Shell Programming - Operator

Amit Patel

OPERATOR

- There are various operators supported by each shell.
- There are following operators supported in BOURNE shell which we are going to discuss

1. Arithmetic Operators.
2. Relational Operators.
3. Boolean Operators.
4. String Operators.
5. File Test Operators.

OPERATOR

- There are various operators supported by each shell.
- There are following operators supported in BOURNE shell which we are going to discuss

1. Arithmetic Operators.
2. Relational Operators.
3. Boolean Operators.
4. String Operators.
5. File Test Operators.

expr Command

- expr is a command line Unix utility which evaluates an expression and outputs the corresponding value.
- Syntax: expr (expression)
- expr evaluates integer or string expressions, including pattern matching regular expressions.
- The operators available
    - for integers: addition, subtraction, multiplication, division and modulus
    - for strings: find regular expression, find a set of characters in a string; in some versions: find substring, length of string
    - for either: comparison (equal, not equal, less than, etc.)

expr Command

- For pure arithmetic, it is often more convenient to use bc. For example:

```
echo "3*4+14/2" | bc
```

- since it accepts the expression as a single argument.
- For portable shell programming use of the length and substr commands is not recommended..

Arithmetic Operator

- There are following arithmetic 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

Arithmetic Operator

Operator	Description	Example
=	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 Operator

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

Relational Operator

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.

Boolean Operator

- There are following boolean operators supported by Bourne Shell.

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.

String Operator

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

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

Decision Making

- While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths.
- So you need to make use of conditional statements that allow your program to make correct decisions and perform right actions.
- Unix Shell supports conditional statements which are used to perform different actions based on different conditions.
- Here we will explain following two decision making statements −
  - The **if...else** statements
  - The **case...esac** statement

Decision Making

- If else statements are useful decision making statements which can be used to select an option from a given set of options.
- Unix Shell supports following forms of if..else statement −
  - if...fi statement
  - if...else...fi statement
  - if...elif...else...fi statement
- Most of the if statements check relations using relational operators discussed in previous chapter.

If…..fi statement

- The if...fi statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.
- Syntax

```
 if [ expression ] then
    Statement(s) to be executed if expression is true
 fi
```

- Here Shell expression is evaluated.
- If the resulting value is true, given statement(s) are executed. If expression is false then no statement would be not executed.
- Most of the times you will use comparison operators while making decisions.

If…..fi statement

- There is a spaces between braces and expression. This space is mandatory otherwise you would get syntax error.
- If expression is a shell command then it would be assumed true if it return 0 after its execution.
- If it is a boolean expression then it would be true if it returns true.

- a is not equal to b

If…..fi statement

```
#!/bin/sh

a=10

b=20

if [ $a == $b ] then

   echo "a is equal to b"

fi

if [ $a != $b ] then

      echo "a is not equal to b"

fi
```

- This will produce following result − a is equal to b.

If…else…fi statement

- The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.
- Syntax

```
 if [ expression ] then
    Statement(s) to be executed if expression is true
 Else
  Statement(s) to be executed if expression is false
 fi
```

- Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

If…else…fi statement

```
#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

   echo "a is equal to b"

else

   echo "a is not equal to b"

Fi
```

This will produce following result − a is not equal to b.

If…elif…fi statement

- The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.
- Syntax:

```
if [ expression 1 ] then
      Statement(s) to be executed if expression 1 is true
   elif [ expression 2 ] then
      Statement(s) to be executed if expression 2 is true
   elif [ expression 3 ] then
     Statement(s) to be executed if expression 3 is true
  else
     Statement(s) to be executed if no expression is true
 fi
```

If…else…fi statement

```
#!/bin/sh
a=10
b=20
if [ $a == $b ] then
   echo "a is equal to b"
elif [ $a -gt $b ] then
   echo "a is greater than b"
elif [ $a -lt $b ] then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

This will produce following result – a is less than b

case…esac statement

- You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.
- Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.
- Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++
- The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression.
- The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

If…elif…fi statement

```
case word in
 pattern1)
   Statement(s) to be executed if pattern1 matches
   ;;
 pattern2)
   Statement(s) to be executed if pattern2 matches
   ;;
 pattern3)
   Statement(s) to be executed if pattern3 matches
   ;;
esac
```

case…esac statement

- Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.
- There is no maximum number of patterns, but the minimum is one.
- When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

case..esac statement

```
#!/bin/sh
FRUIT="kiwi"
case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

This will produce following result − New Zealand is famous for kiwi.

case..esac statement

```
#!/bin/sh
FRUIT="kiwi"
case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

This will produce following result − New Zealand is famous for kiwi.

To read a file line by line in Shell:

Option 1:

```
$ cat file | while read line
 do
        echo $line
 done
```

Option 2:

```
$ while read line
do
        echo $line
done < file
```

To read a file line by line in Shell:

- In the first option, we use the cat command and pipe the output to the while command. However, in the Option 2, it is purely internal where the file is read using the input file descriptor.
- Tip : Always prefer internal commands if possible.