Operating Systems: Configuration & Use
CIS345

PROGRAMMING THE BOURNE AGAIN SHELL

Mostafa Z. Ali

mzali@just.edu.jo
Introduction

• This chapter introduces additional Bourne Again Shell commands, builtins, and concepts that carry shell programming to a point where it can be useful

• The first part of this chapter covers programming control structures, which are also known as control flow constructs

• The next part of this chapter discusses parameters and variables, going into detail about array variables, local versus global variables, special parameters, and positional parameters

Tip: Do not name a shell script test

You can unwittingly create a problem if you give a shell script the name `test` because a Linux utility has the same name. Depending on how the `PATH` variable is set up and how you call the program, you may run your script or the utility, leading to confusing results.
Control Structures

• The *control flow commands* alter the order of execution of commands within a shell script
• Control structures include the *if...then, for...in, while, until,* and *case* statements
• The *break* and *continue* statements work in conjunction with the control structures to alter the order of execution of commands within a script
if...then

if test-command
  then
    commands
  fi

$ cat if1
  echo -n "word 1:"
  read word1
  echo -n "word 2:"
  read word2
  if test "$word1" = "$word2"
    then
      echo "Match"
  fi
  echo "End of program."
$ if1
  word 1: peach
  word 2: peach
  Match
  End of program.

uses an if structure to execute commands based on the result returned by the test builtin when it compares the two words

In this example the test-command is test "\$word1" = "\$word2". The test builtin returns a true status if its first and third arguments have the relationship specified by its second argument. If this command returns a true status (= 0), the shell executes the commands between the then and fi statements
• In the Bourne Again Shell, test is a builtin—part of the shell. It is also a stand-alone utility kept in /usr/bin/test
• You usually use the builtin version if it is available and the utility if it is not

```
$ cat chkargs
if test $# -eq 0
then
  echo "You must supply at least one argument."
  exit 1
fi
echo "Program running."
$ chkargs
You must supply at least one argument.
$ chkargs abc
Program running.
```

The if structure at the beginning of a script checks that you have supplied at least one argument on the command line.
The -eq test operator compares two integers, where the $# special parameter takes on the value of the number of command line arguments.

A test like the one shown in chkargs is a key component of any script that requires arguments.
Sometimes the script simply tests whether arguments exist (as in `chkargs`), while other scripts test for a specific number or specific kinds of arguments.

```bash
$ cat is_ordfile
if test $# -eq 0
    then
        echo "You must supply at least one argument."
        exit 1
fi
if test -f "$1"
    then
        echo "$1 is an ordinary file in the working directory"
    else
        echo "$1 is NOT an ordinary file in the working directory"
fi
```

After verifying that at least one argument has been given on the command line, the following script tests whether the argument is the name of an ordinary file (not a directory or other type of file) in the working directory.

**Table 11-1** Options to the `test` builtin

<table>
<thead>
<tr>
<th>Option</th>
<th>Tests file to see if it</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d</td>
<td>Exists and is a directory file</td>
</tr>
<tr>
<td>-e</td>
<td>Exists</td>
</tr>
<tr>
<td>-f</td>
<td>Exists and is an ordinary file (not a directory)</td>
</tr>
<tr>
<td>-r</td>
<td>Exists and is readable</td>
</tr>
<tr>
<td>-s</td>
<td>Exists and has a size greater than 0 bytes</td>
</tr>
<tr>
<td>-w</td>
<td>Exists and is writable</td>
</tr>
<tr>
<td>-x</td>
<td>Exists and is executable</td>
</tr>
</tbody>
</table>
It is a good practice to test arguments in shell programs that other people will use.

```bash
$ cat chkargs2
if [ $# -eq 0 ]
    then
        echo "Usage: chkargs2 argument..." 1>&2
        exit 1
fi
echo "Program running."
exit 0
$ chkargs2
Usage: chkargs2 arguments
$ chkargs2 abc
Program running.
```

The error message that **chkargs2** displays is called a usage message and uses the **1>&2** notation to redirect its output to standard error.

The example uses the bracket ([[]]) synonym for test.
The brackets must be surrounded by whitespace (SPACESs or TABs).
• The usage message is commonly employed to specify the type and number of arguments the script takes

```bash
$ cp
cp: missing file argument
Try 'cp --help' for more information.
```

Many Linux utilities provide usage messages similar to the one in `chkargs2`
if...then...else

\begin{verbatim}
if test-command
  then
    commands
  else
    commands
fi

if test-command; then
  commands
else
  commands
fi
\end{verbatim}

Because a semicolon (;) ends a command just as a NEWLINE does, you can place `then` on the same line as `if` by preceding it with a semicolon.
arguments that are filenames, it displays the files on the terminal. If the first argument is \texttt{-v} (called an option in this case), \texttt{out} uses \texttt{less} to display the files one page at a time.

\texttt{out} uses the \texttt{shift} builtin to shift the arguments to get rid of the \texttt{-v} and displays the files using \texttt{less}.

If the result of the test is \texttt{false} (if the first argument is not \texttt{-v}), the script uses \texttt{cat} to display the files.

\textbf{optional} In \texttt{out} the \texttt{--} argument to \texttt{cat} and \texttt{less} tells these utilities that no more options follow on the command line and not to consider leading hyphens (\texttt{-}) in the following list as indicating options. Thus \texttt{--} allows you to view a file with a name that starts with a hyphen. Although not common, filenames beginning with a hyphen do occasionally occur. (You can create such a file by using the command \texttt{cat > --fname}.) The \texttt{--} argument works with all Linux utilities that use the \texttt{getopts} builtin (page 454) to parse their options; it does not work with \texttt{more} and a few other utilities. This argument is particularly useful when used in conjunction with \texttt{rm} to remove a file whose name starts with a hyphen (\texttt{rm -- --fname}), including any that you create while experimenting with the \texttt{--} argument.
The *elif* statement combines the *else* statement and the *if* statement and allows you to construct a nested set of *if...then...else* structures.
The first if statement uses the Boolean operator AND (−a) as an argument to test

The double quotation marks around the arguments to echo that contain ampersands (&) prevent the shell from interpreting the ampersands as special characters.
Debugging Shell Scripts

• You can use the shell’s `-x` option to help debug a script. This option causes the shell to display each command before it runs the command.

```bash
$ bash -x lnks letter /home
+ ['2 -eq 0 -o 2 -gt 2 ']
+ [' -d letter ']
+ file=letter
+ ['2 -eq 1 ']
+ [' -d /home ']
+ directory=/home
+ ['! -f letter ']
...
```

Each command that the script executes is preceded by the value of the `PS4` variable—a plus sign (+) by default, so you can distinguish debugging output from script-produced output.

