CHAPTER 15

Kron Shell

In this chapter we will summaries some of the important aspects of Korn Shell in relation to Bourn Shell. The following table explains the metacharacters in Korn shell. If one observes, they are almost same as Bash.

Table 15.1 : Korn Shell Metacharacters

Metacharacter	Description
Syntactic	
|	Separates commands that are part of a pipeline.
&&	Runs the next command if the current command succeeds.
||	Runs the next command if the current command fails.
;	Separates commands that should be executed sequentially.
;;	Separates elements of a case construct.
&	Runs commands in the background.
( )	Groups commands in a sub-shell as a separate process.
{ }	Groups commands without creating a sub-shell.
Filename	
/	Separates the parts of a file's pathname.
?	Matches any single character except a leading dot (.).
*	Matches any character sequence except a leading dot (.).
[ ]	Matches any of the enclosed characters.
~	Specifies a home directory when it begins a filename.
Quotation	
\	Specifies that the following character should be interpreted literally; that is, without its special meaning to the shell.
`...'	Specifies that any of the enclosed characters (except for the `) should be interpreted literally; that is, without their special meaning to the shell.

"..."	Provides a special form of quoting. Specifies that the dollar sign ($), ` (grave accent), \ (backslash), and) (close parenthesis) characters keep their special meaning, while all other enclosed characters are interpreted literally; that is, without their special meaning to the shell. Double quotes (" ") are useful in making variable assignments.
Input/Output	
<	Redirects input.
>	Redirects output to a specified file.
<<	Redirects input and specifies that the shell should read input up to a specified line.
>>	Redirects output and specifies that the shell should add output to the end of a file.
>&	Redirects both diagnostic and standard output and appends them to a file.
Substitution	
${...}	Specifies variable substitution.
%	Specifies job number substitution.
`...`	Specifies command output substitution.

15.1 A NOTE ON REGULAR EXPRESSIONS IN KORN SHELL

Ksh has its own regular expressions.

Use an * for any string. So to get all the files ending it .c use *.c.

A single character is represented with a ?. So all the files starting with any sign followed bye 44.f can be fetched by: ?44.f.

Especially in ksh there are quantifiers for whole patterns :

?(pattern) matches zero or one times the pattern.

***(pattern)** matches any time the pattern.

+(pattern) matches one or more time the pattern.

@(pattern) matches one time the pattern.

!(pattern) matches string without the pattern.

So one can question a string in a variable like: if [[$var = fo@(?4*67).c]];then ...

15.2 ALIASES

The command aliases feature is supported in Korn shell also which allows us to abbreviate long command lines or rename commands. We do this by creating aliases for long command lines that we frequently use.

For example, assume that we often need to move to the directory /usr/chang/reports/status. We can create an alias status, which will move us to that directory whenever we enter it on the command line.

In addition, aliases allow us to make up more descriptive names for commands. For example, we could define an alias named rename for the mv command.

To create aliases, use the alias command. The general format of the alias command is the following :

alias *aliasname=command*

This is same as Bourn shell. The *aliasname* entry specifies the name you want to use. The *command* entry specifies either the original command or a series of commands. If the *command* has more than one part (has spaces), enclose the whole expression in single quotes.

For example, to create the alias status that moves us to the directory /usr/chang/reports/status, enter the following command :

alias status='cd /usr/chang/reports/status'

The usual way to define aliases is to place them in our .kshrc file so that we can use them whenever you log in or start a new shell.

To display all alias definitions, enter the following command :

$ alias

To display the definition of a particular alias, enter the following command :

$ alias *aliasname*

The *aliasname* entry specifies the particular alias for which we are requesting a definition.

The Korn shell allows us to export the aliases which we create. Variables that are exported are passed to any sub-shell's that are created so that when we execute a shell procedure or new shell, the alias remains defined. (Variables that are not exported are used only by the login shell.)

To export an alias, use the following form of the alias command :

alias -x *aliasname=command*

The -x flag specifies that we want to export the alias. The *aliasname* entry specifies the name you want to use. The *command* entry specifies either the original command or a series of commands. If the *command* has more than one part, enclose the whole expression in single quotes.

For example, to export an alias definition for the rm command, enter the following :

alias -x rm='rm -i '

We can enter the preceding command in one of two ways :

- Edit the .kshrc or .profile file if we want an alias exported whenever we log in.
- Export an alias on the command line if we want the alias exported only for the current login session.

To remove an alias for the current login session, use the unalias command. The general format of the unalias command is the following :

unalias *aliasname*

The *aliasname* entry specifies the alias which we want to remove.

