Shells & Shell Programming (Part B)

Software Tools

EECS2031 Winter 2018

Manos Papagelis

Thanks to Karen Reid and Alan J Rosenthal

for material in these slides

CONTROL STATEMENTS

Control Statements

- Conditional statements
  - if/then/else
  - test
  - case
- Loops - Repetitive task statements
  - for
  - while

if/then/else

```
if TEST-COMMANDS; then
   CONSEQUENT-COMMANDS;
else
   ALTERNATE-CONSEQUENT-COMMANDS
fi
```

Test

- The built-in command `test` is used to construct conditional statements in Bourne shell (sh)
- Equivalent to […] in bash

`test` arguments

`-d filename`	Exists as a directory
`-f filename`	Exists as a regular file
`-r filename`	Exists as a readable file
`-w filename`	Exists as a writable file
`-x filename`	Exists as an executable file
`-z string`	True if empty string
`str1 = str2`	True if str1 equals str2
`str1 != str2`	True if str1 not equal to str2
`int1 -eq int2`	True if int1 equals int2
`-ne, -gt, -lt, -le`	
`-a, -o`	And, or

test: example

```
#!/bin/bash

if test -z "$1"; then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi
```

[…]: example

```
#!/bin/bash
directory="./BashScripting"

# bash check if directory exists
if [ -d $directory ]; then
  echo "Directory exists"
else
  echo "Directory does not exists"
fi
```

if/test relationship

- `test` is a command that returns a value
- If statements check the return value of the command
- `test` equivalent (brackets): `[   ]`

```
if test ! -d notes
then
   echo not found
else
   echo found
fi
```

=

```
if [ ! -d notes ]
then
   echo not found
else
   echo found
fi
```

case

```
case EXPRESSION in
  PATTERN1) COMMAND-LIST;;
  PATTERN2) COMMAND-LIST;;
  ...
  PATTERNN) COMMAND-LIST;;
esac
```

case: example

```
#!/bin/bash
echo "What is your preferred programming / scripting language"
echo "1) bash"
echo "2) perl"
echo “3) I do not know !"
read choice;
case $choice in
    1) echo "You selected bash";;
    2) echo "You selected perl";;
    3) exit
esac
```

for

```
for NAME [ in LIST ]; do
  COMMANDS;
done
```

for: example

```
#!/bin/bash

for f in $( ls /var/ ); do
  echo $f
done
```

for: example 2

Assume that:

```
% ls *.xml
% file1.xml  file2.xml  file3.xml
```

Then:

```
% ls *.xml > list
% for i in `cat list`; do cp "$i" "$i".bak ; done
% ls *.xml*
```

What would the output be?

OUTPUT：

```
% file1.xml file1.xml.bak file2.xml file2.xml.bak file3.xml file3.xml.bak
```

while

```
while CONTROL-COMMAND; do
  CONSEQUENT-COMMANDS;
done
```

while: example

```
#!/bin/bash
COUNT=6

while [ $COUNT -gt 0 ]; do
  echo Value of count is: $COUNT
  let COUNT=COUNT-1
done
```

Note: **let** is used (as one way) to evoke arithmetic evaluation of an expression.

while: example 2

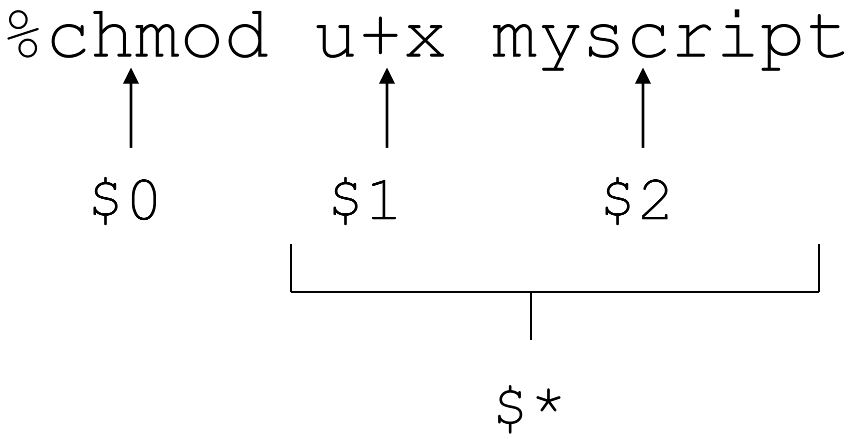
```
#!/bin/bash

# This script opens 4 terminal windows.
i="0"
while [ $i -lt 4 ]; do
  xterm &
  i=$[$i+1]
done
```

