Shell Scripts

Hour 15

- Objectives
  - Running shell scripts
  - Shell output: the echo and print commands
  - Command substitution
  - Conditional execution: the && and || operators
  - Flow of control statements: if, case, while, for, and select
  - Boolean expressions
  - Catching signals

It is possible to write programs in Bourne, Korn, or C shell

Shell scripts
- Are programs written in a shell language (e.g., Bourne, Korn, C Shell)
  - Bourne is a subset of Korn and is traditionally used for system maintenance (especially for start up and shutdown scripts)
  - Korn is popular for general system programming and product support
  - C Shell is similar to the C programming language but is not commonly used
- Are text files that are interpreted (i.e., they are not compiled)
- Have Unix commands as their basic statements
- Support simple flow-control statements

Execute scripts with the appropriate processor
- csh fileName
- sh fileName
- ksh fileName
Making Shell Scripts Executable

Changing a file’s execute permission mode

- Enter a “comment” that denotes the correct shell
  - `#!/bin/sh`
  - `#!/bin/ksh`
  - `#!/bin/csh`
  - The `#` character must be the first character in the file (i.e., there cannot be a blank line above or a space to the left)

- Make the script executable with the `chmod` command:
  `chmod a+x fileName`

- Run the program by entering the file name (followed by any appropriate options or arguments) and pressing enter:
  `fileName`

- You can run a script from any shell (your command interpreter shell can be different from the script shell)

Common Shell Script Features

These are the same in all three shells

- Comments begin with `#`
  - From `#` to the end of the line is a comment
  - Comments are ignored by the shell; they are there for humans

- Console output (screen)
  - `echo [-n] stuff to output [\c]`
    - Suppress printing the new-line character with `-n` (BSD/Solaris)
    - Suppress printing the new-line character with `\c` (SVR4)
  - Examples:
    - `echo The answer is $answer`
    - `echo -n "Enter the year: "`
    - `echo "Enter the year: \c"`

- Command substitution (implemented with grave accents)
  - Output of command is substituted as a string in place of command
  - `directory=`pwd`
  # Bourne and Kom
Here Documents

Scripts that provide a “standard in” to other commands

- When a script executes `command << word` it copies all text between the statement and word to command’s standard in

```
INSTALL="/var/compilers"

LIB=`sed -e "s/DIRECTORY/$INSTALL/" << DONE
DIRECTORY/pascal/lib
DONE`
```

Exit Status & Conditional Execution

Useful in shell scripts

- When a command or process terminates, it returns an exit status or value
  - 0 indicates success, non-zero [1-255] indicates an error
  - C program: `exit(0);` or `_exit(0);` or `return 0;` from `main`
  - Shell script: `exit 0`

- `command1 && command2`
  - Run `command2` only if `command1` exits successfully (i.e., if `command1` returns a zero exit value)
  - `gcc easter.c && a.out`

- `command1 || command2`
  - Run `command2` only if `command1` fails (i.e., if `command1` returns a non-zero exit value)
  - `gcc easter.c || echo compile error`
Bourne Shell Variables

Predefined by the shell

- **Variables**
  - Programmer picks variable names
  - Variables do not need to be defined/declared
  - $ is used to access the contents of a variable
  - $ is not used to set the contents of a variable

- **Predefined variables**
  - $? Exit status of last command
  - $$ Process ID of current shell process
  - $# Number of command line or positional parameters
  - $n 1 ≤ n ≤ 9; positional parameter n
  - The command **cal3 1 2002** assumes that cal3 is an executable script
    - $# is 2
    - $1 is 1
    - $2 is 2002

Bourne Shell Programming

Continued

- **Assignment**
  - `variable=value` # spaces on either side of = are not allowed
  - e.g.: `count=10` or `prompt="Press enter to continue"`

- **Console (keyboard) input**
  - `read varName`
  - `read choice`

- **Bourne uses the `expr` command to perform (int) arithmetic**
  - `*, /, %`
  - `+, -`
  - `=, >, >=, <, <=, !=`
  - `&`
  - `|`
  - e.g. `count=``expr $count + 1` # spaces around + are required!
Bourne / Korn Example

Simple shell script

```sh
#!/bin/sh

echo -n "Number of pennies: "
read money

echo "$money" pennies is equal to:

dollars=`expr $money / 100`
money=`expr $money % 100`

quarters=`expr $money / 25`
money=`expr $money % 25`

dimes=`expr $money / 10`
money=`expr $money % 10`

nickels=`expr $money / 5`

