The Length of a String The length of a string is a relatively simple matter; the regular expression .* signifies to **expr** that it has to print the number of characters matching the pattern, i.e., the length of the entire string:

```
$ expr "abcdefghijkl" : '.*'          Space on either side of : required
12
```

Here, **expr** has counted the number of occurrences of any character (.*). This feature is useful in validating data entry. Consider that you want to validate the name of a person accepted through the keyboard so that it doesn't exceed, say, 20 characters in length. The following **expr** sequence can be quite useful for this task:

```
while echo "Enter your name: \c" ; do          echo always returns true
    read name
    if [ `expr "$name" : '.*'` -gt 20 ] ; then
        echo "Name too long"
    else
        break                                  break terminates a loop
    fi
done
```

Extracting a Substring **expr** can extract a string enclosed by the escaped characters \(and \). If you wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
$ stg=2003
$ expr "$stg" : '..\(..\)'          Extracts last two characters
03
```

Note the pattern group \(..\). This is the tagged regular expression (TRE) used by **sed** *(15.11.3)*, but it is used here with a somewhat different meaning. It signifies that the first two characters in the value of $stg have to be ignored and two characters have to be extracted from the third character position. (There's no \1 and \2 here.)

Locating Position of a Character **expr** can also return the location of the first occurrence of a character inside a string. To locate the position of the character d in the string value of $stg, you have to count the number of characters which are not d ([^d]*), followed by a d:

```
$ stg=abcdefgh ; expr "$stg" : '[^d]*d'
4
```

expr also duplicates some of the features of the **test** statement, and also uses the relational operators in the same way. They are not pursued here because **test** is a built-in feature of the shell, and is consequently faster. The Korn shell and Bash have built-in string handling facilities; they don't need **expr**. These features are taken up in Chapter 24.

16.10 $0: CALLING A SCRIPT BY DIFFERENT NAMES

In our discussion on links *(13.2.2)*, we raised the possibility of calling a file by different names and doing different things depending on the name by which it is called. In fact, there are a number of UNIX commands that do exactly that. Now that we know how to extract a string with **expr**, it's time we designed a single script, **comc.sh** (Fig. 16.9), that compiles, edits or runs the last modified C program. The script file will have three more names, but before developing it, let's understand the compiling mechanism used by the **cc** or **gcc** compiler.

A C program has the `.c` extension. When compiled with **cc** *filename*, it produces an executable file named **a.out**. However, we can provide a different name to the executable using the **-o** option. For instance, **cc -o foo foo.c** creates an executable named **foo**. We must be able to extract the "base" filename after dropping the extension, and with **expr** it should be a simple matter.

```
#!/bin/sh
# comc.sh: Script that is called by different names
#
lastfile=`ls -t *.c | head -1`
command=$0                          # Assigning a special parameter to a variable - OK
executable=`expr $lastfile : '\(.*\).c'`     # Removes .c; foo.c becomes foo
case $command in
        *runc) $executable ;;                # Runs the executable
         *vic) vi $lastfile ;;
        *comc) cc -o $executable $lastfile &&
               echo "$lastfile compiled successfully" ;;
esac
```

Fig. 16.9 comc.sh

First, we store the name of the C program that was last modified in the variable `lastfile`. Next, we extract the base filename by dropping the `.c` extension using the TRE feature of **expr**. The **case** conditional now checks the name (saved in the variable `command`) by which the program is invoked. Observe that the first option (`runc`) simply executes the value evaluated by the variable `executable`. The only thing left to do now is to create three links:

```
ln comc.sh comc
ln comc.sh runc
ln comc.sh vic
```

Now you can run **vic** to edit the program, **comc** to compile it and **runc** to execute the object code. We'll only compile it here:

```
$ comc
hello.c compiled successfully
```

Note that this script works only with a C program that is stored, along with any functions, in one file. If functions are stored in separate files, this script won't work. In that case, **make** is the solution. **make** is discussed in Appendix B.

16.11 while: LOOPING

None of the pattern scanning scripts developed so far offers the user another chance to rectify a faulty response. Loops let you perform a set of instructions repeatedly. The shell features three types of loops—**while, until** and **for**. All of them repeat the instruction set enclosed by certain keywords as often as their control command permits.

The **while** statement should be quite familiar to most programmers. It repeatedly performs a set of instructions till the control command returns a true exit status. The general syntax of this command is as follows:

```
while condition is true
do                                  Note the do keyword
    commands
done                                Note the done keyword
```

The *commands* enclosed by **do** and **done** are executed repeatedly as long as *condition* remains true. You can use any UNIX command or **test** as the *condition*, as before.

We'll start with an orthodox **while** loop application. The script, **emp5.sh** (Fig. 16.10), accepts a code and description in the same line, and then writes the line to `newlist`. It then prompts you for more entries. The loop iteration is controlled by the value of `$answer`.