You must export `PS4` if you set it in the shell that calls the script.

```bash
$ export PS4='>>>> ' This command sets PS4 to >>>> followed by a SPACE and exports it
```
• You can also set the \(-x\) option of the shell running the script by putting the following set command at the top of the script

\texttt{set \(-x\)}

• Turn the debugging option off with a plus sign

\texttt{set +x}

• The \texttt{set \(-o\ xtrace\)} and \texttt{set +o\ xtrace\} commands do the same things as \texttt{set \(-x\)} and \texttt{set +x}, respectively
for...in

for loop-index in argument-list
  do
    commands
  done

done

$ cat fruit
for fruit in apples oranges pears bananas
do
    echo "$fruit"
done
echo "Task complete."

$ cat dirfiles
for i in *
do
    if [ -d "$i" ]
        then
            echo "$i"
    fi
done

This script lists the names of the directory files in the working directory by looping over all the files, using test to determine which files are directories.
for

• It is the same as the for...in structure except for where it gets values for the loop-index
• The for structure performs a sequence of commands, usually involving each argument in turn

```
$ cat for_test
for arg
do
    echo "$arg"
done
$ for_test candy gum chocolate
candy
gum
chocolate
```
The first line of the script, for arg, implies for arg in "$@", where the shell expands "$@" into a list of quoted command line arguments "$1" "$2" "$3" and so on.
while

```bash
while test-command
do
  commands
done
```

- The script uses `test` with the `–lt` argument to perform a numerical test.
- For numerical comparisons, you must use `–ne` (not equal), `–eq` (equal), `–gt` (greater than), `–ge` (greater than or equal to), `–lt` (less than), or `–le` (less than or equal to).
- For string comparisons use `=` (equal) or `!=` (not equal).
$ cat count
#!/bin/bash
number=0
while [ "$number" -lt 10 ]
  do
    echo -n "$number"
    ((number +=1))
  done
echo
$ count
0123456789
$
until

until test-command
do
  commands
done

$ cat until
secretname=jenny
name=noname
echo "Try to guess the secret name!"
echo
until [ "$name" = "$secretname" ]
do
  echo -n "Your guess: "
  read name
done
echo "Very good."

$ until
Try to guess the secret name!

Your guess: helen
Your guess: barbara
Your guess: rachael
Your guess: jenny
Very good
The `locktty` script is similar to the `lock` command on Berkeley UNIX and the Lock Screen menu selection in GNOME. The script prompts you for a key (password) and uses an `until` control structure to lock the terminal.

- The `until` statement causes the system to ignore any characters typed at the keyboard until the user types in the key on a line by itself, which unlocks the terminal.

To clear the screen:

```bash
$tput clear
```

The `stty -echo` command causes the terminal not to display characters typed at the keyboard, thereby preventing the key that the user enters from appearing on the screen.

```bash
stty echo
```
break and continue

You can interrupt a **for**, **while**, or **until** loop by using a **break** or **continue** statement.

The **break** statement transfers control to the statement after the **done** statement which terminates execution of the loop. The **continue** command transfers control to the **done** statement, which continues execution of the loop.

```
$ cat brk
for index in 1 2 3 4 5 6 7 8 9 10
do
  if [ $index -le 3 ]; then
    echo "continue"
    continue
  fi
  # echo $index
  if [ $index -ge 8 ]; then
    echo "break"
    break
  fi
done

$ brk
continue
continue
continue
4
5
6
7
8
break
```
case

```bash
$ cat casel
echo -n "Enter A, B, or C: ">
read letter
case "$letter" in
  A)
    echo "You entered A"
    ;;
  B)
    echo "You entered B"
    ;;
  C)
    echo "You entered C"
    ;;
  *)
    echo "You did not enter A, B, or C"
    ;;
esac
```

The right parenthesis is part of the case control structure, not part of the pattern.

$ casel
Enter A, B, or C: B
You entered B
```bash
$ cat case2
echo -n "Enter A, B, or C: "
read letter
case "$letter" in
  a|A)
    echo "You entered A"
    ;;
  b|B)
    echo "You entered B"
    ;;
  c|C)
    echo "You entered C"
    ;;
  *)
    echo "You did not enter A, B, or C"
    ;;
esac

$ case2
Enter A, B, or C: b
You entered B
```
• The **pattern** in the **case** structure is analogous to an ambiguous file reference. It can include any of the special characters and strings shown in the Table below.

<table>
<thead>
<tr>
<th>Table 11-2 Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pattern</strong></td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>[...]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
select

- The **select** control structure displays a menu, assigns a value to a variable based on the user’s choice of items, and executes a series of commands

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

- If you omit the keyword **in** and the list of arguments, **select** uses the positional parameters in place of the **arg**

- The menu is formatted with numbers before each item

```
select fruit in apple banana blueberry kiwi orange watermelon STOP
```

displays the following menu:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>apple</td>
<td>3</td>
<td>blueberry</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>banana</td>
<td>4</td>
<td>kiwi</td>
<td>6</td>
</tr>
</tbody>
</table>
• The **select** structure uses the values of the **LINES** and **COLUMNS** variables to determine the size of the display. (**LINES** has a default value of 24; **COLUMNS** has a default value of 80.) With **COLUMNS** set to 20, the menu looks like this:

1) apple  
2) banana  
3) blueberry  
4) kiwi  
5) orange  
6) watermelon  
7) STOP

• After displaying the menu **select** displays the value of **PS3**, the special **select** prompt
• The default value of **PS3** is ?#
• If you make an invalid entry, `varname` is set to null
• If you press RETURN without entering a choice, the shell redisplay the menu and the **PS3** prompt
• The `select` structure continues to issue the **PS3** prompt and execute the commands until something causes it to exit—typically a `break` or `exit` statement
• A `break` statement exits from the loop and an `exit` statement exits from the script
$ cat fruit2
#!/bin/bash
PS3="Choose your favorite fruit from these possibilities: "
select FRUIT in apple banana blueberry kiwi orange watermelon STOP
  do
    if [ "$FRUIT" == "" ]; then
      echo -e "Invalid entry.\n"
      continue
    elif [ $FRUIT = STOP ]; then
      echo "Thanks for playing!"
      break
    fi
  echo "You chose $FRUIT as your favorite."
  echo -e "That is choice number $REPLY\n"
done
$ fruit2
1) apple   3) blueberry   5) orange   7) STOP
2) banana  4) kiwi       6) watermelon
Choose your favorite fruit from these possibilities: 3
You chose blueberry as your favorite.
That is choice number 3.

