Shell Programming

Shell Programming (bash)

- Commands run from a file in a subshell
- A great way to automate a repeated sequence of commands.
- File starts with `#!/bin/bash`
  - absolute path to the shell program
  - not the same on every machine.
- Can also write programs interactively by starting a new shell at the command line.
  - Tip: this is a good way to test your shell programs

General Principles

- Almost everything is treated as text strings
- Spaces matter regularly
- Remember the difference between return values and standard output
- Once you need a data-structure, use Python

Example

- In a file:

```
#!/bin/bash
echo "Hello World!"
```

- At the command line:

```
wolf% bash
sh-2.05b$ echo "Hello World"
Hello World
sh-2.05b$ exit
exit
wolf%
```

Commands

- You can run any program in a shell by calling it as you would on the command line.
- When you run a program like grep or ls in a shell program, a new process is created.
- There are also some built-in commands where no new process is created.

```
echo        test
set         shift
read        wait
exit
```

"man bash" to see all builtins

Variables

- local variables – spaces matter
  - `name=value` – assignment
  - `$name` – replaced by value of name
  - variables can have a single value or list of values.
- Single value:
  - `bindir="/usr/bin"`
- List of values (separated by spaces):
  - `searchdirs="~/tests $HOME/test2 ."`

Example:
($ is the default sh prompt)

```
$ bindir="/usr/bin"
$ searchdirs="~/tests $HOME/test2 ."
$ echo $searchdirs
~/tests /u/reid/test2 .
$ echo $bindir
/usr/bin
```

String Replacement

- Scripting languages are all about replacing text or strings, unlike other languages such as C or Java which are all about data structures.
- Variables are placeholders where we will substitute the value of the variable.
- Example:

```
iters="1 2 3 4"
for i in $iters; do
echo $i
done
```

=

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

Quoting

- Double quotes inhibit wildcard replacement only.
- Single quotes inhibit wildcard replacement, variable substitution and command substitution.
- Back quotes cause command substitution.
- Practice and pay attention.

| | | \

“today os `date`”
+ works fine
“you are $USER”
+ works fine
‘you are $USER’
+ does not expand

Single and double quotes are on the same key. Back quote is often on the same key as ~.

Quoting example

- `$ echo Today is date`
- `Today is date`
- `$ echo Today is `date``
- `Today is Thu Sep 19 12:28:55 EST 2002`
- `$ echo "Today is `date`"`
- `Today is Thu Sep 19 12:28:55 EST 2002`
- `$ echo 'Today is `date`'`
- `Today is `date``

require both read and execute permission to execute .sh file properly

Another Quoting Example

- What do the following statements produce if the current directory contains the following non-executable files?

```
a b c
$ echo *
$ echo ls *
$ echo `ls *`
$ echo "ls *"
$ echo 'ls *'
$ echo `*`
```

command not found: foo.txt

```
" – double quotes
' – single quote
` - back quote
```

Test

- The built-in command `test` is used to construct conditional statements in Bourne shell

-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

Control statements

- for loop

```
for color in red green blue pink
do
    echo The sky is $color
done
```

- if statements – if then elif then else fi

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

=

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

More on if

- If statements just check the return value of the command.
- test is just a command that returns a value.
- E.g.,

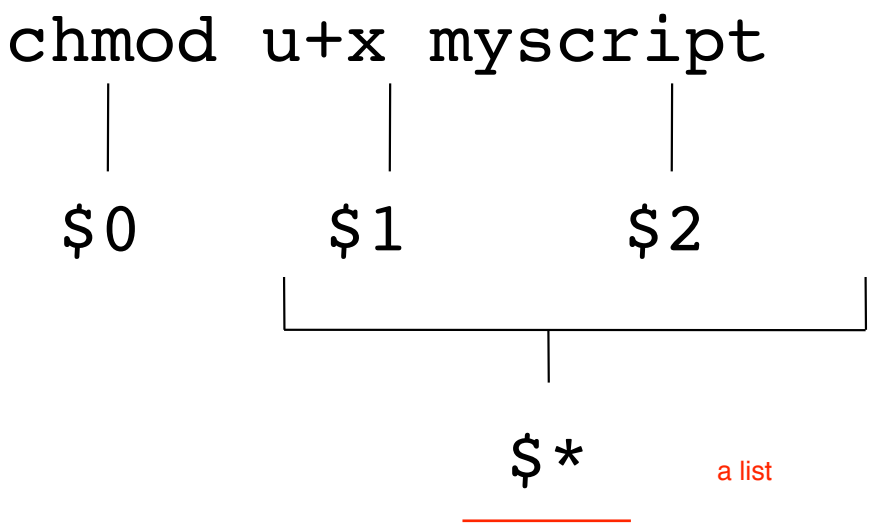
grep returns 0 if found, 1 if not found, 2 cannot exist/open

```
if grep name file
then
        echo found
else
        echo not found
fi
```

> /dev/null
+ send stdout to nowhere

Command line arguments

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



Positional Parameters

- Example:
  - (Remember to run `chmod u+x giant`)

giant one "two three" four
+ treated as 4 arguments… quotes disappear
+ the idea is that we use
for arg in "$*" instead of for arg in $*

`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
```

Positional Parameters

	What it references
$0	Name of the script
$#	Number of positional parameters not including $0
$*	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

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
```

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 $arg
done
```

Quotes mean arguments are all in one string.

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

One element for each argument.

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

Quotes in the arg list are preserved

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

Does not preserve quotes in arg list.

expr

- Since shell scripts work by text replacement, we need a special function for arithmetic.

```
x=1
x=`expr $x + 1`
y=`expr 3 \* 5`
```

- Or better yet: `y=$((3*5))`

String matching using expr

- `expr $string : $substring`

- Returns the length of matching substring at beginning of string.
- I.e., 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.

read

- read 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 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.

Subroutines

- You can create your own functions or subroutines:

```
myfunc() {
    arg1=$1
    arg2=$2
    echo $arg1 $globalvar
    return 0
}
```

- globalvar="I am global"
- myfunc num1 num2