```
#!/bin/sh
# emp5.sh: Shows use of the while loop
#
answer=y                            # Must set it to y first to enter the loop
while [ "$answer" = "y" ]                       # The control command
do
    echo "Enter the code and description: \c" >/dev/tty
    read code description                       # Read both together
    echo "$code|$description" >> newlist            # Append a line to newlist
    echo "Enter any more (y/n)? \c" >/dev/tty
    read anymore
    case $anymore in                            # Also accepts yes, YES etc.
        y*|Y*) answer=y ;;                      # Also accepts no, NO etc.
        n*|N*) answer=n ;;                      # Any other reply means y
            *) answer=y ;;
    esac                    while.sh
done
```

Fig. 16.10 **emp5.sh**

We have redirected the output of two **echo** statements to `/dev/tty` for reasons that will be apparent later. We'll make a small, but significant modification later, but let's run it first:

```
$ emp5.sh
Enter the code and description: 03 analgesics
Enter any more (y/n)? y
Enter the code and description: 04 antibiotics
Enter any more (y/n)? [Enter]                       No response, assumed to be y
Enter the code and description: 05 OTC drugs
Enter any more (y/n)? n
```

When you see the file `newlist`, you'll know what you have actually achieved:

```
$ cat newlist
03|analgesics
04|antibiotics
05|OTC drugs
```

Did redirection with /dev/tty achieve anything here? No, nothing yet, but after we make a small change in the script, it will. Note that you added a record to newlist with the >> symbol. This causes newlist to be opened every time **echo** is called up. The shell avoids such multiple file openings and closures by providing a redirection facility at the **done** keyword itself:

```
done > newlist
```

Make this change in the script and remove the redirection provided with the >> symbols. This form of redirection speeds up execution time as newlist is opened and closed only once. Because this action redirects the standard output of all commands inside the loop, we redirected some statements to /dev/tty so that they can't be redirected again at the **done** keyword.

> **Note** Redirection is also available at the **fi** and **esac** keywords, and includes input redirection and piping:
>
> | | |
> |---|---|
> | `done < param.lst` | *Statements in loop take input from* `param.lst` |
> | `done \| while true` | *Pipes output to a* `while` *loop* |
> | `fi > foo` | *Affects statements between* `if` *and* `fi` |
> | `esac > foo` | *Affects statements between* `case` *and* `esac` |

16.11.1 Using while to Wait for a File

Let's now consider an interesting **while** loop application. There are situations when a program needs to read a file that is created by another program, but it also has to wait till the file is created. The script, **monitfile.sh** (Fig. 16.11), periodically monitors the disk for the existence of the file, and then executes the program once the file has been located. It makes use of the external **sleep** command that makes the script pause for the duration (in seconds) as specified in its argument.

```
#!/bin/sh
# monitfile.sh: Waits for a file to be created
#
while [ ! -r invoice.lst ]      # while the file invoice.lst can't be read
do
   sleep 60                     # Sleep for 60 seconds
done
alloc.pl              # Execute this program after exiting the while loop
```

Fig. 16.11 monitfile.sh

The loop executes as long as the file invoice.lst can't be read (! -r means not readable). If the file becomes readable, the loop is terminated and the program **alloc.pl** is executed. This script is an ideal candidate to be run in the background like this:

```
alloc.pl &
```

We used the **sleep** command to check every 60 seconds for the existence of the file. **sleep** is also quite useful in introducing some delay in shell scripts.

16.11.2 Setting Up an Infinite Loop

Suppose you, as the system administrator, want to see the free space available on your disks every five minutes. You need an infinite loop, and it's best implemented by using **true** as a dummy control command with **while**. **true** does nothing except return a true exit status. Another command named **false** returns a false value. You can set up this loop in the background as well:

```
while true ; do                    This form is also permitted
   df -t                           df reports the free space on the disk
   sleep 300
done &                             & after done runs loop in background
```

With the job now running in the background, you can continue your other work, except that every five minutes you could find your screen filled with **df** output *(17.6.1)*. You can't use the interrupt key to kill it; you'll have to use **kill $!**, which kills the last background job *(10.8.1)*.

> **Note** The shell also offers an **until** statement which operates with a reverse logic used in **while**. With **until**, the loop body is executed as long as the condition remains *false*. Some people would have preferred to have written a previous **while** control command as **until [-r invoice.lst]**. This form is easily intelligible.

16.12 for: LOOPING WITH A LIST

The shell's **for** loop differs in structure from the ones used in other programming languages. There is no three-part structure as used in C, **awk** and **perl**. Unlike **while** and **until**, **for** doesn't test a condition, but uses a list instead:

