

# Bourne Shell Scripting

## Introduction

- Bourne (`sh`) / BASH (`bash`) are more than command interpreters
  - they are also a programming language
    - variables, control structures (conditional expressions, loops)
- shell script
  - simple text file containing a sequence of commands and statements interpreted by the `sh`
  - anything you type on a command line can be part of the script

- Con's:
  - syntax (ie. spacing) is very picky
  - error messages are not very helpful
- Pro's:
  - quick and simple to write

## Shell Script Example

```
#!/bin/bash
# read in users name
echo "Enter your name:"
read name

# read in a number
echo "Hello $name. Guess a number between 0-9:"
read guess
if [ $guess -eq 8 ]
then
    echo "$name, your right!"
else
    echo "$name, your wrong!"
fi
```

## Executing a Shell Script

- Make the script runnable by setting execution bit:

```
% chmod u+x myscript.sh
```
- Now you can run the script by any of the following

```
% ./myscript.sh
```

```
% /bin/sh myscript.sh
```

## Shell Variables

- Variable
  - memory location that can be referenced by a name rather than an address
- shell variable
  - sequence of characters, numbers, and underscores (“\_”)
  - must begin with a char or “\_”
  - stored as a string of chars – even when assigned a number

## Two types of shell variables:

### 1. Environment Variables

- customize your shell environment
- most are read-write (ex. PATH)
- others are read-only (i.e. command line arguments, PID)

## Common Read-Only Shell Variables

<b>Env Variable</b>	<b>Variable Description</b>
\$0	Name of Program
\$1 - \$9	Values of command line args 1-9
\$*	Values of all command line args
\$@	Values of all command line args; each quoted i.e. "-Plexmark_lp"
\$#	Number of command line args
\$\$	PID of current process
\$?	Exit status of most recent command
#!	PID of most recent background cmd

## 2. User-Defined Variables

- temporary locations to store values during computation of the script
- don't need to be declared in advance
- initialized to the null string
- `env` command displays *many* shell variables
- `set` commands displays *all* shell vars

## Reading/Writing Shell Variables

- Assigning a value to a shell variable follows the simple syntax of

```
% var=value
```

  - notice the lack of white space
  - to include spaces enclose the string in quotes

```
% var="Hello world"
```
- Inside of quotes everything except metacharacters are treated as a string

- **More examples:**

```
% myvar=hello world
bash: world: command not found
% echo $myvar
hello
% echo \ $myvar
 $myvar
% star=*
% echo $star
[names of all files in pwd]
% echo "$*"
*
```

## Exporting Variables

- **User-defined variables are not inherited by commands or subshells. Consider the following,**

```
% myvar=hello; echo $myvar
hello
% cat > myscript.sh
echo $myvar
^D
% mychod u+x myscript.sh; ./myscript.sh

% export myvar; ./myscript.sh
hello
```

- myvar was not known by myscript until it was exported (not in scope)
- changes to myvar inside of myscript's scope doesn't change the value in the shell's scope

```
% myvar=hello
% cat > myscript.sh
echo $myvar
myvar=world
^D
% ./myscript
hello
```

## Command Substitution

- Enclosing a command in backquotes causes the command to be substituted for its results. For example:

```
% echo "The current dir is `pwd`"
The current dir is /home/jay
```

## Resetting Variables

- A variable retains a value until script finished or the variable is reset to null

```
% echo $myvar  
hello  
% unset myvar; echo $myvar
```

- Explicit assignment to null

```
% myvar=
```

## Read-Only Variables

- If a variable remains constant then it can be made a symbolic constant using the `readonly` command

- Syntax:

```
readonly [variable-list]
```

- For example,

```
% pi=3.14  
% readonly pi  
% pi=6  
bash: pi: readonly variable
```



## Argument Handling

- Command line arguments (args) passed to a script can be referenced via \$1-\$9
- \$# identifies the args count
- \$\* and @\$ store all args
- To access more than nine args we use the shift command to move the args to the left (\$5 becomes \$4 etc.).
- Syntax: Shift args left by n  
`shift [n]`

- Set command can be used to fix args position

```
% set `date`  
Sat Oct 13 14:04:53 EDT 2007  
% echo "$@"  
Sat Oct 13 14:04:53 EDT 2007  
% echo "$2 $3, $6"  
Oct 13, 2007
```

## Formatting `echo` Output

Character	Meaning
<code>\b</code>	Backspace
<code>\c</code>	Print line without moving cursor to next line
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\\</code>	Backslash
<code>\ON</code>	Character with ASCII octal number N

## Getting User Input

- The `read` command can be used to receive input from the user. `read` consumes an entire line at once.
- Syntax:  

```
read variable-list
```
- Number of elements in `variable-list` dictates how the input will be spliced.

