Test

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

`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.

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.,

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

Command line arguments

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

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
```

set and shift

- set – assigns positional parameters to its arguments.

```
$ set `date`
$ echo “The date today is $2 $3, $6”
The date today is Aug 27, 2001
```

- 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
```

Quotes 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 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` #need to escape *
```

- Can also be used for string manipulation, but we will mostly leave text manipulation for Python.

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
```

- 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.

NAME

cut - remove sections from each line of files

SYNOPSIS

cut [<u>OPTION</u>]... [<u>FILE</u>]...

DESCRIPTION

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

-c, **--characters**=<u>LIST</u>	output only these characters
-d, **--delimiter**=<u>DELIM</u>	use DELIM instead of TAB for field delimiter
-f, **--fields**=<u>LIST</u>	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.

The power of pipelines

- How many people with cdf accounts are using the bash shell as their default shell?
- First we need to know that the default shell is stored in `/etc/passwd`

```
g4wang:x:10461:1009:Wang Guoyu:/h/u3/g4/00/g4wang:/var/shell/bash
g4ali:x:10462:1009:Ali Muhammad:/h/u3/g4/00/g4ali:/var/shell/tcsh
g4lily:x:10463:1009:Hu Lily:/h/u3/g4/00/g4lily:/var/shell/tcsh
g4daniel:x:10464:1009:Chu Daniel C:/h/u3/g4/00/g4daniel:/var/shell/tcsh
g4yk:x:10465:1009:Kim Youngki:/h/u3/g4/00/g4yk:/var/shell/tcsh
g4kimukr:x:10466:1009:Kim Uk Rae:/h/u3/g4/00/g4kimukr:/var/shell/bash
g4kongja:x:10467:1009:Kong Jason:/h/u3/g4/00/g4kongja:/var/shell/tcsh
```

The power of pipelines

- Solution: (almost)

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

- Answer: 107

- How many CDF accounts are there?

```
wc /etc/passwd
```

- Answer: 5577

Another problem

- If I am logged into seawolf, how can I find out how many people are running bash or tcsh right now?
- 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.

```
root       1254  0.0  0.0  2480 1052 ?        S     2004   0:00 /bin/bash /
g1gros     4151  0.0  0.0  2484 1532 pts/23   S    Jan13   0:00 -bash
pgries    29010  0.0  0.0  3456 2464 pts/0    S    09:12   0:00 -bash
g1gros      865  0.0  0.0  2452 1464 pts/7    S    10:08   0:00 -bash
krueger    4228  0.0  0.0  1340  472 pts/6    S    11:57   0:00 grep bash
```

- Solution: `ps aux | grep bash`
- Step 3: Weed out the grep process (man grep)
- Solution :

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

More on grep and pipes

- Step 4: Get rid of duplicate 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

More on grep and pipes

- Now get rid of duplicates

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

- And finally, count them…

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

`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` and `xargs`

- `find . -name "*.java" -print`
  - Displays the names of all the Java files in directories in and below the current working directory.
- `xargs`
  - Build and execute command lines from standard input.

```
find . -name "*.java" -print | xargs grep "import junit"
```