COMMAND LINE ARGUMENTS

Command line arguments

- positional parameters: **variables** that are assigned according to position in a string
- Command line arguments are placed in positional parameters:



```
%chmod u+x myscript
```

$0 $1 $2

$*

Positional Parameters

- Example:

giant

```
#!/bin/sh
echo arg1: $1
echo arg2: $2
echo name: $0
echo all: $*
```

```
$ giant fee fie fo fum
arg1: fee
arg2: fie
name: giant
all: fee fie fo fum
```

(Remember to run `chmod u+x giant` or `chmod 711 giant`)

Positional Parameters

Variable	What it references
$0	Name of the script
$#	Number of positional parameters
$*	Lists all positional parameters
$@	Same as $* except when in quotes
“$*”	Expands to a single argument (“$1 $2 $3”)
“$@”	Expands to separate arguments (“$1” “$2” “$3”)
$1 .. $9	First 9 positional parameters
${10}	10th positional parameter (need to use braces)

set and shift

- set – assigns positional parameters to its arguments.

```
$ set `date`
$ echo "The date today is $2 $3, $6"
The date today is May 25, 2006
```

- shift – change the meaning of the positional parameters

giant2

```
#!/bin/sh
while test "$1"
do
   echo $1
   shift
done
```

```
$ giant2 fee fie fo fum
fee
fie
fo
fum
```

Iterating over arguments

- Don’t use this one unless you know that the argument list will always be short
- sh allows only 9 positional parameters

```
#!/bin/sh
while test "$1"
do
   echo $1
   shift
done
```

- The method below is more portable.
- Use this one.

```
#!/bin/sh
for arg in "$@"
do
   echo $arg
done
```

Even more on quotes

- Getting the quotes right on a loop or similar commands can be a bit tricky.
- The following 4 loops do different things:

```
for arg in "$*"
do
   echo $
done
```

Quotes mean arguments are all in one string.

```
for arg in "$@"
do
   echo $arg
done
```

Blanks in the arg list are preserved

```
for arg in $*
do
   echo $
done
```

One element for each argument.

```
for arg in $@
do
   echo $ar
done
```

Does not preserve blanks in arg list.

EXPRESSIONS

expr

- Since shell scripts work by text replacement, we need a special function for arithmetic. Strings may be evaluated as numbers using **expr**

```
x=1
expr $x         #evaluates to 1
expr $x + 3     #evaluates to 4

x=`expr $x + 3`#evaluates to x=4
y=`expr 3 * 5` #doesn’t work
```

expr: more examples

```
% a=3
% b=$a" + 9"                          # string concatenation
% echo $b
3 + 9
% expr $b
12
% expr $a * $a                        # wrong! - need to escape *
expr: syntax error
% expr $a \* $a
9
```

String matching using expr

```
expr $string : $substring
```

- Returns the length of matching `substring` at the beginning of `string`. Example:

```
% string="hello"
% substring="hel"
% expr $string : $substring
3
```

- It returns 0 if the `substring` is not found at the beginning of `string`.
- Useful in some simple cases. If you need anything more complicated use Python, Perl, **sed** or **awk**.

READING USER INPUT

read

- reads one line from standard input and assigns successive words to the specified variables. Leftover words are assigned to the last variable.

name

```
#!/bin/sh
echo "Enter your name:"
read fName lName
echo  "First: $fName"
echo  "Last: $lName"
```

```
$ name
Enter your name:
Alexander Graham Bell
First: Alexander
Last: Graham Bell
```

Reading User Input: example

```
#!/bin/bash
echo "Hi, please type a word:"
read  word
echo "The word you entered is: $word"
echo "Can you please enter two words? "
read word1 word2
echo "Here is your input: \"$word1\" \"$word2\"“
echo "What are your three favorite colours ? "

# -a makes read command to read into an array
read -a colours
echo "My favorite colours are ${colours[0]},
  ${colours[1]} and ${colours[2]}:-)"
```

Reading from a file

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

- Reads one line at a time from a file.
- `$file` contains the name of the file that will be read from.