Choose your favorite fruit from these possibilities: 99
Invalid entry.

Choose your favorite fruit from these possibilities: 7
Thanks for playing!
Here Document

• A Here document allows you to redirect input to a shell script from within the shell script itself

• A Here document is so called because it is *here—immediately accessible* in the shell script

```bash
$ cat birthday
grep -i "$1" <<+
Alex       June 22
Barbara    February 3
Darlene    May 8
Helen      March 13
Jenny      January 23
Nancy      June 26
+
$ birthday Jenny
Jenny      January 23
$ birthday june
Alex       June 22
Nancy      June 26
```

• The two less than (<<) symbols in the first line indicate that a Here document follows
• One or more characters that delimit the Here document follow the less than symbols—this example uses a plus sign
• The opening delimiter must appear adjacent to the less than symbols, the closing delimiter must be on a line by itself

• In the example it is as though you had redirected standard input to `grep` from a file, except that the file is embedded in the shell script
File Descriptors

• before a process can read from or write to a file it must open that file
• When a process opens a file, Linux associates a number (called a file descriptor) with the file
• Each process has its own set of open files and its own file descriptors
• A typical Linux process starts with three open files: standard input (file descriptor 0), standard output (file descriptor 1), and standard error (file descriptor 2)
• Recall that you redirect standard output with the symbol > or the symbol 1> and that you redirect standard error with the symbol 2>
• The Bourne Again Shell opens files using the `exec` builtin as follows:

```
exec n> outfile
exec m< infile
```

• The first line opens `outfile` for output and holds it open, associating it with file descriptor `n`
• The second line opens `infile` for input and holds it open, associating it with file descriptor `m`
• The `<&` token duplicates an input file descriptor; use `>&` to duplicate an output file descriptor
• Once you have opened a file, you can use it for input and output in two different ways:
  – First, you can use I/O redirection on any command line, redirecting standard output to a file descriptor with `>&n` or redirecting standard input from a file descriptor with `<&n`
  – Second, you can use the `read` and `echo` builtins
• When you have finished using a file, you can close it with:

  `exec n<&-`

---

**A function is not a shell script**

**tip** The `mycp` example is a shell function; it will not work as you expect if you execute it as a shell script. (It will work: The function will be created in a very short-lived subshell, which is probably of little use.) You can enter this function from the keyboard. If you put the function in a file, you can run it as an argument to the `. (dot) builtin` (page 279). You can also put the function in a startup file if you want it to be always available (page 333).
function mycp ()
{
    case $# in
        0)
            # zero arguments
            # file descriptor 3 duplicates standard input
            # file descriptor 4 duplicates standard output
            exec 3<&0 4<&1
            ;;
        1)
            # one argument
            # open the file named by the argument for input
            # and associate it with file descriptor 3
            # file descriptor 4 duplicates standard output
            exec 3< $1 4<&1
            ;;
        2)
            # two arguments
            # open the file named by the first argument for input
            # and associate it with file descriptor 3
            # open the file named by the second argument for output
            # and associate it with file descriptor 4
            exec 3< $1 4> $2
            ;;
    esac
}

- If you supply only one argument, the script copies the file named by the argument to standard output
- If you invoke `mycp` with no arguments, it copies standard input to standard output
The real work of this function is done in the line that begins with `cat`. The rest of the script arranges for file descriptors 3 and 4, which are the input and output of the `cat` command, to be associated with the appropriate files.
Parameters and Variables

Array Variables

• The Bourne Again Shell supports one-dimensional array variables

• The subscripts are integers with zero-based indexing (i.e., the first element of the array has the subscript 0)

```
name=(element1 element2 ...)
```

```
$ NAMES=(max helen sam zach)
assigns four values to the array NAMES

$ echo ${NAMES[2]}
reference a single element of an array
sam
```

• The subscripts [*] and [@] both extract the entire array but work differently when used within double quotation marks
• An `@` produces an array that is a duplicate of the original array
• An `*` produces a single element of an array (or a plain variable) that holds all the elements of the array separated by the first character in `IFS` (normally a SPACE)
• The `declare` builtin with the `–a` option displays the values of the arrays (and reminds you that bash uses zero-based indexing for arrays):

```
$ A="${NAMES[@]}"
$ B="${NAMES[@]}"

$ declare -a
declare -a A='([0]="max helen sam zach")'
...
```

-> you can see that `NAMES` and `B` have multiple elements
-> In contrast, `A`, which was assigned its value with an `*` within double quotation marks, has only one element
echo attempts to display element 1 of array A

You can apply the ${#name[*]} operator to array variables, returning the number of elements in the array

The same operator, when given the index of an element of an array in place of *, returns the length of the element
Locality of Variables

- By default variables are local to the process in which they are declared
- Under bash, export makes a variable available to child processes
- The shell places the value of the variable in the calling environment of child processes. This call by value gives each child process a copy of the variable for its own use

```bash
$ cat extest1
cheese=american
echo "extest1 1: $cheese"
subtest
echo "extest1 2: $cheese"
$ cat subtest
cheese=swiss
echo "subtest 1: $cheese"
$ extest1
extest1 1: american
subtest 1:
subtest 2: swiss
extest1 2: american
```

The export statement in this example is equivalent to the following two statements:
```bash
cheese=american
export cheese
```
Functions

• Because functions run in the same environment as the shell that calls them, variables are implicitly shared by a shell and a function it calls

```bash
$ function nam () {
  > echo $myname
  > myname=zach
  > }

$ myname=sam
$ nam
sam
$ echo $myname
zach
```

• You need to make sure that the names of the variables used within the function do not interact with variables of the same name in the programs that call the function. Local variables eliminate this problem.

