Lecture # 7 – Programming the Bourne Shell (Chapter 8)

- Reading user input

  The read command is used to retrieve information from the keyboard.

  Obtain one line of input, and place the results in a variable called “answer”:
  $ read answer

  Get one line of input, and place the first word in “first”, and remainder of line in “last”
  $ read first last

- Arithmetic

  There are no constructs in Bourne shell to support arithmetic calculations.

  Integer math is done with the expr command.
  Floating point math can be done with a variety of commands including bc.

  Examples:

  ```
  $ expr 1 + 4
  5

  $ expr 4 \* 4
  16

  $ num=`expr $num + 1`
  ```

  Examples:

  ```
  $ n=`echo “scale=3; 13 / 2” | bc`
  $ echo $n
  6.500
  ```

- If Command

  ```
  if <cond>
  then
          <Commands>
  else
          <Commands>
  fi
  ```

  <cond> can be any command, a return value of zero is true, and nonzero is false

else clause is optional
The else-if type construct is “elif”.

There is a program called test, which is helpful for standard comparisons.

For example: if test “$word1” = “$word2”; then
Can also use [] to call test (example: if [$# = 0]; then) NOTICE SPACES AROUND []
See man page for test, and man page for sh.

String tests:

String1 = string2	string1 is equal to string2
String1 != string2	string1 is not equal to string2
String	string is not null
-z string	length of string is zero
-n string	length of string is non-zero

Integer tests:

int1 –eq int2	int1 is equal to int2
int1 –ne int2	int1 is not equal to int2
int1 –gt int2	int1 is greater than int2
int1 –ge int2	int1 is greater or equal to int2
int1 –lt int2	int1 is less than int2
int1 –le int2	int1 is less than or equal to int2

Logical tests:

Expr1 –a expr2	logical AND
Expr1 –o expr2	logical OR
! expr	logical NOT

File tests:

-d filename	directory existence
-f filename	file existence (not a directory)
-r filename	file exists and is readable
-s filename	file is nonzero size

Examples:

```
if  [ $# -ne 3 ]
then
        echo “usage: $0 <arg1> <arg2>”
        exit 1
fi
```

```
if [ $age –fe 0 –a $age –lt 13 ]
then
        echo “a child is a garden of verses”
elif [ $age –ge 13 –a $age –lt 20 ]
then
        echo “rebel without a cause”
else
        echo “other”
fi

if [ -d $file ]
then
        echo “$file is a directory”
fi
```

- Case

```
case string in
        Pattern1)
                Commands
                ;;
        *)
                Default commands
                ;;
esac
```

The first pattern to match determines the commands to be executed.
Can use simple regular expressions (*, ?, [], |)

Example:

```
read letter
case “$letter” in
        a|A)    echo “You entered A”
                ;;
        b|B)    echo “You entered B”
                ;;
        *)      echo “You did not enter A, or B”
                ;;
esac
```

- For Command

```
for loop-index in arglist
do
        commands
done
```

Loop executes once for each value in arglist; begins with do, and stops with done

Example:

```
$  cat fruit
for fruit in apples oranges pears bananas
do
        echo $fruit
done

$  fruit
apples
oranges
pears
bananas
```

Another variation of the for loop:

```
for loop-index
do
        commands
done
```

This version runs once for each command line argument with loop-index being each arg.

Example with command substitution:

```
for file in `ls $dir`
do
        echo $file
done
```

- While Command

```
while <cond>
do
        commands
done
```

```
$ cat count
number=0

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

- Until Command

  Very similar to the while loop, but exits when <cond> is true instead of false

```
until <cond>
do
        commands
done
```

  Example:

```
$ cat until.sh
secret=jenny
name=noname
echo Try and guess the secret
echo
until [  "$name" = "$secretname" ]
do
        echo "Your guess: \c"
        read name
done
echo Very good.
```

- Break and Continue

  Break transfers control to the statement following "done".
  Continue transfers control to the "done" statement

- I/O redirection and subshells

  Input can be piped or redirected to a loop from a file. Output can also be piped or redirected to a file from a loop. The shell starts a subshell to handle the I/O redirection and pipes.

  Any variables defined within the loop will not be known to the rest of the script when the loop terminates.

Example:

```
cat $1 | while read line
do
        [ $count –eq 1 ] && echo “processing file $1…” > /dev/tty
        echo $count $line
        count=`expr $count + 1`
done > temp.$$
```

Example:

```
while read line
do
        echo $line
done < testing > outfile
```