Scripting & Computer Environments

Shell Scripting II

IIIT-H

Aug 28, 2013

...Previously & Today...

- Previously: Shell Scripting I
  - Creating shell scripts
  - Shell Variables
  - Quotes
  - Reading Input in Shell
  - Shell metacharacters:

; . ’ " \ , ` : * ? - ∼

- Today:
  - Expressions
  - Flow Control
    1. Selection
    2. Looping

...Previously & Today...

- Previously: Shell Scripting I
  - Creating shell scripts
  - Shell Variables
  - Quotes
  - Reading Input in Shell
  - Shell metacharacters:

`#   ;   .   '   "   \   ,   ` + "`" + `   :   *   ?   -   ~`

- Today:
  - Expressions
  - Flow Control
    1. Selection
    2. Looping

Brainstorm

- Shell Script?

Brainstorm

- Shell Script?

Shell Script

Simply, a text file that contains executable commands.

- When to use them?

- When not to use them?

Recap: Shell Variables

- No need to declare, no type.
- To read their values, precede them by a dollar sign ($).
- Local vs Environment Variables.
- Environment variables are passed to child processes but locals are not (i.e. their scopes differ).

Example

```
export MyVar='Hello'
echo $MyVar
bash
echo $MyVar
```

Example

```
x=5
echo $x
bash
echo $x
```

Quotes

- Single Quote (')

  Preserves the literal meaning of each character within it, except itself.

- Double Quote (")

  Preserves the literal meaning of all characters within it (except $, \ and itself).

- Back Quote / Back Tick (`)

  Executes the command it encloses (same as `$(command)`)

Recap: Shell Arguments

- Passing arguments to our scripts is via positional parameters (a.k.a. command-line arguments)
- *A*re predefined buffers in the shell script.
- `$1` through `$9` (read about the `shift` command)
- During execution, the shell puts the first argument as `$1`, the second as `$2` and so on.

Other Special parametres/variables:

- Name of the script (`$0`)
- Number of arguments (`$#`)
- All parameters (`$*` and `$@`)
- Exit staus (`$?`)

Recap: Shell Arguments

- Passing arguments to our scripts is via positional parameters (a.k.a. command-line arguments)
- *Are* predefined buffers in the shell script.
- `$1` through `$9` (read about the `shift` command)
- During execution, the shell puts the first argument as `$1`, the second as `$2` and so on.

Other Special parametres/variables:

- Name of the script (`$0`)
- Number of arguments (`$#`)
- All parameters (`$*` and `$@`)
- Exit staus (`$?`)

Exit Status

- Commands return a value to the system when they terminate.
- The value (b/n 0 and 255) denotes success/failure of command's execution.

```
ls -l /bin                  (0 is success)
echo $?
ls -l /IDoNotExist
echo $?                     (any other value is failure)
```

The `test` operator

```
test expression
[ expression ]             [] is shorthand
```

- Performs a variety of checks.
- Returns exit status of 0 if `expression` is true; 1 otherwise.

Exit Status

- Commands return a value to the system when they terminate.
- The value (b/n 0 and 255) denotes success/failure of command's execution.

```
ls -l /bin                  (0 is success)
echo $?
ls -l /IDoNotExist
echo $?                     (any other value is failure)
```

The `test` operator

```
test expression
[ expression ]          [] is shorthand
```

- Performs a variety of checks.
- Returns exit status of 0 if `expression` is true; 1 otherwise.

Expression

Sequence of `operators` & `operands` that reduces to a single value.

```
x=2                 $x + $y

y=4                 ($x * $y) / $x - $y
```

Use the `expr` command to evaluate expressions.

- Some operators:
  - Arithmetic operators
  - Comparison operators
  - Logical operators
  - File operators
  - Test operator

Arithmetic Operators

- Addition, Subtraction (+, -)
- Multiplication, Division (*, /)
- Exponentiation (**)
- Modulus (%)
- Increment, Decrement (++, --)

- Short-hand assignments possible.

```
+=    -=    *=    /=    %=
```

- Doing integer arithmetic using the $(( )) construct and the `let` shell built-in.
- How about floating-point arithmetic?

Arithmetic Operators

Addition, Subtraction	`(+, -)`
Multiplication, Division	`(*, /)`
Exponentiation	`(**)`
Modulus	`(%)`
Increment, Decrement	`(++, --)`

- Short-hand assignments possible.

  `+=` `-=` `*=` `/=` `%=`

- Doing integer arithmetic using the $(( )) construct and the `let` shell built-in.
- How about floating-point arithmetic?