• When used within a function, the `typeset builtin` declares a variable to be local to the function it is defined in.
A function being entered from the keyboard; it is not a shell script:

```bash
function count_down () {
    typeset count
    count=$1
    while [ $count -gt 0 ]
    do
        echo "$count..."
        ((count=count-1))
        sleep 1
        done
        echo "Blast Off."
    }

count=10
count_down 4
4...
3...
2...
1...
Blast Off!!
}
```

-> assignment is enclosed between double parentheses, which cause the shell to perform an arithmetic evaluation.

-> Within the double parentheses you can reference shell variables without the leading dollar sign ($).
Special Parameters

$$\textbf{\$\$: PID Number}$$

- Special parameters enable you to access useful values pertaining to command line arguments and the execution of shell commands.
- You reference a shell special parameter by preceding a special character with a dollar sign ($$)
- As with positional parameters, it is not possible to modify the value of a special parameter by assignment.
- The shell stores in the $$ parameter the PID number of the process that is executing it.

```
$ echo $$
5209
$ ps
  PID TTY  TIME CMD
  5209 pts/1  00:00:00 bash
  6015 pts/1  00:00:00 ps
```

Because `echo` is built into the shell, the shell does not have to create another process when you give an `echo` command.
The shell creates a new shell process when it runs a shell script.

The `id2` script displays the PID number of the process running it (not the process that called it—the substitution for `$$` is performed by the shell that is forked to run `id2`).

`id2` displays its name (`$0`) and the PID of the subshell that it is running in.

The value of the PID number of the last process that you ran in the background is stored in `$!`.
$? : Exit Status

- When a process stops executing for any reason, it returns an *exit status to the parent process*
- The exit status is also referred to as a *condition code or a return code*.
- The $? variable stores the exit status of the last command.
- By convention a nonzero exit status represents a *false value and means that the* command failed.
- A zero is *true* and indicates that the command was successful.

```
$ ls es
es
$ echo $?
0
$ ls xxx
ls: xxx: No such file or directory
$ echo $? 1
```
You can specify the exit status that a shell script returns by using the `exit` builtin, followed by a number, to terminate the script.

```bash
$ cat es
echo This program returns an exit status of 7.
exit 7
$ es
This program returns an exit status of 7.
$ echo $?
7
$ echo $?
0
```
Positional Parameters

$#: Number of Command Line-Arguments

- The *positional parameters* comprise the command name and command line arguments
- They are called *positional* because within a shell script, you refer to them by their position on the command line
- The $# parameter holds the number of arguments on the command line

```bash
$ cat num_args
echo "This script was called with $# arguments."
$ num_args sam max zach
This script was called with 3 arguments.
```
$0: Name of the Calling Program

- You can use the base-name utility and command substitution to extract and display the simple filename of the command.

```bash
$ cat abc
echo "The command used to run this script is $0"
$ abc
The command used to run this script is ./abc
$ /home/sam/abc
The command used to run this script is /home/sam/abc
```

```bash
$ cat abc2
echo "The command used to run this script is $(basename $0)"
$ /home/sam/abc2
The command used to run this script is abc2
```
$1–$n: Command Line Arguments

- The first argument on the command line is represented by parameter $1$, the second argument by $2$, and so on up to $n$
- For values of $n$ over 9, the number must be enclosed within braces

```
$ cat display_5args
echo First 5 arguments are $1 $2 $3 $4 $5

$ display_5args jenny alex helen
First 5 arguments are jenny alex helen
```
**$* AND $@: Represent All Command-Line Arguments**

- The $* variable represents all the command line arguments

```bash
$ cat display_all
echo All arguments are $*

$ display_all a b c d e f g h i j k l m n o p
All arguments are a b c d e f g h i j k l m n o p
```

```bash
$ cat showargs
echo "$0 was called with $# arguments, the first is :$1:.

$ showargs a b c
./showargs was called with 3 arguments, the first is :a:.
$ echo $xx

$ showargs $xx a b c
./showargs was called with 3 arguments, the first is :a:.
$ showargs "$xx" a b c
./showargs was called with 4 arguments, the first is ::
```

-> The quotation marks are particularly important when you are using positional parameters as arguments to commands
-> Without double quotation marks, a positional parameter that is not set or that has a null value disappears
The `$*` and `$@` parameters work the same way except when they are enclosed within double quotation marks.

Using "$*" yields a single argument (with SPACEs or the value of `IFS` between the positional parameters), whereas "$@" produces a list wherein each positional parameter is a separate argument.

This difference typically makes "$@" more useful than "$*" in shell scripts.
**shift**: Promotes Command Line Arguments

- The `shift` builtin promotes each command line argument. The first argument (which was `$1`) is discarded.
- The second argument (which was `$2`) becomes the first argument (now `$1`), the third becomes the second, and so on.
- Because no “unshift” command exists, you cannot bring back arguments that have been discarded.
- An optional argument to `shift` specifies the number of positions to shift (and the number of arguments to discard); the default is 1.
Repeatedly using `shift` is a convenient way to loop over all the command line arguments in shell scripts that expect an arbitrary number of arguments.

```bash
$ cat demo_shift
echo "arg1= $1   arg2= $2   arg3= $3"
shift
echo "arg1= $1   arg2= $2   arg3= $3"
shift
echo "arg1= $1   arg2= $2   arg3= $3"
shift
echo "arg1= $1   arg2= $2   arg3= $3"

$ demo_shift alice helen jenny
arg1= alice   arg2= helen   arg3= jenny
arg1= helen   arg2= jenny   arg3= 
arg1= jenny   arg2= arg3= 
arg1= arg2= arg3= 
```
set: Initializes Command Line Arguments

- When you call the set builtin with one or more arguments, it assigns the values of the arguments to the positional parameters, starting with $1

```
$ cat set_it
set this is it
```

- Combining command substitution with the set builtin is a convenient way to get standard output of a command in a form that can be easily manipulated in a shell script

```
$ echo $3 $2 $1
$ set_it
it is this
```
```bash
$ date
$ cat dataset
set $(date)
echo $*
echo
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
echo "Argument 6: $6"
echo
echo "$2 $3, $6"

$ dataset

Argument 1: Wed
Argument 2: Jan
Argument 3: 2
Argument 6: 2008

Jan 2, 2008
```
Expanding Null and Unset Variables

• The expression `${name}` (or just $name if it is not ambiguous) expands to the value of the name variable

• If name is null or not set, bash expands `${name}` to a null string

• The Bourne Again Shell provides the following alternatives to accepting the expanded null string as the value of the variable:
  • Use a default value for the variable.
  • Use a default value and assign that value to the variable.
  • Display an error.