```

Shell Script Branches

Bourne and Korn shell scripts

- test and bracket notations are equivalent

```sh
if test expression
then
  commands
elif test expression
then
  commands
else
  commands
fi
```

- required spaces

```sh
if [ expression ]
then
  commands
elif [ expression ]
then
  commands
else
  commands
fi
```
Expressions

See `man test` for more expressions

- Expression evaluation is not built into the Bourne shell: use the Unix `test` command or the `[ ]` notation

- Expressions
  - `-d file`
  - `-f file`
  - `-r file`
  - `-w file`
  - `-x file`
  - `str1 = str2`
  - `str1 != str2`

- `if [ $# -gt 2 -o "$1" = "-h" ]`
- `if test $count -eq 0 ↔ if [ $count -eq 0 ]`

Hour 15 Unix Slide 11 of 20

Simple if Example

Bourne and Kom

- `-d fileName` is true if `fileName` exists and is a directory
- `!` changes true to false and false to true

# create .junk directory if it doesn't exist

```bash
if [ ! -d ${HOME}/.junk ]
then
    mkdir ${HOME}/.junk
fi
```

Hour 15 Unix Slide 12 of 20
### if-Statement Examples

Valid in Bourne and Korn shell scripts

```bash
if [ $# -gt 2 -o "$1" = "-h" ]
then
    echo "Usage: cal3 [ month [ year ]]
    exit
fi

if [ $# -eq 2 ]
then
    month=$1
    year=$2
else
    month=`date +%m`  # Command substitution
    year=`date +%Y`   # ` is the grave accent
fi
```

### Shell Script Constant Branches

Bourne and Korn Shell scripts

```bash
case expression in
    pat1) commands
        ;;
    pat2) commands
        ;;
    pat3 | pat4) commands
        ;;
esac

case "$1" in
    "quit") exit
        ;;
    "list") ls -l
        ;;
    *) echo "$1 is not recognized
esac
```
Shell Script Loops

Bourne and Korn Shell scripts

```bash
while [ expression ]
do
  commands
done

while [ "$1" != "" ]
do
  lpr -Plp $1
  shift     # $2 to $1
done

for variable in list
do
  commands
done

for file in junk money
do
  lpr -Plp $file
done
```

if / while Example

Nested structures

```bash
if [ "$1" = "-l" ]
then
  ls -l ${HOME}/.junk
elseif [ "$1" = "-p" ]
then
  rm -f ${HOME}/.junk/
else
  while [ "$1" != "" ]
  do
    mv $1 ${HOME}/.junk
    shift
  done
fi
```

# list contents of .junk
# purge .junk
# move all files to .junk
while / case Example

More nested structures

```bash
while test 1 -eq 1
do
echo "p purge the trashcan"
echo "l list the contents of the trashcan"
echo "L long listing of the contents of the trashcan"
echo "e exit"
echo -n "Please enter a choice: "
read choice
case $choice in
  "p" | "P")
    rm -f $HOME/.junk/*
    ;;
  "l")
    ls $HOME/.junk;
  "L")
    ls -l $HOME/.junk;
  "e" | "E")
    exit 0;
  *) echo Invalid option $choice
esac
done
```

Shell Script Functions

Bourne and Korn Shell Scripts

- Must be defined before use
  - Name and empty parentheses
  - Statements are enclosed in braces
  - Arguments are accessed by $1..$9
- Called by name (without parens)
- A useful organization is
  - global variable initialization
  - function definitions
  - “main” program

```bash
# initializations
variable=value

printvals()
{
  echo $1
}

# main stuff
printvals 100
```
Bourne Shell Functions

A simple example

```bash
#!/bin/sh
# a Bourne shell function. Must be defined before use.
# function arguments are also accessed by ($1..$9)

args()
{
    while [ "$1" != "" ]
    do
        echo $1
        shift
    done
}

echo $1 $2 # print shell command line arguments

# call the function with 11 arguments
args see the quick red fox jump over the lazy brown dog

echo $1 $2 # command line arguments haven't changed
```

Catching Signals

Signals are sent from one process to another

- `trap [ [ command ] sig1 ... ]`
- execute command (or shell function) if `sig1, sig2, ...` is received
- if `command` is `""` `sig1, sig2, ...` is/are ignored
- if `command` is absent, trap(s) for `sig1, sig2 ...` is/are reset