```
for variable in list
do
   commands                        Loop body
done
```

The loop body also uses the keywords **do** and **done**, but the additional parameters here are *variable* and *list*. Each whitespace-separated word in *list* is assigned to *variable* in turn, and *commands* are executed until *list* is exhausted. A simple example can help you understand things better:

```
$ for file in chap20 chap21 chap22 chap23 ; do
>    cp $file ${file}.bak
>    echo $file copied to $file.bak
> done
chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak
chap23 copied to chap23.bak
```

The *list* here comprises a series of character strings (chap20 and onwards, representing filenames) separated by whitespace. Each item in the list is assigned to the variable `file`. `file` first gets the value chap20, then chap21, and so on. Each file is copied with a `.bak` extension and the completion message displayed after every file is copied.

16.12.1 Possible Sources of the List

The list can consist of practically any of the expressions that the shell understands and processes. `for` is probably the most often used loop in the UNIX system, and it's important that you understand it thoroughly.

List from Variables You can use a series of variables in the command line. They are evaluated by the shell before executing the loop:

```
$ for var in $PATH $HOME $MAIL ; do echo "$var" ; done
/bin:/usr/bin:/home/local/bin:/usr/bin/X11:.:/oracle/bin
/home/henry
/var/mail/henry
```

You have to provide the semicolons at the right places if you want to enter the entire loop in a single line. The three output lines represent the values of the three environment variables.

List from Command Substitution You can also use command substitution to create the list. The following **for** command line picks up its list from the file `clist`:

```
for file in `cat clist`
```

This method is most suitable when the list is large and you don't consider it practicable to specify its contents individually. It's also a clean arrangement because you can change the list without having to change the script.

List from Wild-cards When the list consists of wild-cards, the shell interprets them as *filenames*. **for** is thus indispensable for making substitutions in a set of files with **sed**. Take for instance this loop which works on every HTML file in the current directory:

```
for file in *.htm *.html ; do
  sed 's/strong/STRONG/g
  s/img src/IMG SRC/g' $file > $$
  mv $$ $file
  gzip $file
done
```

In this loop, each HTML filename is assigned to the variable `file` in turn. **sed** performs some substitution on each file and writes the output to a temporary file. This filename is numeric—expanded from the variable $$ (the PID of the current shell). The temporary file is written back to the original file with **mv**, and the file is finally compressed with **gzip**.

List from Positional Parameters **for** is also used to process positional parameters that are assigned from command line arguments. The next script, **emp6.sh** (Fig. 16.12), scans a file repeatedly for each argument. It uses the shell parameter **"$@"** (and not **$***) to represent all command line arguments.

```
#!/bin/sh
# emp6.sh -- Using a for loop with positional parameters
#
for pattern in "$@" ; do                    # Decided not to use $* - Section 16.7.2
   grep "$pattern" emp.lst || echo "Pattern $pattern not found"
done
```

Fig. 16.12 emp6.sh

Execute the script by passing four arguments, one of which is a multiword string:

```
$ emp6.sh 2345 1265 "jai sharma" 4379
2345|j.b. saxena     |g.m.     |marketing |12/03/45|8000
1265|s.n. dasgupta   |manager  |sales     |12/09/63|5600
9876|jai sharma      |director |production|12/03/50|7000
Pattern 4379 not found
```

Since **for** is mostly used with "$@" to access command line arguments, a blank list defaults to this parameter. Thus, these two statements mean the same thing:

```
for pattern in "$@"
for pattern                                   "$@" is implied
```

Note that the script won't work properly if we replaced "$@" with $*. Make this change and then see for yourself how the script behaves.

16.12.2 basename: Changing Filename Extensions

We'll discuss yet another external command, **basename**, only because it's most effective when used inside a **for** loop. Working in tandem, they are quite useful in changing the extensions of a group of files. **basename** extracts the "base" filename from an absolute pathname:

```
$ basename /home/henry/project3/dec2bin.pl
dec2bin.pl
```

When **basename** is used with a second argument, it strips off the string from the first argument:

```
$ basename ux2nd.txt txt                      txt stripped off
ux2nd.
```

You can now use this feature to rename filename extensions from txt to doc:

```
for file in *.txt ; do                        Stores left part of filename
  leftname=`basename $file txt`
  mv $file ${leftname}doc
done
```

If **for** picks up seconds.txt as the first file, leftname stores seconds (without a dot). **mv** simply adds a .doc to the extracted string (seconds). You don't need **expr** for this job at all!