• You can choose one of these alternatives by using a modifier with the variable name. In addition, you can use `set –o nounset` to cause bash to display an error and exit from a script whenever an unset variable is referenced
Uses a Default Value

- The `:-` modifier uses a default value in place of a null or unset variable while allowing a nonnull variable to represent itself.

```bash
${name:-default}
```

```bash
$ ls ${LIT:-/home/alex/literature}
$ ls ${LIT:-$HOME/literature}
```

The default can itself have variable references that are expanded.
Assigns a Default Value

- The := modifier does not change the value of a variable. You may want to change the value of a null or unset variable to its default in a script, however.
- You can do so with the := modifier:
  ```
  ${name:=default}
  ```
- The shell expands the expression `${name:=default}` in the same manner as it expands `${name:=default}` but also sets the value of name to the expanded value of `default`.
- Shell scripts frequently start with the : (colon) builtin followed on the same line by the := expansion modifier to set any variables that may be null or unset.
- The : builtin evaluates each token in the remainder of the command line but does not execute any commands.
- Without the leading colon (:), the shell evaluates and attempts to execute the “command” that results from the evaluation.
• Use the following syntax to set a default for a null or unset variable in a shell script (there is a SPACE following the first colon):

```
: ${name:=default}
```

• When a script needs a directory for temporary files and uses the value of `TEMPDIR` for the name of this directory, the following line makes `TEMPDIR` default to `/tmp`:

```
: ${TEMPDIR:=/tmp}
```
?: Displays an Error Message

- If the variable is null or unset, the :? Modifier causes the script to display an error message and terminate with an exit status of 1:

  ${name:?message}

- If you omit message, the shell displays the default error message (parameter null or not set)

  cd ${TESTDIR:?$(date +%T) error, variable not set.}
Built-in Commands

**type**: Displays Information About a Command

- Commands that are built into a shell do not fork a new process when you execute them.
- The `type` builtin provides information about a command:

  ```
  $ type cat echo who if lt
  cat is hashed (/bin/cat)
  echo is a shell builtin
  who is /usr/bin/who
  if is a shell keyword
  lt is aliased to 'ls -ltrh | tail'
  ```

  The preceding output shows the files that would be executed if you gave `cat` or `who` as a command.

  ```
  $ cat read1_no_quote
  echo -n "Go ahead: "
  read firstline
  echo You entered: $firstline
  $ read1_no_quote
  Go ahead: *
  You entered: read1 read1_no_quote script.1
  $ ls
  read1  read1_no_quote  script.1
  ```

  Without writing double quotation, if you write something like “*” as input, then it will produce “*” as output, and does not get expanded.
read: Accepts User Input

```bash
$ cat read1
  echo -n "Go ahead: "
  read frontline
  echo "You entered: $firstline"
$ read1
  Go ahead: This is a line.
  You entered: This is a line.

$ cat read1_no_quote
  echo -n "Go ahead: "
  read frontline
  echo You entered: $firstline
$ read1_no_quote
  Go ahead: *
  You entered: read1 read1_no_quote script.1
$ ls
  read1  read1_no_quote  script.1
```
• When you do not specify a variable to receive read’s input, bash puts the input into the variable named **REPLY**

• You can use the **–p** option to prompt the user instead of using a separate echo command

```
$ cat read1a
read -p "Go ahead: "
echo "You entered: $REPLY"
```

```
$ cat read2
read -p "Enter a command: " cmd
cmd
echo "Thanks"
```

The **read2** script prompts for a command line and reads the user’s response into the variable **cmd**
The shell executes the command and then displays Thanks.

If cmd does not expand into a valid command line, the shell issues an error message.

The read builtin assigns one word (a sequence of nonblank characters) to each variable.
$ read3
Enter something: this is something else, really.
Word 1 is: this
Word 2 is: is
Word 3 is: something else, really.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a name (array)</td>
<td>Assigns each word of input to an element of array <code>aname</code>.</td>
</tr>
<tr>
<td>-d delim (delimiter)</td>
<td>Uses <code>delim</code> to terminate the input instead of NEWLINE.</td>
</tr>
<tr>
<td>-e (Readline)</td>
<td>If input is coming from a keyboard, use the Readline Library (page 322) to get input.</td>
</tr>
<tr>
<td>-n num (number of characters)</td>
<td>Reads <code>num</code> characters and returns. As soon as the user types <code>num</code> characters, <code>read</code> returns; there is no need to press RETURN.</td>
</tr>
<tr>
<td>-p prompt (prompt)</td>
<td>Displays <code>prompt</code> on standard error without a terminating NEWLINE before reading input. Displays <code>prompt</code> only when input comes from the keyboard.</td>
</tr>
<tr>
<td>-s (silent)</td>
<td>Does not echo characters.</td>
</tr>
<tr>
<td>-un (file descriptor)</td>
<td>Uses the integer <code>n</code> as the file descriptor that <code>read</code> takes its input from.</td>
</tr>
</tbody>
</table>

```
read -u4 arg1 arg2
```

is equivalent to
```
read arg1 arg2 <&4
```
See “File Descriptors” (page 429) for a discussion of redirection and file descriptors.
The placement of the redirection symbol (<) for the while structure is critical.

