CHAPTER 5
THE KORN SHELL

ALIASES

The Korn shell allows you to create and customize your ow ommands by using the **alias** command.

alias [*word* **[=***string***]]**

```
$ alias dir='ls -F' # option-F appends / for files, * for executables
$ dir # same as writing ls -F
a.c* b.d d1/ d2/
$ dir *.c
a.c*
$ alias dir
dir=ls -F
$ unalias dir # remove the alias
$ alias dir
dir alias not found
```

ARITHMETIC

let *expression*

The *expression* in the **let** command may contain the following operators

!	:	logical negation
*** / %**	:	multiplication, division, remainder
+ -	:	addition, subtraction
<= >= < >	:	relational operations
== !=	:	equality, inequality

*base***#***number* : *number* represented in given *base* which can be between 2 and 36

In the expressions do not use any space around operators and do not use $ in front of variables.

```
$ let x = 2 + 2 #no space or tabs allowed
ksh: syntax error
$ let x=2+2
$ echo $x
4
$ let y=x+4
```

```
$ echo $y
8
$ letx=2#100+2#100
$ echo $x
8
$
```

In order to prevent metacharacter interpretation in let command use

((*list***))**

which is equivalent to

let "*list*"

```
$ ((x=4))$ ((y=x*4))
$ echo $y
16
$
```

If an expression in the let command evaluates to zero, its return code is 1, otherwise it is 0.The return code may be used by the decision making control structures such as an if statement.

```
$ ((x=4))
$ if ((x>0))
> then
> echo x is positive
> fi
x is positive
$
```

FUNCTIONS

The Korn shell allows you to define functions that may be invoked as shell commands. The syntax is as follows:

function *name*
{
list of commands
}

or equivalently

```
function ( )
{
list of commands
}
```

```
$ cat func1.ksh
#!/bin/ksh
message ( )
{
hi there
}
i=
while ((i<=3))
do
   message
   let i=i+1
done
$ func1.ksh
hi there
hi there
hi there
$
```

The parameters passed to a function are accessibl via the standard positional parameter mechanism.

```
$ cat func2.ksh
#!/bin/ksh
f ( )
{
echo parameter 1=$1
echo parameter list=$*
}
# main program
f 1 # call f with one parameter
f cat dog # call f with two parameters
f $3 $2 # call f with the third and cond arguments of the script
$ func2.ksh 2 3 4
parameter 1= 1
parameter list= 1
parameter 1=cat
parameter list= cat dog
parameter 1=4
parameter list= 4 3
```

The return command returns the flow of control back to the caller and has the following syntax:

return [*value***]**

When **return** is used without an agument, the funtion call returns with the exit code of the last command that was executed in the function; otherwise it returns with its exit value set to value. The exit code is accessible from the caller via the **$?** variable.

```
$ cat func3.ksh
#!/bin/ksh
f ( ) # two parameter function
{
((returnValue=$1*$2))
return $returnValue
}
# main
f 3 4  # call function
result=$?  #save exit code
echo return value from function was $result
$func3.ksh
return value from function was 12
$
```

A function executes in the same context as the process that calls it. This means that it shares the same variables and current working directory.

```
$ cat func4.ksh
#!/bin/ksh
f ( )
{
x=5
}
# main
x=1
echo $x
f
echo $x
$ func4.ksh
1
5
$
```

The **typset** command allows the creation and manipuation of local variables. Specifically a variable created using the **typset** command is limited in scope tothe function in which it i created and all of the function that the defining function calls.If a variable of the same name already exist, its original value is preserved when the function returns.

```
$ cat func5.ksh
#!/bin/ksh
f ( ) # two parameter function
{
typeset x # declare local variable
((x=$1*$2))
```

```
echo local x=$x
return $x
}
# main
x=1
echo global x=$x
f 3 4  # call function
result=$?  #save exit code
echo return value from function was $result
echo globl x=$x
$func5.ksh
global x=1
local x=12
return value from function was 12
global x=1
$
```

RECURSION

Recursive factorial using exit code:

```
$ cat fact1.ksh
#!/bin/ksh
factorial ( ) #one parameter function
{
if (($1<=1))
then
   return 1
else
   typeset tmp
   typeset result
   ((tmp=$1-1))
   factorial $tmp # recursive call to the function itself
   (( result= $?*$1))
   return $result
fi
}
main
factorial $1
echo factorial $1 = $?
$ fact1.ksh 5
factorial 5 = 120
```

Recursive factorial using standard output:

```
$ cat fact2.ksh
#!/bin/ksh
factorial ( ) #one parameter function
{
if (($1<=1))
```

```
then
   echo 1
else
   typeset tmp
   typeset result
   ((tmp=$1-1))
   ((result= `factorial $tmp`*$1)) # recursive call
   echo $result
fi
}
# main
echo factorial $1 = `factorial $1`
$ fact2.ksh 5
factorial 5 = 120
$
```

SHARING FUNCTIONS

In order to share the source code of a function between several scripts, place it in a separate file and th read it using the "**.**" built in command at the start of the script that uses the function. Suppose that the source code of the function factorial was saved in a file called fact.ksh

```
$cat fact3.ksh
#!/bin/ksh
.fact.ksh
# main
echo factorial $1 = `factorial $1`
$ fact3.ksh 5
factorial 5 = 120
$
```

ARRAYS

The Korn shell supports simple one dimensional arrays. To create an array, simply assign a value to variable *name* using an *index* between 0 and 511 in brackets.

name`[`*index*`]=`*value*

Array elements are created when needed. To acces an array elment use the following syntax:

`${` *name*`[`*index*`]}`

If the *index* is omitted, it is 0 by default.

```
$ cat squares.ksh
i=0
while ((i<=5))
do
   ((squares[$i]=i*i))
   ((i=i+1))
done
echo 3 squared is ${squares[3]}
echo list of all squares is ${squares[*]}
$ squares.ksh
3 squared is 9
list of all squares is 0 1 4 9 16 25
```