To remove an alias for the current and all future login sessions, do the following :

1. Enter the following command :

   $ unalias *aliasname*

   The *aliasname* entry specifies the alias which we want to remove.

2. Edit the .kshrc file (or the file on our system that contains alias definitions) and remove the alias definition. Then, save the file.
3. Enter the following command to reexecute the .kshrc file :

$ source .kshrc

15.3 HISTORY

Like Bourn shell, Korn shell also supports history. History buffering characteristics can be controlled by setting many parameters. The following table shows the history related parameter.

Table 15.2 : Reexecuting History Buffer Commands

Command	Description
r	Reexecutes the previous command
r *n*	Reexecutes the command specified by *n*. For example, using the history buffer shown in the previous display, r 5 reexecutes the cd /usr/sales command.
r -*n*	Reexecutes a previous command relative to the current command. For example, using the history buffer shown in the previous display, r-2 invokes command number 16, cd /usr/chang.
r *string*	Reexecutes the most recent command that has first characters matching those specified by *string*. For example, using the history buffer shown in the previous display, r cp invokes command number 7, cp report report5.

If we want to increase or decrease the number of commands stored in our history buffer, set the *HISTSIZE* variable in our .profile file. This variable has the following format :

HISTSIZE=*n*

The *n* entry specifies the number of command lines we want to store in the history buffer.

For example, to store 15 commands in the history buffer, use the following command :

HISTSIZE=15

The Korn shell also allows us to edit current command lines as well as reuse those already entered in the command history buffer. To use this feature, we must know how to use a text editor such as vi or emacs. For information on these features, see the following section.

The command line editing functions for the Korn shell are extensive. This section covers only the most basic functions. For more detailed information, see the ksh reference page.

To display the command history buffer and/or to edit its contents, use the built-in command fc (fix command). The fc command has the following two formats :

1. fc [-e *editor*] [-nlr] [*first*] [*last*]

This command format allows us to display and/or edit any number of command lines in our buffer.

- The -e *editor* entry specifies the editor (usually vi or emacs) you want to use in editing the command line. If we do not specify -e, the fc command displays the lines, but does not allow you to edit them.
- The -n flag specifies that you want to list the command lines in the buffer without numbers. The -l flag specifies that you want to list the command lines in the buffer with numbers. If we do not specify a line number or a range of line numbers, the last 16 lines we entered will be listed.
- The -r flag specifies that we want to list the command in the buffer in reverse order.
- The *first* and *last* entries specify a range of command lines in the buffer. We may specify them either with numbers or with strings.

If we want to specify a default editor for the -e flag, define the *FCEDIT* variable in your .profile script. For example, if we want to make emacs our default editor, enter the following variable definition :

FCEDIT=emacs

2. fc -e - [*old=new*] [*string*]

This command allows us to immediately replace an *old* string with a *new* string within any previous command line.

- The -e - entry specifies that we want to make a replacement.
- The *old=new* entry specifies that we want to replace the *old* string with the *new* string.
- The *string* entry specifies that the Korn shell should make the edit to the most recent command line in the buffer containing the *string*.

Example

To display command lines 15 to 18, enter the following command :

```
$   fc -l 15 19
15 ls –la
16 cd /tmp
17 pwd
18 cd /u/ben/reports
19 more sales
$
```

We may also list the same command lines by specifying command strings instead of line numbers, as in the following example :

```
$   fc -l ls more
15 ls -la
16 pwd
17 cd /u/ben/reports
18 more sales
$
```

Example

To display and edit command lines 15 to 18 with the vi editor, enter the following command :

```
$  fc -e vi 15 18
ls -la
pwd
cd /u/ben/reports
more sales
~
~
~
~
```

After making our edits, write and exit the file with the :wq! command. The command lines in the file are then re-executed.

Example

Assume that we have just entered the echo hello command, and now want to replace hello with goodbye. To do the replacement and re-execute the command line, enter the following command :

```
$  echo hello
hello
$  fc -e - hello=goodbye echo
echo goodbye
goodbye
```

15.4 FILENAME COMPLETION

The Korn shell allows us to enter a portion of a filename or pathname at the shell prompt and the shell will automatically match and complete the name. If there is more than one filename or pathname that matches the criterion, the shell will provide us with a list of possible matches.

To activate the filename completion mechanism, define the *EDITOR* variable in our .profile file. For example, if we want to use the vi editor, enter the following variable definition in your .profile file :

```
EDITOR=vi
```

To demonstrate how filename completion works, assume that our editor is vi and that we have the salesreport1, salesreport2, and salesreport3 files in our current directory. To display a long listing and to activate filename completion, enter the following command :

```
$  ls -l salesreport[Escape]=
1) salesreportfeb
2) salesreportjan
3) salesreportmar
$ ls -l salesreport
```

The system redisplays our command, and the cursor is now at the end of salesreport. If we want to choose salesreportjan, type a (the vi append command) followed by jan, then press Return. The listing for salesreportjan will be displayed.

