Borne Shell

- Background
  - Early Unix shell that was written by Steve Bourne of AT&T Bell Lab.
  - Basic shell provided with many commercial versions of UNIX
  - Many system shell scripts are written to run under Bourne Shell
  - A long and successful history

Bourne Shell Programming

- Control structures
  - if … then
  - for … in
  - while
  - until
  - case
  - break and continue

if … then

- Structure

```
if test-command
   then
      commands
fi
```

```
Example:

if test "$word1" = "$word2"
   then
      echo "Match"
fi
```

3

test command

- Command test is a built-in command
- Syntax

```
test expression
[ expression ]
```

  - The test command evaluate an expression
  - Returns a condition code indicating that the expression is either true (0) or false (not 0)
- Argument
  - Expression contains one or more criteria
    - Logical AND operator to separate two criteria: -a
    - Logical OR operator to separate two criteria: -o
    - Negate any criterion: !
    - Group criteria with parentheses
  - Separate each element with a SPACE

4

Integer Test

- Test Operator for integers: int1 relop int2

Relop	Description
-gt	Greater than
-ge	Greater than or equal to
-eq	Equal to
-ne	Not euqal to
-le	Less than or equal to
-lt	Less than

Exercise

- Create a shell script to check there is at least one parameter
  - Something like this:

```
...
if test $# -eq 0
    then
        echo " you must supply at least one arguments"
        exit  1
  fi
...
```

Test for files' attribute

- The test built-in options for files

Option	Test Performed on file
-d filename	Exists and is a directory file
-f filename	Exists and is a regular file
-r filename	Exists and it readable
-s filename	Exists and has a length greater than 0
-u filename	Exists and has setuid bit set
-w filename	Exists and it writable
-x filename	Exists and it is executable
… …	… …

Exercise

- Check weather or not the parameter is a non-zero readable file name
  - Continue with the previous script and add something like

```
if [ -r  "$filename" –a –s "$filename" ]
then
    … …
fi
```

String Test

Criteria	meaning
String	True if string is not the null string
-n string	True if string has a length greater than zero
-z string	True if string has a length of zero
String1 = string2	True if string1 is equal to string2
String1 != string2	True if string1 is not equal to string2

Exercise

- Check users confirmation
  - First, read user input

```
echo  -n "Please confirm: [Yes | No] "
read user_input
```

  - Then, compare it with standard answer 'yes'

```
if  [  "$user_input"  =  Yes  ]
then
        echo "Thanks for your confirmation!"
fi
```

  - What will happen if no "" around $user_input and user just typed return?

if…then…else

- Structure

```
if test-command
   then
      commands
   else
      commands
fi
```

  - You can use semicolon (;) ends a command the same way a NEWLINE does.

```
if [ …  ]; then
   … …
fi

if [ 5 = 5 ]; then echo "equal"; fi
```


if…then…elif

- Structure

```
if test-command
   then
      commands
   elif test-command
    then
          commands
   .
   .
   .
    else
       commands
fi
```

for… in

- Structure

  for ***loop-index*** in ***argument_list***
  do
  commands
  done

  Example:

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

13

for

- Structure

  for ***loop-index***
  do
  commands
  done

  - Automatically takes on the value of each of command line arguments, one at a time. Which implies

    for arg in "$@"

14

while

- Structure

```
while test_command
do
        commands
done
```

Example:

```
while [ "$number" –lt 10 ]
do
        … …
        number=`expr $number + 1`
done
```

15

until

- Structure

```
until test_command
do
        commands
done
```

Example:

```
secretname=jenny
name=noname
until  [ "$name" = "$secretname" ]
do
       echo -e " Your guess: \c"
       read name
done
```

16

break and continue

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - Transfer control to the statement TO the done statement
  - Skip the test statements for the current iteration
  - Continues execution of the loop

17

Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index –le 3 ]; then
            echo continue
            continue
        fi
        echo $index
        if  [ $index –ge 8 ]; then
            echo "break"
            break
        fi
 done
```

18

case

- Structure

```
case  test_string in
pattern-1 )
    commands_1
      ;;
pattern-2 )
    commands_2
      ;;
… …
esac
```

- default case: catch all pattern

```
* )
```

19

case

- Special characters used in patterns

Pattern	Matches
*	Matches any string of characters.
?	Matches any single character.
[…]	Defines a character class. A hyphen specifies a range of characters
|	Separates alternative choices that satisfy a particular branch of the case structure

20

Example