## read Example

```
echo "Enter some text:"
read text1
echo text1 ← entire line of text
echo "Enter more text:"
read text2 text3
echo $text2 ← first word of text
echo $text3 ← remainder of text
```

## Control Flow Statements

- Conditional expressions which alter sequential program execution
  - two-way branch
    - `if` statement
  - multi-way branch
    - `if` and `case` statements
  - repetition/loop
    - `for`, `while` and `until` statements

## if Statements

- if-then-elif-else-fi statement
  - Test if a condition is true or false

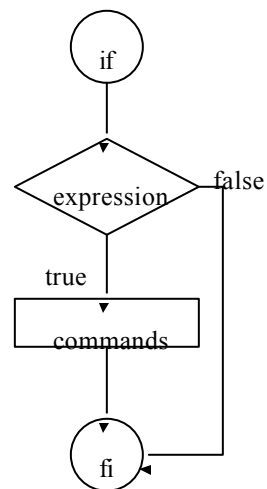
- Syntax:

```
if expression
  then
    [elif expression
      then
        true-commands
    ]
    [else
      false-commands
    ]
fi
```

## Simple if Statement

- Two-way branching:

```
if expression
  then
    commands
fi
```



## test Command

- Used in conjunction with an if statement
- Evaluates to true or false
- For example,

```
if test -f "$myfile"
then
    echo "File exists"
else
    echo "File does not exists"
fi
```

## File Testing

<b>Expression</b>	<b>Return Value</b>
-d file	True if file is a directory
-f file	True is file is an ordinary file
-r file	True is file is readable
-w file	True is file is writable
-x file	True is file is executable
-s file	True is file length is nonzero

## Integer Testing

<b>Expression</b>	<b>Return Value</b>
<code>i1 -eq i2</code>	True if <code>i1</code> equals <code>i2</code>
<code>i1 -ge i2</code>	True if <code>i1</code> is greater than or equal to <code>i2</code>
<code>i1 -gt i2</code>	True if <code>i1</code> is greater than <code>i2</code>
<code>i1 -le i2</code>	True if <code>i1</code> is less than or equal to <code>i2</code>
<code>i1 -lt i2</code>	True if <code>i1</code> is less than <code>i2</code>
<code>i1 -ne i2</code>	True if <code>i1</code> does not equal <code>i2</code>

## String Testing

<b>Expression</b>	<b>Return Value</b>
<code>s</code>	True if <code>s</code> is non-null
<code>s1 = s2</code>	True if <code>s1</code> equals <code>s2</code>
<code>s1 != s2</code>	True if <code>s1</code> does not equal <code>s2</code>
<code>-n s</code>	True if length of <code>s</code> is greater than 0
<code>-z s</code>	True if length of <code>s</code> is 0

## Logical Operators

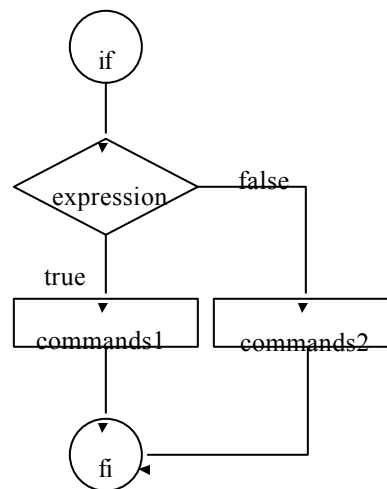
- All of the test expressions can be used with the following operators

!	Logical NOT
-o	Logical OR
-a	Logical AND
(expression)	Grouping multiple expressions together

## if-else Statement

- Two-way branching:

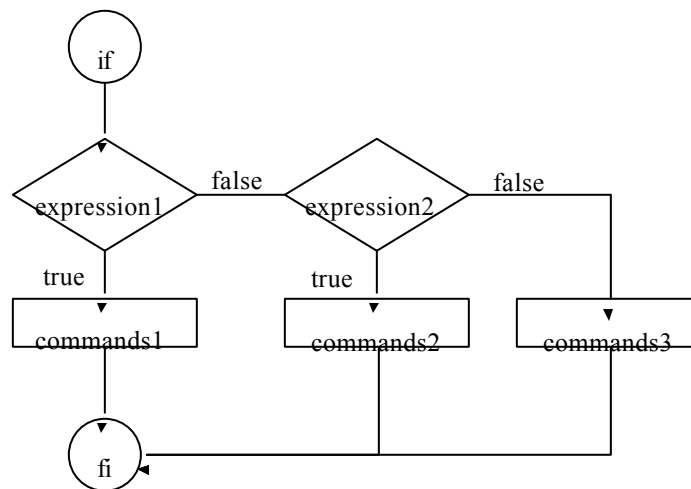
```
if expression
  then
    commands1
  else
    commands2
fi
```



