Software Tools

Shell Programming

Shells

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

Shell Scripts

- A shell script is just a file containing shell commands, but with a few extras:
  - The first line of a shell script should be a comment of the following form:

```
#!/bin/sh
```

    for a Bourne shell script. Bourne shell scripts are the most common, since C Shell scripts have buggy features.

  - A shell script must be readable and executable.

```
chmod u+rx scriptname
```

  - As with any command, a shell script has to be “in your path” to be executed.
    - If “.” is not in your PATH, you must specify “`./scriptname`” instead of just “`scriptname`”

Shell Script Example

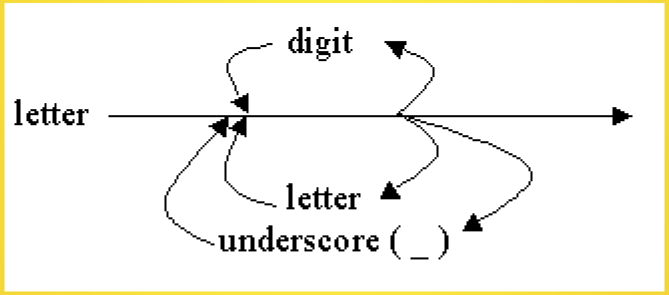
- Here is a “hello world” shell script:

```
$ ls -l
-rwxr-xr-x  1 horner   48 Feb 19 11:50 hello*
$ cat hello
#!/bin/sh
# comment lines start with the # character
echo "Hello world"
$ hello
Hello world
$
```

- The echo command functions like a print command in shell scripts.

Shell Variables

- The user variable name can be any sequence of letters, digits, and the underscore character, but the first character must be a letter.



- To assign a value to a variable:

```
number=25
name="Bill Gates"
```

- There cannot be any space before or after the “=“
- Internally, all values are stored as strings.

Shell Variables

- To use a variable, precede the name with a “$”:

```
$ cat test1
#!/bin/sh
number=25
name="Bill Gates"
echo "$number $name"
$ test1
25 Bill Gates
$
```

User Input

- Use the `read` command to get and store input from the user.

```
$ cat test2
   #!/bin/sh
   echo "Enter name: "
   read name
   echo "How many girlfriends do you have? "       read number
   echo "$name has $number girlfriends!"
   $ test2
   Enter name:
   Bill Gates
   How many girlfriends do you have?
   too many
   Bill Gates has too many girlfriends!
```

User Input

- `read` reads one line of input from the keyboard and assigns it to one or more user-supplied variables.

```
$ cat test3
#!/bin/sh
    echo "Enter name and how many girlfriends:"
    read name number
    echo "$name has $number girlfriends!"
    $ test3
    Enter name and how many girlfriends:
    Bill Gates 63
    Bill has Gates 63 girlfriends!
    $ test3
    Enter name and how many girlfriends:
    BillG 63
    BillG has 63 girlfriends!
    $ test3
    Enter name and how many girlfriends:
    Bill
    Bill has  girlfriends!
```

- Leftover input words are all assigned to the last variable.

$

- Use a backslash before $ if you really want to print the dollar sign:

```
$ cat test4
#!/bin/sh
echo "Enter amount: "
read cost
echo "The total is: \$$cost"
$ test4
Enter amount:
18.50
The total is $18.50
```

$

- You can also use single quotes for printing dollar signs.
- Single quotes turn off the special meaning of all enclosed dollar signs:

```
$ cat test5
#!/bin/sh
   echo "Enter amount: "
   read cost
   echo 'The total is: $' "$cost"
   $ test5
   Enter amount:
   18.50
   The total is $ 18.50
```

expr

- Shell programming is not good at numerical computation, it is good at text processing.
- However, the `expr` command allows simple integer calculations.
- Here is an interactive Bourne shell example:

```
$ i=1
$ expr $i + 1
2
```

- To assign the result of an `expr` command to another shell variable, surround it with backquotes:

```
$ i=1
$ i=`expr $i + 1`
$ echo "$i"
2
```

expr

- The `*` character normally means “all the files in the current directory”, so you need a “\” to use it for multiplication:

```
$ i=2
$ i=`expr $i \* 3`
$ echo $i
6
```

- `expr` also allows you to group expressions, but the “(“ and “)” characters also need to be preceded by backslashes:

```
$ i=2
$ echo `expr 5 + \( $i \* 3 \)`
11
```

expr Example

```
$ cat test6
#!/bin/sh
   echo "Enter height of rectangle: "
   read height
   echo "Enter width of rectangle: "
   read width
   area=`expr $height \* $width`
   echo "The area of the rectangle is $area"
   $ test6
   Enter height of rectangle:
   10
   Enter width of rectangle:
   5
   The area of the ractangle is 50
   $ test6
   Enter height of rectangle:
   10.1
   Enter width of rectangle:
   5.1
   expr: non-numeric argument
```

Does not work for floats!