For more detailed information on filename completion, see the ksh(1) reference page.

15.5 BUILT-IN VARIABLES

The Korn shell also provides variables that can be assigned values. The shell sets some of these variables, and we can set or reset all of them.

Table 15.3 describes selected Korn shell built-in variables that are of the most interest to general users.

Table 15.3 : Built-In Korn Shell Variables

Variable	Description
HOME	Specifies the name of your login directory. The cd command uses the value of *HOME* as its default value. In Korn shell procedures, use *HOME* to avoid having to use full pathnames - something that is especially helpful if the pathname of your login directory changes. *HOME* is set by the login command.
PATH	Specifies the directories through which your system should search to find and execute commands. The shell searches these directories in the order specified here. Usually, *PATH* is set in the .profile file.
CDPATH	Specifies the directories that the cd command will search to find the specified argument to cd. If the cd argument is null, or if it begins with a slash (/), dot (.), or dot dot (..), then *CDPATH* is ignored. Usually, *CDPATH* is set in your .profile file.
MAIL	The pathname of the file where your mail is deposited. *MAIL* is usually set in your .profile file.
MAILCHECK	Specifies in seconds how often the shell checks for mail (600 seconds is the default). If the value of this variable is set to 0, the shell checks for mail before displaying each prompt. *MAILCHECK* is usually set in your .profile file.
SHELL	Specifies your default shell. This variable should be set and exported by your .profile file.
PS1	Specifies the default Korn shell prompt, and its default value is the dollar sign ($). *PS1* is usually set in your .profile file. If *PS1* is not set, the shell uses the standard primary prompt string.
PS2	Specifies the secondary prompt string - the string that the shell displays when it requires more input after entering a command line. The standard secondary prompt string is a > symbol followed by a space. *PS2* is usually set in your .profile file. If *PS2* is not set, the shell uses the standard secondary prompt string.

HISTFILE	Specifies the pathname of the file that will be used to store the command history. This variable is usually set in your .profile file.
EDITOR	Specifies the default editor for command line editing at the shell prompt and for filename completion. This variable is usually set in your .profile file.
FCEDIT	Specifies the default editor for the fc command. This variable is usually set in your .profile file.
HISTSIZE	Specifies the number of previously entered commands that are accessible by this shell. The default is 128. This variable is usually set in your .kshrc file.

15.6 BUILT-IN COMMANDS

Table 15.4 describes selected Korn shell commands that are of the most interest to general users. For a complete list of Korn shell built-in commands, see the ksh reference page.

Table 15.4 : Built-In Korn Shell Commands

Command	Description
alias	Assigns and displays alias definitions.[Table Note 1]
cd	Allows you to change directories. If no directory is specified, the value of the *HOME* shell variable is used. The *CDPATH* shell variable defines the search path for this command.[Table Note 2]
echo	Writes arguments to the standard output. For more information and the command format, see the ksh(1) reference page.
export	Marks the specified variable for automatic export to the environments of subsequently executed commands.[Table Note 3]
fc	Allows you to display, edit, and re-execute the contents of the command history buffer.[Table Note 4]
history	Displays the contents of the command history buffer.[Table Note 1]
jobs	Displays the job number and the PID number of current background processes.
pwd	Displays the current directory.[Table Note 5]
set	Assigns and displays variable values.[Table Note 6]
times	Displays the accumulated user and system times for processes run from the shell.
trap	Runs a specified command when the shell receives a specified signal.[Table Note 6]
umask	Specifies the permissions to be subtracted from the default permissions set by the creating program for all new files created.[Table Note 7]
unalias	Removes alias definitions.[Table Note 1]
unset	Removes values that have been assigned to variables.[Table Note 6]

Korn shell allows us to access command line arguments more than 9. For example ${10}, ${11}, etc., refers to 10^{th} and 11^{th} command line arguments given to a shell script.

15.7 A NOTE ON KORN SHELL PROGRAMMING CONSTRUCTS

All the programming constructs which are available in Bourn Again shell are available with Korn shell. Thus, we don't want reproduce the examples. We shall give some examples which are specific to Korn shell only.

if then fi:

Example

The following example is specific to Korn shell. Not available in Bourn shell. See the expression with [[ ]].

```
if [[ $value -eq 7 ]];then
   print "$value is 7"
fi
or:
if [[ $value -eq 7 ]]
then
   print "$value is 7"
fi
or:
if [[ $value -eq 7 ]];then print "$value is 7";fi
```

if then else fi:

This explains uses of if then else if in Korn shell.

