Lecture #11 – Programming the Korn Shell (Chapter 12)

- Reading user input

  Similar to Bourne shell (i.e. $ read answer)

  New features:

```
# display prompt and read response from single command
$ read response?”Do you feel okay?”

# read from file descriptor number 3
$ read –u3 line
```

  Examples:

```
while read –u3 line1 && read –u4 line2
do
        print “$line1:$line2”
done 3< file1 4< file2
```

- Math

```
$ typeset –i num
$ num=hello
/bin/ksh:  hello:  nad number
$ num=5 + 5
/bin/ksh: +: not found
$ num=5+5
$ echo $num
10
$ num=”4 * 6”
$ echo $num
24

$ num=15
$ typeset –i2 num
$ print $num
2#1111
$ typeset –i8 num
$ print $num
8#17
```

The let command and (( )):

```
$  i=5
$  let i=i+1
$  print $i
6

$  (( i = i * 6)
$  print $i
36
```

- Control flow commands

In general, Bourne shell syntax will work here (note a few additions)

Ksh supports a new version of the test command using [[ ]]:

String tests:

string = pattern	String matches pattern
string != pattern	String does not match pattern
string1 < string2	ASCII value of string1 is less than string2
string1 > string2	ASCII values of string1 is greater than string2
-n string	string is nonzero in length, nonnull parameter
-z string	string is zero in length, null parameter

Examples:

```
read answer

if [[ $answer = [Yy]* ]]
then
        echo "yes"
fi
```

Binary file testing:

file1 –nt file2	File1 is newer than file2
file1 –ot file2	File1 is older than file2
file1 –ef file2	File1 is another name for file2

Logical operators:

&&	Logical AND, replaces -a
||	Logical OR, replaces -o

File tests:

-a file	file exists
-e file	file exists
-L file	file exists and is a symbolic link
-O file	file exists and owned by UID of running shell
-G file	same as -O but for group
-S file	file exists and is a socket

Numeric testing (can use let command here):

```
if (( $# < 1 ))
then
        print “usage: $0 <number>” 1>&2
        exit 1
fi
```

- select command

new command to display menu

Syntax:

```
select varname [in arg...]
do
        cmds
done
```

Example:

```
PS3=”Please enter which fruit: “

select fruit in apple banana orange
do
        case $REPLY in
                1)  echo “apple”
                    break;;
                2)  echo “banana”
                    break;;
                3)  echo “orange”
                    break;;
done
```

- Getopts (Option Processing)

UNIX Conventions for command line options

```
$ ls –l –r –t
$ ls –lrt
$ cc –o prog prog.c
```

getopts makes processing these options easier
getopts optstring varname [arg …]

optstring is a list of the valid option letters (: follows letter if that option takes arg)
Leading colon means allows you to handle errors with "?" case
Example: "dxo:lt:r" means –d –x –o –l –t –r are valid options, and –o and –t take args

Varname is the variable to use for options (will use cmd line args if not specified)

OPTIND is 1 when scripts starts, and increments after each getopts call
OPTARG is the value of the argument for the option if one is required

Example:

Let's say we want a program to take:

-b	to ignore white space at the start of input lines
-t <dir>	use this directory for temporary files
-u	translate all output to uppercase

```
SKIPBLANKS=
TMPDIR=/tmp
CASE=lower

while getopts :bt:u arg
do
        case $arg in
        b)
                SKIPBLANKS=TRUE;;
        t)
                if [ -d "$OPTARG" ]
                then
                        TMPDIR=$OPTARG
                else
                        print "$0: $OPTARG is not a directory."
                        exit 1
                fi;;
        u)
```

```
                        CASE=upper;;
                :)
                        print “$0: You must apply an argument to $OPTARG.”
                        exit 1;;
                \?)
                        print “Invalid option $OPTARG ignored”;;
                esac
        done
        shift $((OPTIND-1))
```

Using VI

- vi History

  The original UNIX editor was called **ed** (line based)
  Later, **ex** was introduced as a superset to **ed** (added optional screen mode)
  Screen mode was so popular; they made a hard link to vi, which starts ex in screen mode
  Linux introduced “vim” which is “vi” with a few added improvements

- General

  Commands are case sensitive
  vi uses a work buffer (i.e. chg are not made to your file until you write or save & exit)
  You can write to a different file name with (:w filename)
  vi –r filename to recover from a crashed terminal session

- Display

  Status is shown on the last line (often line 24)

  Sometimes text lines will be shown as @ and can be redrawn with ^L or ^R from command mode

  ~ lines indicate positions beyond the end of the file

- Cursor Movement

h, j, k, l	Move cursor left, down, up, or right (If you type a number, then h, j, k, or l you will move that many char)
w	forward word
b	back one word
H	Go to top of screen
M	Go to middle of screen
L	Go to bottom of screen
^D	Down a half screen
^U	Up a half screen
^F	Forward full screen
^B	Back full screen
#G	Go to a specific line number
G, $	Go to end of file

- Editing commands

I	Go to beginning of line, and change to insert mode
A	Go to end of line, and change to insert mode
o, O	Open a blank line below (above), and change to insert mode
r	Replace single character
R	Replace (overwrite) until <ESC>
^V	Escapes the next character (so you can type special characters)
u	undo

See 'd' commands on p. 434

dd	delete current line
dw	delete word
d/<text>	delete forward up to but not including the next occurrence of "text"

See 'c' commands on p. 435

cw	change to end of word
cc	change current line

- Search and Replace

/string/<return>	Search for string (can be regular expression)
/	Repeat previous search (n)
?	Find in reverse direction (N)
:[address]s/search/replace/g	address is current line if omitted otherwise, address can be line number or range (. is current line, % is whole buffer, $ is last line) g is for replacing multiple occurrences on same line

- Miscellaneous

J	Join current line with next line
^G	Display status information
:f	File Information
.	Repeat previous command
yy	Yank current line to general buffer
p, P	put yanked line from general buffer below (above) current line

- Named Buffers

  There are 26 named buffers (identified by lowercase letter)

“[a-z]#yy	Yank # lines into named buffer [a-z]
“[a-z]p	Put lines from named buffer [a-z] below current line

- Read and write

:r file	Read a file and place contents at current line
:w[!] file	Write to another file name (! Forces)
:[address]w >> file	Write a range of lines and append to a file