About the Shells

- The shells provide an interface for controlling programs and their input and output.
- The **Bourne** (**sh**), **C** (**csh**), **Korn** (**ksh**), and **Bash** (**Bourne-Again**) shells are line-oriented, with features dating back to the earliest UNIX shells for Teletype terminals
- All shells operate in a loop:

  1) Display a prompt
  2) Read input command line
  3) Perform substitutions in command line
  4) Execute command
  5) Loop to step 1

- The substitutions in step 3 may be of: variables, filenames, command output, aliases, depending on the shell. We will deal primarily with the Bash shell.

Command Sequencing and Grouping

- Commands can be sequenced with **;** or grouped with **()** as follows:

  $ date > text; who >> text
  $ (date; who) > text

- A ***cd*** within parentheses affects only the commands following up to the closing parenthesis:

  $ cd ; (cd / ; pwd ; ls -a) ; pwd

- Whitespace is optional around **;**, (**,**)

Command Substitution

- Many commands that take arguments ignore standard input; e.g., ***ls, file, echo, cd, rm***.
- ***ls*** prints a list of file names to standard output; ***file*** takes filenames as arguments and determines the file type.
- Suppose we wanted to use the output of ***ls*** as "input" to ***file***?

  WON'T WORK ==> ***$ ls | file*** <== WON'T WORK

The standard output of **ls** can be given as arguments to file using backquotes:

$ cd
$ file `ls`

a.out: ELF 32-bit LSB executable, ….
Hello.c: C program text

- The ***ls*** command in backquotes is run first; the shell uses its standard output as arguments to ***file***.
- Standard output of any command can be made into arguments to any other command:

  $ date

  Sat Dec 9 17:51:34 PST 2000

  $ echo The date is `date`

  The date is Sat Dec 9 17:51:43 PST 2000

  $ echo The date is `date | awk '{print $2, $3 ",", $6}'`

  The date is Dec 9, 2000

Defining Shell Variables

- Shell variables can hold single string values; created with =, and accessed with $:

  $ ub=/usr/bin
  $ cd $ub; pwd

  /usr/bin

- To include whitespace or metacharacters:

  $ curse='@$*&>##! is a legalized virus!'
  $ echo $curse

  @$*&>##! is a legalized virus!

- To include output from a command:

  $ dir=`pwd`

Metacharacter Suppression

- In addition to single quotes, double quotes and backslash can be used to suppress shell metacharacters, with the following rules:
  - Single quotes suppress the special meaning of ALL metacharacters
  - Double quotes suppress the special meaning of all metacharacters EXCEPT \ (backslash), $ and ` (backquote)
  - Backslash suppresses the special meaning of the immediately following character (except between '…..')
- Variable and command substitution occur between "……" but NOT between '...'
- **Shell metacharacters** are those with non-literal meaning; e.g. the redirection and pipe symbols: **>, <, |**
- Spaces, tabs are metacharacters: they separate arguments.
- Single quotes **suppress** metacharacter **interpretation:**

  $ echo howdy | wc -c # one arg to echo
  6

  $ echo howdy '|' wc -c # four args to echo
  howdy | wc -c

  $ echo '…..howdy | wc -c' # one arg to echo
  …..**howdy | wc -c**

- Patterns for ***grep*** and ***sed*** often contain metacharacters; it's a good habit to surround arguments with single quotes:

  $ grep '|' chap2

  $ sed 's/>/>>/g' /foo/bar/readme

About Shell Functions

- A **shell function** associates a name with a list of commands; when the name is given as a command, the list of commands is executed.
- General form:

  name() { list }
- For example

  $ mydate() { date | awk '{ print $2, $3 ",", $6 }'; }
  $ mydate

  Jan 19, 2001

Function Arguments

- Arguments to shell functions can be accessed through **positional parameters***:*

  $1, **$2, ...,$9.**
- A shell function may be many lines long; during function definition, the secondary prompt PS2 (default ">") is displayed:

```
$ nd() {
> cd $1
> PS1="`pwd` $ "
> }
$ nd /bin
/bin $
```

Shell Scripts

- A sequence of shell commands stored in a file is a **shell script.**
- The **.** (dot) built-in causes the shell to execute commands from a script

```
$ cat f_defs
echo Defining nd and mydate
nd() { cd $1; PS1="`pwd` $ "; }
mydate() { date | awk ' { print $2, $3 ",", $6 }'; }
$ . f-defs
Defining nd and mydate
$ nd /bin
/bin $
```

- Alternatively, shell scripts can be executed by setting the execute bit on the file.