Example

```
if [[ $name = "John" ]];then
   print "Your welcome, ${name}."
else
   print "Good bye, ${name}!"
fi
```

if then elif then else fi:

Example

The following example explains the use of if then elif then else if in Korn shell.

```
if [[ $name = "John" ]];then
   print "Your welcome, ${name}."
elif [[ $name = "Hanna" ]];then
   print "Hello, ${name}, who are you?"
else
   print "Good bye, ${name}!"
fi
```

To compare strings one uses "=" for equal and "!=" for not equal.

To compare numbers one uses "-eq" for equal "-ne" for not equal as well as "-gt" for greater than and "-lt" for less than.

Example

```
if [[ $name = "John" ]];then
   print "Same"
fi
if [[ $size -eq 1000 ]];then
   print 'Same"
fi
```

In Korn shell also, with "&&" for "AND" and "||" for "OR" one can combine statements:

Example

```
if [[ $price -lt 1000 || $name = "Hanna" ]];then
  # commands....
fi
if [[ $name = "Fred" && $city = "Denver" ]];then
  # commands
```

case

This is same as Bourn Shell

```
case $var in
   john|fred) print $invitation;;
   martin) print $declination;;
   *) print "Wrong name...";;
esac
```

While loop

This loop is also works in the same fashion as that of Bourn shell"s while loop.

```
let x=5
while (( x -=1 ))
do
echo x
done
```

Example

```
keeplooping=1;
while [[ $keeplooping -eq 1 ]] ; do
        read quitnow
        if [[ "$quitnow" = "yes" ]] ; then
                keeplooping=0
        fi
        if [[ "$quitnow" = "q" ]] ; then
                break;
        fi
done
```

Example

The following example explains the use of break

```
while read line;do
    if [[ $line = *!(.c) ]];then
        break
    else
        print $line
    fi
done
```

Example

The following example explains the use of continue

```
while read line
do
   if [[ $line = *.gz ]];then
      continue
   else
      print $line
   fi
done
while [[ $count -gt 0 ]];do
   print "\$count is $count"
   (( count -= 1 ))
done
```

until loop

The following example explains the use of until loop in Korn shell.

```
until [[ $answer = "yes" ]];do
   print -n "Please enter \"yes\": "
   read answer
   print ""
done
```

for loop

The for loop syntax is same as for loop of Bourn shell. It is given as :

for var in list do done

Example

The following example explains the use of for loop.

```
for foo in $(ls);do
  if [[ -d $foo ]];then
  print "$foo is a directory"
  else
  print "$foo is not a directory"
  fi
done
```

Example

This prints all the files of C.W.D. As nothing is given, for considers files of C.W.D as the list.

```
for arg;do
   print $arg
done
```

Example

The following example reads a line and prints the same.

To get each line of a file into a variable iteratively do :

```
{ while read myline;do
   # process $myline
done } < filename
```

Example

To catch the output of a pipeline each line at a time in a variable use :

```
last | sort | {
while read myline;do
   # commands
done }
```

Until Loop

```
Korn shell supports until loops also.
```

Example

The following example explains the use of until loop in Korn shell.

```
until [[ $stopnow -eq 1 ]] ; do
   echo just run this once
   stopnow=1;
   echo we should not be here again.
done
```

Functions

A function (= procedure) must be defined before it is called, because ksh is interpreted at run time. It knows all the variables from the calling shell except the command line arguments. But has it's own command line arguments so that one can call it with different values from different places in the script. It has an exit status but cannot return a value like a c function can.

Creating a Function

One can make one in either of the following two ways :

```
function foo {
   # commands...
}
foo(){
   # commands...
}
```

Calling the Function

To call it just put it's name in the script: foo. To give it arguments do: foo arg1 arg2 ...

The arguments are there in the form of $1...$n and $* for all at once like in the main code.

And the main $1 is not influenced bye the $1 of a particular function.

Return

The return statement exits the function immediately with the specified return value as an exit status.

The following function calculates the square of the first command line argument.

```
function sqr()
{
(( s = $1*$1 ))
return s
}
```

We can use the above function like :

```
sqr 2
eho $?
4
```

The following version also does the same.

```
function sqr1()
{
s = `expr $1 \* $1 `
return s
}
```

15.8 CONCLUSIONS

In this chapter, we have summarized the important aspects of Korn shell in relation to bourn shell.

Some example Korn shell programs are included.