It is important that you place the redirection symbol at the done statement and not at the call to read.
Each time you redirect input, the shell opens the input file and repositions the read pointer at the start of the file:

```
$ read linel < names; echo $linel; read line2 < names; echo $line2
Alice Jones
Alice Jones
```

Here each `read` opens `names` and starts at the beginning of the `names` file. In the following example, `names` is opened once, as standard input of the subshell created by the parentheses. Each `read` then reads successive lines of standard input.

```
$ (read linel; echo $linel; read line2; echo $line2) < names
Alice Jones
Robert Smith
```

Another way to get the same effect is to open the input file with `exec` and hold it open (refer to “File Descriptors” on page 429):

```
$ exec 3< names
$ read -u3 linel; echo $linel; read -u3 line2; echo $line2
Alice Jones
Robert Smith
$ exec 3<&-
```
exec: Executes a Command

• The **exec** builtin has two primary purposes: to run a command without creating a new process and to redirect a file descriptor—including standard input, output, or error—of a shell script from within the script

• **exec** executes a command in place of (overlays) the current process

  *exec command arguments*

• because **exec** does not return control to the original program, it can be used only as the last command that you want to run in a script
The following script shows that control is not returned to the script:

```bash
$ cat exec_demo
who
exec date
echo "This line is never displayed."

$ exec_demo
jenny pts/7 May 30 7:05 (bravo.example.com)
hls pts/1 May 30 6:59 (:0.0)
Mon May 26 11:42:56 PDT 2007
```

- `exec < infile` causes all subsequent input to a script that would have come from standard input to come from the file named `infile`
- `exec > outfile 2> errfile` redirects standard output and standard error to `outfile` and `errfile`, respectively
- When you use `exec` in this manner, the current process is not replaced with a new process, and `exec` can be followed by other commands in the script
• When you redirect the output from a script to a file, you must make sure that the user sees any prompts the script displays.

• The `/dev/tty` device is a pseudonym for the screen the user is working on; you can use this device to refer to the user’s screen without knowing which device it is.

```
$ cat to_screen1
echo "message to standard output"
echo "message to standard error" 1>&2
echo "message to the user" > /dev/tty

$ to_screen1 > out 2> err
message to the user
$ cat out
message to standard output
$ cat err
message to standard error
```
• Putting this command at the beginning of the previous script changes where the output goes

```
exec > /dev/tty
```

redirects the output from a script to the user’s screen

```
$ cat to_screen2
exec > /dev/tty
echo "message to standard output"
echo "message to standard error" 1>&2
echo "message to the user" > /dev/tty
```

Following the exec command, all output sent to standard output goes to `/dev/tty` (the screen)

```
$ to_screen2 > out 2> err
message to standard output
message to the user
```

• One disadvantage of using exec to redirect the output to `/dev/tty` is that all subsequent output is redirected unless you use exec again in the script

• You can also redirect the input to read (standard input) so that it comes from `/dev/tty` (the keyboard):

```
read name < /dev/tty
```

or

```
exec < /dev/tty
```
**trap: Catches a Signal**

- A *signal* is a report to a process about a condition. Linux uses signals to report interrupts generated by the user (for example, pressing the interrupt key) as well as bad system calls, illegal instructions, and other conditions.
- The `trap` builtin catches, or traps, one or more signals, allowing you to direct the actions a script takes when it receives a specified signal.
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Number</th>
<th>Generating condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not a real signal</td>
<td>EXIT</td>
<td>0</td>
<td>Exit because of exit command or reaching the end of the program (not an actual signal but useful in trap)</td>
</tr>
<tr>
<td>Hang up</td>
<td>SIGHUP or HUP</td>
<td>1</td>
<td>Disconnect the line</td>
</tr>
<tr>
<td>Terminal interrupt</td>
<td>SIGINT or INT</td>
<td>2</td>
<td>Press the interrupt key (usually CONTROL-C)</td>
</tr>
<tr>
<td>Quit</td>
<td>SIGQUIT or QUIT</td>
<td>3</td>
<td>Press the quit key (usually CONTROL-SHIFT-I or CONTROL-SHIFT-A)</td>
</tr>
<tr>
<td>Kill</td>
<td>SIGKILL or KILL</td>
<td>9</td>
<td>The kill command with the <code>-9</code> option (cannot be trapped; use only as a last resort)</td>
</tr>
<tr>
<td>Software termination</td>
<td>SIGTERM or TERM</td>
<td>15</td>
<td>Default of the <code>kill</code> command</td>
</tr>
<tr>
<td>Stop</td>
<td>SIGTSTP or TSTP</td>
<td>20</td>
<td>Press the suspend key (usually CONTROL-Z)</td>
</tr>
<tr>
<td>Debug</td>
<td>DEBUG</td>
<td></td>
<td>Executes <code>commands</code> specified in the <code>trap</code> statement after each command (not an actual signal but useful in trap)</td>
</tr>
<tr>
<td>Error</td>
<td>ERR</td>
<td></td>
<td>Executes <code>commands</code> specified in the <code>trap</code> statement after each command that returns a nonzero exit status (not an actual signal but useful in trap)</td>
</tr>
</tbody>
</table>
• When it traps a signal, a script takes whatever action you specify: It can remove files or finish any other processing as needed, display a message, terminate execution immediately, or ignore the signal

• If you do not use trap in a script, any of the six actual signals listed in the Table above

\[
\text{\texttt{trap [\textquoteleft\texttt{commands}\textquoteright\]} [\texttt{signal}]}}
\]

• The optional \texttt{commands} part specifies the commands that the shell executes when it catches one of the signals specified by \texttt{signal}

• The \texttt{signal} can be a signal name or number—for example, INT or 2

• If \texttt{commands} is not present, \texttt{trap} resets the \texttt{trap} to its initial condition, which is usually to exit from the script
The *trap* builtin does not require single quotation marks around *commands* as shown in the preceding syntax, but it is a good practice to use them.

The single quotation marks cause shell variables within the *commands* to be expanded when the signal occurs, not when the shell evaluates the arguments to trap.

Even if you do not use any shell variables in the *commands*, you need to enclose any command that takes arguments within either single or double quotation marks.