## if-elif-else Statement

- Multi-way branching:

```
if expression1
  then
    commands1
  elif expression2
    commands2
  else
    commands3
fi
```





## for Statement

- Counted loop that repeats the same sequence of commands (loop body) usually a known number of iterations.
- Syntax:

```
for variable [in list]
do
    commands
done
```

## for Loop Example

- Print all the files in the current dir that end with .ps

```
for file in *.ps
do
    lpr -Plexmark_lp $file
done
```

## while Statement

- Repeat the loop body while a condition remains true.
- Syntax:

```
while condition
do
  commands
done
```
- `condition` is evaluated at the start of every iteration prior to entering the loop body (and therefore executing `commands`)

## while Loop Example

- Keep guessing until user gets correct number:

```
guess=0
while [ $guess != 8 ]
do
  echo "Guess a number between 0-9:"
  read guess
done
echo "$name, your right!"
```

## until Statement

- Opposite of while loop
- Repeat until some condition is true
- Syntax:

```
until condition
do
  commands
done
```

## until Loop Example

- Keep guessing until user gets correct number:

```
guess=0
until [ $guess = 8 ]
do
  echo "Guess a number between 0-9:"
  read guess
done
echo "$name, your right!"
```


## break and continue Commands

- Interrupt execution of a loop iteration
- break
  - jump to command following done
  - “break” out of current iteration
- continue
  - jump to command following done
  - “continue” at next iteration

## break Example

- Jump out of loop if some cond is true

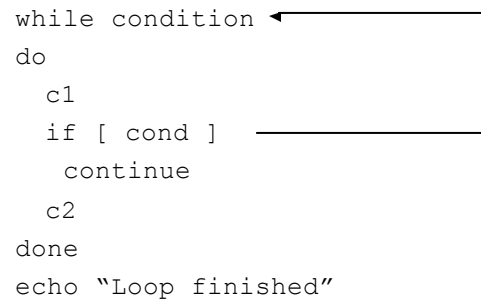
```
while condition
do
  c1
  if [ cond ]
    break
  c2
done
echo "Loop finished" ◀
```

A diagram illustrating the effect of the 'break' command. A horizontal line extends from the 'break' statement to the right, then turns 90 degrees down, then 90 degrees left, and finally 90 degrees down to point at the 'echo' statement, indicating that the loop is exited and execution continues with the command following the loop.

## continue Example

- Jump to the start of the next iteration if some cond is true

```
while condition ←  
do  
  c1  
  if [ cond ]  
    continue  
  c2  
done  
echo "Loop finished"
```



## case Statement

- Multi-way branch similar to if-elif
  - More concise and readable

- Syntax:

```
case str in  
  pattern1) commands1  
            ;;  
  pattern2) commands2  
            ;;  
esac
```

## case Statement Example

- User menu:

```
echo "Enter a command"
read input
case "$input" in
  a|A)      echo "A"
            ;;
  b|B)      echo "B"
            ;;
  x|X)      echo "Good-bye"
            ;;
esac
```

## Functions

- Program statements that accomplish a specific task can be grouped into a function
  - improves structure, robustness, readability, reuse
  - smaller, easier to program
- Syntax:

```
MyFunction ()
{
  commands
}
```

## Example of a Simple Function

```
% gohome ()
>{
>   echo `pwd`;
>   cd;
>   echo `pwd`
>}
% gohome
/tmp
/home/Jason
```

## Processing Numeric Data

- Shell scripting is better suited for string processing
  - variables stored as strings
  - arithmetic
    - convert variable into a number perform computation and then translate it back to string for storage
- Enabled by the *evaluate expression* (`expr`) command

## expr Command

Operator	Explanation
\	Return 1st expr if it is not null (or 0), else return 2nd expr
\&	Return 1st expr if neither is null (or 0), else return 0
=	Equal to
\>	Greater than
\>=	Greater than or equal to
\<	Less than
\<=	Less than or equal to
!=	Not equal to
+, -, \*, /, %	addition, subtraction, multiplication, division, modulo

## Script Debugging

- Running a script with the `-x` (echo) and `-v` (verbose) options aids debugging
- This will report (echo) all of the scripts commands before they are executed