```
#!/bin/sh
echo "\n Command MENU\n"
echo " a. Current data and time"
echo " b. Users currently logged in"
echo " c. Name of the working directory\n"
echo "Enter a,b, or c:  \c"
read answer
echo
case "$answer" in
  a)
     date
     ;;
  b)
     who
     ;;
  c)
     pwd
     ;;
   *)
     echo "There is no selection: $answer"
      ;;
esac
```


echo and read

- The backslash quoted characters in echo
  - \c suppress the new line
  - \n new line
  - \r return
  - \t tab
- Read
  - read variable1 [variable2 …]
    - Read one line of standard input
    - Assign each word to the corresponding variable, with the leftover words assigned to last variables
    - If only one variable is specified, the entire line will be assigned to that variable.

Example: bundle

```
#!/bin/sh
#bundle: group files into distribution package

echo "# To Uble, sh this file"
for i
do
   echo "echo $i"
   echo "cat >$i <<'END of $i' "
   cat $i
   echo "END of $i"
done
```

- Will this program work for sure?

23

Built-in: exec

- Execute a command:
  - Syntax: exec command argument
  - Run a command without creating a new process
    - Quick start
    - Run a command in the environment of the original process
    - Exec does not return control to the original program
    - Exec can be the used only with the last command that you want to run in a script
    - Example, run the following command in your current shell, what will happen?

      $exec who

24

Built-in: exec

- Redirect standard output, input or error of a shell script from within the script
  - exec < infile
  - exec > outfile 2> errfile
  - Example:

```
sh-2.05b$ more redirect.sh
exec > /dev/tty
echo "this is a test of redirection"

sh-2.05b$ ./redirect.sh 1 > /dev/null 2 >& 1
this is a test of redirection
```

25

Catch a signal: built-in trap

- Built-in trap
  - Syntax: trap 'commands' signal-numbers
  - Shell executes the commands when it catches one of the signals
  - Then resumes executing the script where it left off.
    - Just capture the signal, not doing anything with it
      trap ' ' signal_number
  - Often used to clean up temp files
  - Signals
    - SIGHUP 1 disconnect line
    - SIGINT 2 control-c
    - SIGKILL 9 kill with -9
    - SIGTERM 15 default kill
    - SIGSTP 24 control-z
    - …

26

Example

```
[ruihong@dafinn ~/cs3451]$ more inter
#!/bin/sh
trap 'echo PROGRAM INTERRUPTED' 2
while true
do
        echo "programming running."
        sleep 1
done
```

27

A partial list of built-in

- bg, fg, jobs — job control
- break, continue — change the loop
- cd, pwd — working directory
- echo, read — display/read
- eval — scan and evaluate the command
- exec — execute a program
- exit — exit from current shell
- export, unset — export/ remove a val or fun
- test — compare arguments

28

A partial list of builtin

- kill sends a signal to a process or job
- set sets flag or argument
- shift promotes each command line argument
- times displays total times for the current shell and
- trap traps a signal
- type show whether unix command, build-in, function
- umask file creation mask
- wait waits for a process to terminate.
- ulimit print the value of one or more resource limits

29

functions

- A shell function is similar to a shell script
  - It stores a series of commands for execution at a later time.
  - The shell stores functions in the memory
  - Shell executes a shell function in the same shell that called it.
- Where to define
  - In .profile
  - In your script
  - Or in command line
- Remove a function
  - Use unset built-in

30

functions

- Syntax

```
function_name()
{
   commands
}
```

- Example:

```
sh-2.05b$ whoson()
> {
> date
> echo "users currently logged on"
> who
> }
sh-2.05b$ whoson
Tue Feb  1 23:28:44 EST 2005
users currently logged on
ruihong  :0          Jan 31 08:46
ruihong  pts/1       Jan 31 08:54 (:0.0)
ruihong  pts/2       Jan 31 09:02 (:0.0)
```


Example

```
sh-2.05b$ more .profile
setenv()
{
        if [ $# -eq 2 ]
                then
                        eval $1=$2
                        export $1
                else
                        echo "usage: setenv NAME VALUE" 1>&2
        fi
}
sh-2.05b$. .profile
sh-2.05b$ setenv T_LIBRARY /usr/local/t
sh-2.05b$ echo $T_LIBRARY
/usr/local/t
```

Exercise

- Let's look at some system scripts
  - /etc/init.d/syslog
  - /etc/init.d/crond

33

Debugging Shell Scripts

- Display each command before it runs the command
  - Set the –x option for the current shell
    - $set –x
  - Use the –x to invoke the script
    - $sh –x command arguments
  - Add the set command at the top of the script
    - set –x
- Then each command that the script executes is preceded by a plus sign (+)
  - Distinguish the output of trace from any output that the script produces
- Turn off the debug with set +x

34

Summary

- Shell is a programming language
- Programs written in this language are called shell scripts.
  - Variable
  - Built-in
  - Control structure
  - Function
  - Call utilities outside of shell
    - find, grep, awk

35