Quoting the *commands* causes the shell to pass to trap the entire command as a single argument.
• After executing the **commands**, the shell resumes executing the script where it left off

```
$ cat inter
#!/bin/bash
trap 'echo PROGRAM INTERRUPTED; exit 1' INT
while true
do
    echo "Program running."
    sleep 1
done
$ inter
Program running.
Program running.
Program running.
CONTROL-C
PROGRAM INTERRUPTED
$
```

In place of true you can use the **(null)** builtin, which is written as a colon and always returns a 0 (true) status
The following shell script, named **addbanner**, uses two traps to remove a temporary file when the script terminates normally or owing to a hangup, software interrupt, quit, or software termination signal.
• When called with one or more filename arguments, `addbanner` loops through the files, adding a header to the top of each
• The header is kept in a file named `~/banner`
• The second trap uses a 0 in place of `signal-number`, which causes trap to execute its command argument whenever the script exits because it receives an `exit` command or reaches its end
• Standard error of the second trap is sent to `/dev/null` for cases in which trap attempts to remove a nonexistent temporary file
• In those cases `rm` sends an error message to standard error; because standard error is redirected, the user does not see this message
**kill: Aborts a Process**

- The **kill** builtin sends a signal to a process or job

  \[\text{kill} \ [-\text{signal}] \ \text{PID}\]

- where **signal** is the signal name or number (for example, \textbf{INT} or 2) and \textbf{PID} is the process identification number of the process that is to receive the signal

- You can specify a job number as \%\textit{n} in place of \textbf{PID}

- If you omit **signal**, \textbf{kill} sends a TERM (software termination, number 15) signal

- The following command sends the TERM signal to job number 1

  \$ \textbf{kill} \ -\text{TERM} \ \%1

- To terminate a program, first try INT (press CONTROL-C, if the job is in the foreground)

- Because an application can be written to ignore these signals, you may need to use the \textbf{KILL} signal, which cannot be trapped or ignored; it is a “sure kill.”
Expressions
Arithmetic Evaluation

• The shell performs arithmetic assignments in a number of ways. One is with arguments to the `let` built-in:

  ```
  $ let "VALUE=VALUE * 10 + NEW"
  ```

• In the preceding example, the variables `VALUE` and `NEW` contain integer values.

• Double quotation marks must enclose a single argument, or expression, that contains spaces. Because most expressions contain spaces and need to be quoted, bash accepts `((expression))` as a synonym for `let "expression"`, obviating the need for both quotation marks and dollar signs:

  ```
  $ ((VALUE=VALUE * 10 + NEW))
  ```
Because each argument to `let` is evaluated as a separate expression, you can assign values to more than one variable on a single line:

```bash
$ let VALUE=VALUE*10+NEW
```

In the following example, the asterisk (*) does not need to be quoted because the shell does not perform pathname expansion on the right side of an assignment.

```bash
$ let "COUNT = COUNT + 1" VALUE=VALUE*10+NEW
```

You need to use commas to separate multiple assignments within a set of double parentheses:

```bash
$ ((COUNT = COUNT + 1, VALUE=VALUE*10+NEW))
```
Arithmetic evaluation versus arithmetic expansion

**tip** Arithmetic evaluation differs from arithmetic expansion. As explained on page 342, arithmetic expansion uses the syntax `$((expression))`, evaluates `expression`, and replaces `$((expression))` with the result. You can use arithmetic expansion to display the value of an expression or to assign that value to a variable.

Arithmetic evaluation uses the `let expression` or `((expression))` syntax, evaluates `expression`, and returns a status code. You can use arithmetic evaluation to perform a logical comparison or an assignment.

```
$ cat age2
#!/bin/bash
echo -n "How old are you? 
read age
if ((30 < age && age < 60)); then
    echo "Wow, in $((60-age)) years, you'll be 60!"
else
    echo "You are too young or too old to play."
fi

$ age2
How old are you? 25
You are too young or too old to play.
```
Logical Evaluation (Conditional Expressions)

• The syntax of a conditional expression is:

  ```
  [[ expression ]] 
  ```

• where expression is a Boolean (logical) expression. You must precede a variable name with a dollar sign ($) within expression.

• The result of executing this builtin, like the test builtin, is a return status.

• Where the test builtin uses –a as a Boolean AND operator, `[[ expression ]]` uses `&&`

• Similarly, where test uses –o as a Boolean OR operator, `[[ expression ]]` uses `||`

• You must surround the `[[ and ]]` tokens with whitespace or a command terminator, and place dollar signs before the variables:

  ```
  if [[ 30 < $age && $age < 60 ]]; then
  ```
• You can also use test’s relational operators –gt, –ge, –lt, –le, –eq, and –ne:

```bash
if [[ 30 -lt $age && $age -lt 60 ]]; then
```

• The > and < operators compare strings for order (for example, "aa" < "bbb"). The = operator tests for pattern match, not just equality: `[[ string = pattern ]]` is true if `string` matches `pattern`

• For example, `[[ artist = a* ]]` is true (= 0), whereas `[[ a* = artist ]]` is false (= 1):

```bash
$ [[ artist = a* ]]
$ echo $?  
0
$ [[ a* = artist ]]
$ echo $?  
1
```
The next example uses a command list that starts with a compound condition. The condition tests that the directory `bin` and the file `src/myscript.bash` exist.

- If this is true, `cp` copies `src/myscript.bash` to `bin/myscript`.
- If the copy succeeds, `chmod` makes `myscript` executable. If any of these steps fails, `echo` displays a message.

```
$ [[ -d bin && -f src/myscript.bash ]] && cp src/myscript.bash /bin/myscript && chmod +x bin/myscript || echo "Cannot make executable version of myscript"
```
String Pattern Matching

<table>
<thead>
<tr>
<th>Table 11-7</th>
<th>String operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operator</strong></td>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>#</td>
<td>Removes minimal matching prefixes</td>
</tr>
<tr>
<td>##</td>
<td>Removes maximal matching prefixes</td>
</tr>
<tr>
<td>%</td>
<td>Removes minimal matching suffixes</td>
</tr>
<tr>
<td>%%%</td>
<td>Removes maximal matching suffixes</td>
</tr>
</tbody>
</table>

- The syntax for these operators is:

```$\{\text{varname op pattern}\}$```
• These operators are commonly used to manipulate pathnames so as to extract or remove components or to change suffixes:

```bash
$ SOURCEFILE=/usr/local/src/prog.c
$ echo ${SOURCEFILE#/*/}
local/src/prog.c
$ echo ${SOURCEFILE##/*/}
prog.c
$ echo ${SOURCEFILE%//*/}
/usr/local/src
$ echo ${SOURCEFILE%%/*/}

$ echo ${SOURCEFILE%.c}
/usr/local/src/prog
$ CHOPFIRST=${SOURCEFILE#/*/}
$ echo $CHOPFIRST
local/src/prog.c
$ NEXT=${CHOPFIRST%//*/}
$ echo $NEXT
local
```
Here the string-length operator, \${#name}, is replaced by the number of characters in the value of name:

```
$ echo $SOURCEFILE
/usr/local/src/prog.c
$ echo ${#SOURCEFILE}
21
```
Operators

- Arithmetic expansion and arithmetic evaluation use the same syntax, precedence, and associativity of expressions as the C language. Check pages 461-463
- The pipe token has higher precedence than operators

```bash
$ cmd1 | cmd2 || cmd3 | cmd4 && cmd5 | cmd6
```
- is interpreted as if you had typed:

```bash
$ (((cmd1 | cmd2) || (cmd3 | cmd4)) && (cmd5 | cmd6))
```

**tip** Do not rely on the precedence rules when you use compound commands. Instead, use parentheses to explicitly state the order in which you want the shell to interpret the commands.
The remainder operator (%) gives the remainder when its first operand is divided by its second.

The && operator causes the shell to test the exit status of the command preceding it.

If the command succeeded, bash executes the next command; otherwise, it skips the remaining commands on the command line.