The awk Programming Language

awk is named after its authors: Al Aho, Peter Weinberger and Brian Kernighan.

- It is a **full programming** language, supporting:
  - built-in and user-defined variables
  - arithmetic and string operations
  - loops and branches
  - built-in and user-defined functions
- **awk**'s simplest and most common use is to select and change the order of fields.
- ***awk*** treats each input line as a sequence of **fields** delimited with whitespace. Fields can be selected by number and printed:

  $ awk '{ print $2 }'
  UNIX Lives
  Lives
  Linux is UNIX
  is
  ^d

- Single quotes make the print **action** text a single argument to awk, with no metacharacter interpretation.
- ***awk*** can be used to reformat output of other commands:

  $ date
  Fri Jan 19 12:11:49 PST 2001

  $ date | awk '{ print $2 $3 $6 }'
  Jan192001

  $ date | awk '{ print $2, $3, $6 }'
  Jan 19 2001

  $ date | awk '{ print $2, $3, ",", $6 }'
  Jan 19, 2001

- Note use of double-quoted strings in print action:

  $ echo Aman Abdulla | awk '{ print "Hello " $1 }'
  Hello Aman

- ***awk***'s field separator can be changed with the -F option; for example, **-F:** makes colon the field separator.

- For example we can print out the login names of all users in the **passwd** file as follows:

```
$ awk -F: '{ print $1 }' /etc/passwd
$ grep /home /etc/passwd | awk -F: '{ print $1 }'
```

User-defined Variables

- As we saw earlier, shell variables have single string values, and are set with = (equal sign):

```
$ one=1 msg="Hello, World"
```

- Values are accessed using **$**

```
$ echo one msg; echo $one $msg
one msg
 1 Hello, World
```

- To concatenate variable values with leading or trailing text, enclose variable names in {}:

```
$ echo $onetwo          # no such variable - empty string

$ echo ${one}two        # value of variable one, text two
1two
```

Positional Parameters and Special Variables

- Arguments to a shell script are accessed by positional parameters:

$1, **$2,..., $9**	Arguments 1 through 9
$0	Script file name
$*	All Positional parameters (from 1)

- Other shell special variables include

$#	Number of positional **parameters** to script
$$	Shell process ID number Exit status of last command
&?	Exit status of last command

The *shift* and *set* Built-in Commands

- The ***shift*** command shifts positional parameters:

```
$ cat shift_test
echo $0 $1 $5 $9 " Args:" $*, No: $#
 shift
 echo $0 $1 $5 $9 " Args:" $*, No: $#

$ shift_test A B C D E F G H I J
shift_test A E I Args: A B C D E F G H I J, No: 10
shift_test B F J Args: B C D E F G H I J, No: 9
```

- The ***set*** command sets the positional parameters to its arguments:

```
$ cat set_test
date
set `date`
echo The date is $2 $3, $6

$ set_test
Fri Jan 19 12:29:07 PST 2001
The date is Jan 19, 2001
```

The *for* Loop

- General form

```
for var [in val_list]
do
    commands
done
```

- For example, to create 10 temp files

```
$ for i in 0 1 2 3 4 5 6 7 8 9
> do
>   cp /dev/null temp_${i}
> done
```

If no value list is given, $* **(the** positional parameter list) is assumed:

```
cat my_print
for fname
do
        lpr $fname; ecbo "${fname}: printed
done

$ my_print code.c letter
code.c: printed
letter: printed
```

The *case* Branch

- General form:

```
case word in
    pattern) commands ;;
esac
```

- *word* is usually a **variable substitution;** pattern uses filename expansion-like metacharacters, with | for "or".