Comparison Operators (Integer)

`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-lt`	Less than
`-le`	Less than or equal to

String Comparison

`s1 == s2`	Equal to
`s1 != s2`	Not equal to
`-z str`	True if `str` is zero/null
`-n str`	True if `str` not null

Comparison Operators (Integer)

`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-lt`	Less than
`-le`	Less than or equal to

String Comparison

`s1 == s2`	Equal to
`s1 != s2`	Not equal to
`-z str`	True if `str` is zero/null
`-n str`	True if `str` not null

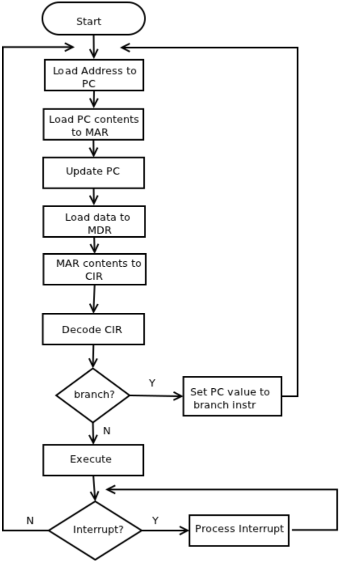
Logical Operators

- `expr1 AND expr2` → `expr1 && expr2`
- `expr1 OR expr2` → `expr1 || expr2`
- `NOT expr` → `!expr`

File Operators

`-e file`	`file` exists?
`-r file`	`file` exists and readable?
`-w file`	`file` exists and writable?
`-x file`	`file` exists and executable?
`-L file`	`file` exists and a symbolic link?
`-f file`	`file` exists and a regular file?
`-d file`	`file` exists and a directory?
`file1 -nt file2`	`file1` newer than `file2`?
`file1 -ot file2`	`file1` older than `file2`?

Flow Control: Instruction Execution Model



PC= Program Counter

MAR= Memory Address Register

MDR= Memory Data Register

CIR= Current Instruction Register

Flow Control: The if Construct

if-then-else (2-way)

```
if <command>
then
<Do this thing>
else
<Do that thing>
fi
```

Example

```
if who | grep $1  > /dev/null
then
echo "$1 is logged in."
else
echo "$1 is not logged in"
fi
```

Flow Control: The if Construct

if-then-else (2-way)

```
if <command>
then
<Do this thing>
else
<Do that thing>
fi
```

Example

```
if who | grep $1  > /dev/null
then
echo "$1 is logged in."
else
echo "$1 is not logged in"
fi
```

if-then-elif-else (Nested if)

```
if <command1>
then
<commands 1>
elif <command2>
then
<commands 2>
...
else
<commands N>
fi
```

The `case` Command (multi-way selection)

```
case <expression> in
     pattern1) command 1 ;;
     pattern2) command 2 ;;
     pattern3) command 3 ;;
     ...
esac
```

- `case` matches `expression` with `pattern1` first.
- If matched, it executes `command 1`. Otherwise, proceeds to `pattern2` and so on.
- Pattern may be a regex (wilcards + EREs).

The `while` Looping construct

```
while <condition>
do
   <commands>
done
```

- Executes `<commands>` if exit status of `<condition>` is 0 i.e. successful.

Example

```
i=1
while [ $i -le 20 ]
do
echo "$i"
i=$(($i+1))       (or  i=`expr $i + 1`)
done
```

The until Looping construct

```
until <condition>
do
   <commands>
done
```

- Executes `<commands>` as long as `<condition>` is non-zero i.e. fails (until condition becomes true).

Example

```
i=1
until [ $i -ge 11 ]
do
echo $i
i=$(($i+1))
done
```

The `for` Looping construct

```
for <variable> in <list>
do
   <commands>
done
```

- Every successive item in `<list>` is assigned to `<variable>` and `<commands>` executed.
- Use the `seq` command to specify range. Its `man` page for more.

Example

```
for i in 1 2 3 4 5
do
echo $i
done
```

For Loop (2)

- Specifying ranges in `for` loop.
  1. `{START..END..INCREMENT}`
  2. `seq START INCREMENT END`

- C-like flavor of `for` loop

```
for (( i=1; i<=5; i++))
do
echo $i
done
```

- To exit `for`, `while` and `until` loops prematurely.

```
while <condition>
do
    <action 1>
    <action 2>
    if <some check>
    then
        break                                  (breaks out)
    fi
<action 3>
<action 4>

done
```

continue

- Skips to the next loop iteration.

```
for i in <some list>
do
    <command 1>
    <command 2>
    if <some check>
    then
        continue                (skips to next iteration)
    fi
<command 3>

done
```

- Input Redirection with Looping
- Output Redirection with Looping
- Pipe to Loops
- Pipe from Loops