Backquotes: Command Substitution

- A command or pipeline surrounded by backquotes causes the shell to:
  - Run the command/pipeline
  - Substitute the output of the command/pipeline for everything inside the quotes
- You can use backquotes anywhere:

```
$ whoami
gates
$ cat test7
#!/bin/sh
   user=`whoami`
   numusers=`who | wc -l`
   echo "Hi $user! There are $numusers users logged on."
   $ test7
   Hi gates! There are      6 users logged on.
```

Control Flow

- The shell allows several control flow statements:
  - `if`
  - `while`
  - `for`

if

- The `if` statement works mostly as expected:

```
$ whoami
clinton
$ cat test7
#!/bin/sh
   user=`whoami`
   if [ $user = "clinton" ]
   then
   echo "Hi Bill!"
   fi
   $ test7
   Hi Bill!
```

- However, the spaces before and after the square brackets [ ] are required.

if then else

- The `if then else` statement is similar:

```
$ cat test7
#!/bin/sh
user=`whoami`
if [ $user = "clinton" ]
then
        echo "Hi Bill!"
else
        echo "Hi $user!"
fi
$ test7
Hi horner!
```

if elif else

- You can also handle a list of cases:

```
$ cat test8
#!/bin/sh
   users=`who | wc -l`
   if [ $users -ge 4 ]
   then
   echo "Heavy load"
   elif [ $users -gt 1 ]
   then
   echo "Medium load"
   else
   echo "Just me!"
   fi
   $ test8
   Heavy load!
```

Boolean Expressions

- Relational operators:

```
-eq, -ne, -gt, -ge, -lt, -le
```

- File operators:

```
-f file    True if file exists and is not a directory
-d file    True if file exists and is a directory
-s file    True if file exists and has a size > 0
```

- String operators:

```
-z string  True if the length of string is zero
-n string  True if the length of string is nonzero
s1 = s2    True if s1 and s2 are the same
s1 != s2   True if s1 and s2 are different
s1  True if s1 is not the null string
```

File Operator Example

```
$ cat test9
#!/bin/sh
if [ -f letter1 ]
then
        echo "We have found the evidence!"
        cat letter1
else
        echo "Keep looking!"
fi
$ test9
We have found the evidence!
How much would it cost to buy Apple Computer?
Best,
Bill
```

And, Or, Not

- You can combine and negate expressions with:

`-a`	And
`-o`	Or
`!`	Not

```
$ cat test10
#!/bin/sh
if [ `who | grep gates | wc -l` -ge 1 -a `whoami` != “gates" ]
then
   echo "Bill is loading down the machine!"
 else
   echo "All is well!"
 fi
 $ test10
 Bill is loading down the machine!
```

while

- The `while` statement loops indefinitely, while the condition is true, such as a user-controlled condition:

```
$ cat test11
#!/bin/sh
resp="no"
while [ $resp != "yes" ]
do
echo "Wakeup [yes/no]?"
   read resp
   done
   $ test11
   Wakeup [yes/no]?
   no
   Wakeup [yes/no]?
   y
   Wakeup [yes/no]?
   yes
   $
```

while

- `while` can also do normal incrementing loops:

```
$ cat fac
#!/bin/sh
echo "Enter number: "
read n
fac=1
i=1
while [ $i -le $n ]
do
fac=`expr $fac \* $i`
i=`expr $i + 1`
   done
   echo "The factorial of $n is $fac"
   $ fac
   Enter number:
   5
   The factorial of 5 is 120
```

break

- The `break` command works like in C++, breaking out of the innermost loop :

```
$ cat test12
#!/bin/sh
while [ 1 ]
do
echo "Wakeup [yes/no]?"
   read resp
   if [ $resp = "yes" ]
   then
   break
   fi
   done
   $ test12
   Wakeup [yes/no]?
   no
   Wakeup [yes/no]?
   y
   Wakeup [yes/no]?
   yes
   $
```