- A ***case*** branch can be the body of a ***for*** loop

```
$ cat term_type
for i
do
    case ${i} in
        tty[O-3])          echo "${i}: TVI 950"       ;;
        tty[45] |ttya)     echo "${i}: DEC VT 220" ;;
        *)                 echo "${i}: Unknown"     ;;
    esac
done

$ term_type ttya tty3 tty07
ttya:    DEC VT 220
tty3:    TVI 950
tty07:   Unknown
```

The *test* Command

- The ***test*** command evaluates an expression and yields an exit code of 0 for true, non-0 for false.

- General form:

    test expr
    or
    [expr]

- ***test*** has numerous primitives for determining features of files and strings:

```
-f file      # true if file exists, is regular file
-d file      # true if file exists, is directory
-x file      # true if file exists, is executable
-s file      # true if file exists, has size > 0

s1 = s2              # true if strings s1, s2 are identical
s1 != s2             # true if s1, s2 not identical
n1 -eq n2            # true if integers n1, n2 equal (also -ne, -gt, -ge, -lt, -le)
```

- Combinations

```
test -f temp -a -x temp              # -a means and
[ -f temp -o -d temp ]               # -o means or
test ! -d temp                       # ! means not
[ ${i} = 'ready' -a \( -f temp -o -d temp -o -d temp \) ]    # parens for grouping
```

- NOTE the use of **whitespace**!

The if/else Branch

- General form

```
if commands
    then commands
    [elif commands then commands]
    [else commands]
fi
```

The ***commands*** following **if** are executed; exit status 0 from last command means true; none-0 means false.

```
if [ -f ${i} ]
then
    echo ${i} is a regular file
else
    echo ${i} is not a regular file
fi
```

The *while* and *until* Loops

- The general form of the ***while*** loop is:

```
while commands
do
        commands
done
```

- For example:

```
while true
do
      sleep 60
      who
done
```

- The general form of the ***until*** loop is:

```
until commands
do
       commands
done
```

- For example:

```
until who | grep Bill
do
     sleep 30
done
echo Bill is logged in!
```

Filtering with Loops and Branches

- Loop or branch output may be filtered; for example:

```
$ cat sys_hogs
for user in `awk -F: '{ print $1 }' /etc/passwd`
do
        if [ ${user} != 'root' -a ${user} != 'aman' ]
        then
                home=`awk -F: "/^${user}:/ { print \\$6 }"  /etc/passwd`
                usage=`du -s ${home} | awk '{ print $1 }'`
                echo ${usage} ${user}
        fi
done | sort -nr | head | awk '{ print $2 ":", $1 }'
```

Complex scripts often require debugging

```
$ sh -x sys_hogs            # trace execution
```

Redirecting Standard Error

- The standard input, output and error streams are associated with **file descriptors :**

  standard input - 0 standard output - 1 standard error - 2

- < redirects 0 (standard input)
- > and >> redirect 1 (standard output)
- Preceding > or >> with 2 redirects standard error

```
myprog < foo > foo.out 2> foo. err
```

- Standard error may be combined with standard output

```
myprog < foo > foo.allout 2>&1
```

The *read* Command

- The ***read*** command waits for a user input line and assigns the input to variables:

```
$ read line
This is a test

$ echo $line
This is a test
```

$ read word rest
This is another test

$ echo $word
This

$ echo $rest
is another test

Arithmetic with *expr*

- The shell has no built-in arithmetic operations.
- The ***expr*** command provides arithmetic and other facilities for use in shell scripts or other contexts.
- ***expr*** takes an arithmetic expression as arguments and prints the value to standard output; metacharacters must be suppressed:

```
$ expr 7 \* \( 2 + 3 \)
35
$ a=1; a=`expr $a + 1 `; echo $a
2
```