FUNCTIONS

functions

```
function FUNCTION {
  COMMANDS;
}
```

or

```
FUNCTION () {
  COMMANDS;
}
```

functions (more)

- You can create your own functions or subroutines:

```
% myfunc() {
   arg1=$1
   arg2=$2
   echo $arg1 $arg2 $globalvar
   return 0
}
% globalvar="I am global"
% myfunc num1 num2
% num1 num2 I am global
```

- Notes:
  - Arguments are passed through positional parameters.
  - Variables defined outside the function are visible within.
  - Return value is the value of the last executed command in the function.

functions: example

```
#!/bin/bash
# BASH FUNCTIONS DECLARATION
function function_B {
  COMMANDS;
}
function function_A {
  echo $1
}

# FUNCTION CALLS
% function_A "Function A."
% function_A.
```

`find [path…] [expression]`

- Expression
  - Options:
    - `-maxdepth level`
  - Tests:
    - `-name pattern`
      - Base of file name matches shell pattern pattern
    - `-newer file`
      - File was modified more recently the file.
  - Actions
    - `-print`
    - `-exec`

find: example

Displays the names of all the Java files in directories in and below the current working directory.

```
find . -name "*.java" -print
```

USING PIPES

The power of pipelines

- Question 1:

  How many people with EECS accounts are using the bash shell as their default shell?

  (We need to know that the default shell is stored in `/etc/passwd`)

```
papaggel:x:18084:2000:Manos Papagelis:/cs/home/papaggel:/cs/local/bin/bash
paras273:x:15708:10000:Parastoo Baghaei Ravari:/cs/home/paras273:/bin/false
pareto:x:9733:7000:Park Search Engine:/cs/home/pareto:/buonly
parham71:x:17252:10000:Parham Amani:/cs/home/parham71:/bin/false
paria:x:12757:3000:Paria Mehrani:/cs/home/paria:/cs/local/bin/tcsh
...
```

The power of pipelines

- Solution:

```
grep bash /etc/passwd | wc -l
```

Answer: 10

The power of pipelines

- Question 2:

  How many EECS accounts are there?

The power of pipelines

- Solution:

```
wc -l /etc/passwd
```

Answer: 2924

Another problem

- Question 3:

  How many people are running bash or tcsh right now?

More on grep and pipes

- Solution Steps:
  - Step 1: Display active processes using `ps`
    - `man ps`
    - `ps` normally shows processes associated with your terminal
    - use the options `aux` to display all processes

More on grep and pipes

– Step 2: Extract the processes running bash.

```
ps aux | grep bash
```

```
root        917  0.0  0.0 115640  1116 ?        S    Jan05   0:56 /bin/bash
papaggel   1623  0.0  0.0 116964  3756 pts/0    Ss+  09:10   0:00 -bash
papaggel  23113  0.1  0.0 116836  3516 pts/14   Ss   10:37   0:00 -bash
papaggel  23309  0.0  0.0 112664   976 pts/14   S+   10:38   0:00 grep
```

– Step 3: Weed out the grep process itself (man grep)

```
ps aux | grep bash | grep -v grep
```

More on grep and pipes

- Step 4: Keep only info about user names
  - Strip out only the name
  - Use `cut` to break each line into fields.
  - Two ways to do it:
    - `cut -d " " -f 1`
      - » Set the delimiter to be a space and select the first field.
    - `cut -c -8`
      - » Select characters from beginning to the 8th one

```
ps aux | grep bash | grep -v grep | cut -d " " -f 1
```

man cut

NAME

cut - remove sections from each line of files

SYNOPSIS

cut [OPTION]... [FILE]...

DESCRIPTION

Print selected parts of lines from each FILE to standard output.

-c, **--characters**=LIST output only these characters

-d, **--delimiter**=DELIM use DELIM instead of TAB for field delimiter

-f, **--fields**=LIST output only these fields

Use one, and only one of **-b**, **-c** or **-f**. Each LIST is made up of one range, or many ranges separated by commas. Each range is one of:

N N'th byte, character or field, counted from 1
N- from N'th byte, character or field, to end of line
N-M from N'th to M'th (included) byte, character or field

The order of bytes, characters or fields in the output will be identical to those in the input. With no FILE, or when FILE is -, read standard input.

More on grep and pipes

– Step 5: Sort them

```
ps aux | grep bash |grep -v grep | cut -d " " -f 1 | sort
```

– Step 6: Get rid of duplicates (if any)

```
ps aux | grep bash |grep -v grep | cut -d " " -f 1 | sort | uniq
```

– Step 7: And finally, count them…

```
ps aux | grep bash |grep -v grep | cut -d " " -f 1 | sort | uniq | wc -l
```

EXAMPLE SHELL SCRIPTS