Shell scripts (bash)

Lecture 1.2

Shell

- In addition to being an interpreter of user commands, shell also provides a high-level programming language
- The script may contain calls for any other tool
- It has variables, loops, conditionals etc.
- You can write sophisticated programs in "shell script"

Script may include call to any other tool

- Your sh program can use any Linux command, including any useful little utilities you write in C or any other programming language
- It's quite common for a complex package to include some C code and some shell scripts, and the shell scripts invoke the C programs

Useful script: connecting to teach.cs

- We can store any shell command in a file and invoke it by running this file
- Let's create a script which connects to teaching lab machines:

```
cat << EOF > cdf
> #!/bin/bash
> ssh wolf.teach.cs.toronto.edu
> EOF

This will create file cdf
chmod 700 cdf
./cdf
```

Now instead of typing the full path to the server, we just invoke ./cdf

Script with command-line arguments

- In file hello

```
#!/bin/bash
salutation="Hello, "
echo "The program $0 is now running "
for arg in $*
do
        salutation="$salutation $arg"
done
echo "$salutation"
```

Interactive script

```
secretname=marina
name=noname
echo "Try to guess the secret name!"
echo
until [[ "$name" = "$secretname" ]]
        do
                read -p "Your guess: " name
        done
echo "Very good."
```

Exit codes

- All shell programs are written in C
- By convention, if a C program succeeds, it exits with code 0, and it exists with a non-zero code in case of failure

If with exit codes

- Boolean values in sh are based on command exit status: "true" command in /bin is just exit 0; "false" is just exit 1.
- Example: command "foo" succeeds or fails:

```
if foo
then
   bar
else
   echo sorry, foo failed
   exit 1
fi
```

test

- A general testing utility exists for testing, called "test". See "man test".

```
if test 2 -lt 3
then
    echo "OK"
fi
```

- Equivalent to:

```
if [[ 2 < 3 ]]
```

Numeric and string comparisons

- "test" tests numeric relations, performs string comparisons and file tests.

  = is string-compare

  -eq is numeric equality (lt, gt, le, ge)

Testing for files

test -f blah

- succeeds iff blah exists and is a regular file

test -s blah

- succeeds iff blah exists and is a regular file and is not zero-sized

test -d blah

- succeeds iff blah exists and is a directory

expr

- Utility program which evaluates various expressions: see "man expr".
- For example, "expr 1 + 2" will output "3".
- All of these parameters have to be different tokens, i.e. different elements of argv (separated by spaces)

Example: increment x by 1

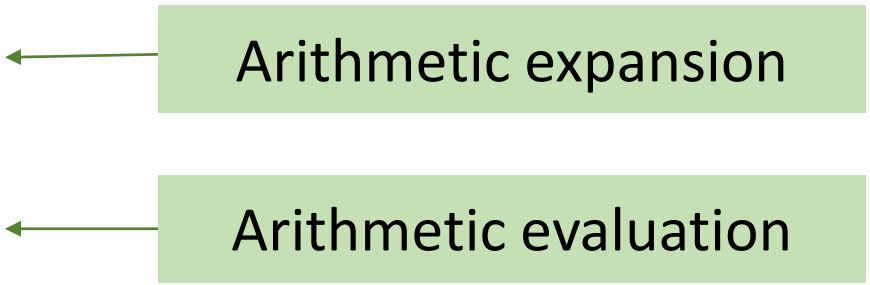
```
#!/bin/bash
x=1; echo $x
count=`expr $count + 1`
echo $count
```

Equivalent to:

```
x=$(( x + 1 ))
```

Or

```
(( $x = $x + 1 ))
```



Interpreting everything: example

- We need to suppress the interpretation of special characters.

- Example: How do we use the echo command to output an actual '>'? Suppose we want to print:

To forward your mail to user@host, type: echo user@host >.forward

- We can't just use:

echo To forward your mail to user@host, type: echo user@host >.forward

Why ?

Suppressing interpreter: 3 methods

1. **backslashes**

echo To forward your mail to user@host, type: echo user@host **\>**.forward

2. **single quotes**

echo **'**To forward your mail to user@host, type: echo user@host >.forward**'**

echo To forward your mail to user@host, type: echo user@host **'>'**.forward

echo To forward your mail to user@host, type: ec**'**ho user@host >.f**'**orward

(although the spaces within single quotes are prevented from separating argv members, which would matter for most commands other than echo)

3. **double quotes**

Suppressing interpreter: 3 methods

address=user@host

a. echo To forward your mail to $address, type: echo $address >.forward
b. echo 'To forward your mail to $address, type: echo $address >.forward'
c. echo "To forward your mail to $address, type: echo $address >.forward"

Which version gives the desired output?

Piping and forking commands

(a;b;c) | sort

(echo This is foo.c; cat foo.c; echo This is bar.c; cat bar.c) | lpr

Globbing in shell scripts

- command-line: "globbing" done by the shell.

- '*' matches any number of any character.
  *.c
  *x*y
- '?' matches any one character.
  a?.pdf
- [list of chars]
  a[1234].pdf
- [range]
  a[1-4].pdf
  use [a-z] to match any lower-case letter
  combine them: [a-xz] matches any lower-case letter except 'y'

What will be printed?

```
expr 2 * 3
```

```
$ mkdir +
$ expr 2 * 3
5
$
```

It depends upon what files are in the current directory, because '*' is substituted accordingly.

Printing all directories: example

```
for i in *
do
   if [[ -d "$i" ]]
   then
         echo "$i"
   fi
done
```

While loop

- "while" – same evaluation as for "if"
- Example: loop which counts 1 to 10:

```
i=0
while test $i -lt 10
do
   i=$( expr $i + 1 )
   echo $i
done
```

```
i=0
while [[ $i < 10 ]]
do
   i=$(( i + 1 ))
   echo $i
done
```

For loop

```
for i in hello goodbye
do
   echo $i, world
done

for i in $(seq 9)
do
        echo $( expr $i '*' $i )
done
```

What is printed here?

For loop with arithmetic evaluation

```
for (( c=1; c<=5; c++ ))
do
   echo "welcome $c times"
done
```