```bash
$ N=10
$ echo $N
10
$ echo $((-N+3))
12
$ echo $N
9
$ echo $((N++ - 3))
6
$ echo $N
10
```

```bash
$ mkdir bkpup && cp -r src bkpup
```
• The `||` separator also causes bash to test the exit status of the first command but has the opposite effect: The remaining command(s) are executed only if the first one failed (that is, exited with nonzero status):

```bash
$ mkdir bkup || echo "mkdir of bkup failed" >> /tmp/log
```

• you could combine the previous two examples as:

```bash
$ (mkdir bkup && cp -r src bkup) || echo "mkdir failed" >> /tmp/log
```

• In the absence of parentheses, `&&` and `||` have equal precedence and are grouped from left to right
• The following examples use the true and false utilities. These utilities do nothing and return *true (0) and false (1) exit statuses, respectively*:

```
$ false; echo $?  
1
```

• The `$?` variable holds the exit status of the preceding command

```
$ true || false && false
$ echo $?  
1

$ (true || false) && false
$ echo $?  
1

$ false && false || true
$ echo $?  
0

$ (false && false) || true
$ echo $?  
0
```
The ternary operator, `? :`, decides which of two expressions should be evaluated, based on the value returned from a third expression:

```
$ ((N=10,Z=0))
$ echo $((N || ((Z+=1)) ))
1
$ echo $Z
0
```

Because the value of `N` is nonzero, the result of the `||` (OR) operation is 1 (true), no matter what the value of the right side is. As a consequence `((Z+=1))` is never evaluated and `Z` is not incremented.

```
$ ((N=10,Z=0,COUNT=1))
$ ((T>N>COUNT?++Z:--Z))
$ echo $T
1
$ echo $Z
1
```
• The following commands use the syntax `base#n` to assign base 2 (binary) values

```bash
$ ((v1=2#0101))
$ ((v2=2#0110))
$ echo "$v1 and $v2"
5 and 6
```

• Next the bitwise AND operator (&) selects the bits that are on in both 5

```bash
$ echo $(( v1 & v2 ))
4
```
• The bitwise inclusive OR operator (|) selects the bits that are on in either 0101 or 0110, resulting in 0111, which is 7 decimal

• The Boolean OR operator (||) produces a result of 1 if either of its operands is nonzero and a result of 0 otherwise

```bash
$ echo $(( v1 && v2 ))
1
$ echo $(( v1 | v2 ))
7
$ echo $(( v1 || v2 ))
1
```

• Next the bitwise exclusive OR operator (^) selects the bits that are on in either, but not both, of the operands 0101 and 0110, yielding 0011, which is 3 decimal

```bash
$ echo $(( v1 ^ v2 ))
3
$ echo $(( ! v1 ))
0
$ echo $(( v1 < v2 ))
1
$ echo $(( v1 > v2 ))
0
```
Shell Programs
A Recursive Shell Script

• To avoid circularity a recursive definition must have a special case that is not self-referential
• A number of Linux system utilities can operate recursively. See the –R option to the chmod, chown, and cp utilities for examples
• Solve the following problem by using a recursive shell function:

Write a shell function named makepath that, given a pathname, creates all components in that pathname as directories. For example, the command makepath a/b/c/d should create directories a, a/b, a/b/c, and a/b/c/d. (The mkdir utility supports a –p option that does exactly this. Solve the problem without using mkdir –p.)
By giving the following command from the shell you are working in, you turn on debugging tracing so that you can watch the recursion work:

```
$ set -o xtrace
```
In the following example, the first line that starts with `+` shows the shell calling `makepath`.

The `makepath` function is called from the command line with arguments of `a/b/c`. Subsequently it calls itself with arguments of `a/b` and finally `a`.

All the work is done (using `mkdir`) as each call to `makepath` returns.

```bash
$ makepath a/b/c
+ makepath a/b/c
+ [[ 5 -eq 0 ]]
+ [[  -d a/b/c ]] 
+ [[  a/b = \a\b\c ]]
+ makepath a/b
+ [[ 3 -eq 0 ]]
+ [[  -d a/b ]]
+ [[  a = \a\b ]]
```
+ makepath a
+ [[ 1 -eq 0 ]]
+ [[ -d a ]]
+ [[ a = \\a ]]
+ mkdir a
+ return 0
+ mkdir a/b
+ return 0
+ mkdir a/b/c
+ return 0

$ makepath /a/b
+ makepath /a/b
+ [[ 4 -eq 0 ]]
+ [[ -d /a/b ]]
+ [[ /a = \\\\a\\\\b ]]
+ makepath /a
+ [[ 2 -eq 0 ]]
+ [[ -d /a ]]
+ [[ '' = \\a ]]
+ makepath
+ [[ 0 -eq 0 ]]
+ return 0
+ mkdir /a
mkdir: cannot create directory '/a': Permission denied
+ return 1
+ return 1
The recursion stops when `makepath` is denied permission to create the `/a` directory.

**Use local variables with recursive functions**

**tip** The preceding example glossed over a potential problem that you may encounter when you use a recursive function. During the execution of a recursive function, many separate instances of that function may be active simultaneously. All but one of them are waiting for their child invocation to complete.

Because functions run in the same environment as the shell that calls them, variables are implicitly shared by a shell and a function it calls so that all instances of the function share a single copy of each variable. Sharing variables can give rise to side effects that are rarely what you want. As a rule, you should use `typeset` to make all variables of a recursive function be local variables. See page 435 for more information.