	condition becomes true.	
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

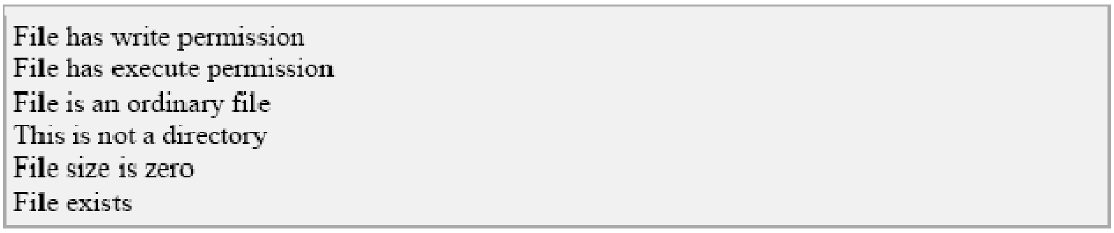
Example

```
file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
   echo "File has read access"
else
```

```
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

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

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi

if [ -s $file ]
then
   echo "File size is zero"
else
   echo "File size is not zero"
fi

if [ -e $file ]
then
   echo "File exists"
else
   echo "File does not exist"
fi
```

This would produce following result:

```
File has read access
```



```
File has write permission
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```