Programming Guide

UNIX System

This dokument was prepared with specific references to use of the UNIX system on a particular processer, the Western Electric 3B20S, which is not presently available except for internal use within the Bell System. However, the information contained herein is generally applicable to use of the UNIX system on various processors which are available in the general trade.

PROGRAMMING GUIDE

UNIX SYSTEM

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

1. INTRODUCTION

This volume describes the three main programming languages supported on the UNIX operating system. These languages are:

- Shell—The **shell** language is both a command language (which handles commands entered from a terminal) and a programming language (where commands are specified in a file and the file is executed).

- C Language—A medium-level programming language which was used to write most of the UNIX operating system. This volume describes the grammar and usage of C language, the libraries that provide additional routines, the **cc**(1) command, and two programs that are useful for checking/debugging C programs.

- Fortran—Fortran 77 and a rational Fortran preprocessor (**ratfor**) are available. This volume describes how Fortran 77 is implemented in terms of the variations from the American National Standard and the interfaces to the UNIX operating system. The **ratfor** preprocessor provides a means by which Fortran 77 can be written in a fashion similar to C language. This preprocessor provides (among other things) simplified control-flow statements.

Throughout this volume, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

NOTES

2. AN INTRODUCTION TO SHELL

INTRODUCTION

The **shell** is a command programming language that provides an interface to the UNIX operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while**, **if then else**, **case**, and **for** are available. Two-way communication is possible between the **shell** and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as **shell** input.

The **shell** can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through ***pipes*** can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The **shell** is both a command language and a programming language that provides an interface to the UNIX operating system. This volume describes, with examples, the UNIX operating system **shell**. The "SIMPLE COMMANDS" part of this section covers most of the everyday requirements of terminal users. Some familiarity with the UNIX operating system is an advantage when reading this section; refer to section "BASICS FOR BEGINNERS" in the UNIX System User's Guide. The "SHELL PROCEDURES" part of this section describes those features of the **shell** primarily intended for use within **shell** commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would also be helpful when reading this section. The last part, "KEYWORD PARAMETERS", describes the more advanced features of the **shell**. See Table 2.A for a defined listing of grammar words used in this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

SIMPLE COMMANDS

Simple commands consist of one or more words separated by blanks. The first word is the **name** of the command to be executed; any remaining words are passed as *arguments* to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument *−l* tells ls(1) to print status information, size, and the creation date for each file.

A. Background Commands

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. A list of currently active processes may be obtained using the ps(1) command.

B. Input/Output Redirection

Most commands produce output to the standard output that is initially connected to the terminal. This output may be directed to a file by using the notation ">", for example:

```
ls −l >file
```

The notation *>file* is interpreted by the **shell** and is not passed as an argument to **ls**(1). If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls**(1). Output may be appended to a file using the notation ">>" as follows:

```
ls −l >>file
```

In this case, *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by using the notation "<", for example:

```
wc <file
```

The command **wc**(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc −l <file
```

can be used.

C. Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, between commands as in

```
ls −l | wc
```

Two or more commands connected in this way constitute a ***pipeline,*** and the overall effect is the same as

```
ls −l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc**(1) when there is nothing to read and halting **ls**(1) when the pipe is full.

A ***filter*** is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1) selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "old". Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged-in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints only the number of file names in the current directory containing the string "old".

D. File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file *main.c*. The "**ls -l**" command alone prints the same information about all files in the current directory.

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to **ls(1)** all file names in the current directory that end in *.c*. The character "*" is a pattern that will match any string including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*. The input

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in subdirectories of */usr/fred*. [The **echo(1)** command is a standard UNIX operating system command that prints its arguments, separated by blanks.] This last feature can be expensive requiring a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".". The input

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

E. Quoting

Characters that have a special meaning to the **shell**, such as

```
< > * ? | & .
```

are called metacharacters. A complete list of metacharacters is given in Table 2.B. Any character preceded by a \ is **quoted** and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single **?**, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line, the sequence **\new-line** (or RETURN) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain new-lines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "D. Evaluation and Quoting" in part "KEYWORD PARAMETERS".

F. Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is "**$** ". The prompt may be changed by entering,

```
PS1=newprompt
```

which sets the prompt to be the string " newprompt" . If a new-line is typed and further input is needed, the **shell** will issue the prompt "> ". Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the **shell** to read another command. The other prompt (>) may be changed (for example) by entering:

```
PS2=more
```

G. The Shell and Login

Following the user's **login**(1), the **shell** is called to read and execute commands typed at the terminal. If the user's login directory contains the file *.profile*, then it is assumed to contain commands and is read immediately by the **shell** before reading any commands from the terminal.

H. Summary

ls
Prints the names of files in the current directory.

ls >file
Puts the output from **ls** into *file.*

ls | wc −l
Prints the number of files in the current directory.

ls | grep old
Prints those file names containing the string "old".

ls | grep old | wc −l
Prints the number of files whose name contains the string "old".

cc pgm.c &
Runs cc in the background.

SHELL PROCEDURES

The **shell** may be used to read and execute commands contained in a file. For example, the following call

```
sh file [ args ... ]
```

calls the **shell** to read commands from *file*. Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the positional parameters $1, $2, ... For example, if the file *wg* contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All UNIX operating system files have three independent attributes (often called "permissions"), ***read***, ***write***, and ***execute*** (rwx). The UNIX operating system command **chmod(1)** may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows **shell** procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special **shell** parameter **$*** is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff −T450 −cm $*
```

which simply prepends some arguments to those already given.

A. Control Flow—"for"

A frequent use of **shell** procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnos* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
        grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telnos* that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred" followed by those for "bert".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in w1 w2
do
   command-list
done
```

A ***command-list*** is a sequence of one or more simple commands separated or terminated by a new-line or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new-line or semicolon. A *name* is a **shell** variable that is set to the words *w1 w2* ... in turn each time the ***command-list*** following **do** is executed. If "in w1 w2 ..." is omitted, then the loop is executed once for each positional parameter; that is, **in $*** is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
        create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new-line) is required before **done**.

B. Control Flow—"case"

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
        case $# in
                1) cat >>$1 ;;
                2) cat >>$2 <$1 ;;
                *) echo 'usage: append [ from ] to' ;;
        esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
        append file
```

$# is the string "1". The standard input is appended (copied) onto the end of *file* using the **cat**(1) command, and

```
        append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
        case word in
                pattern) command-list ;;
                ...
        esac
```

The shell attempts to match word with each pattern in the order in which the patterns appear. If a match is found, the associated ***command-list*** is executed and execution of the **case** is complete. Since * is the pattern that matches any string, it can be used for the default case.

Caution: No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

```
        case $# in
                *) ... ;;
                *) ... ;;
        esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

```
for i
do
        case $i in
                -[ocs]    ... ;;
                -*)       echo 'unknown flag $i' ;;
                *.c)      /lib/c0 $i ... ;;
                *)        echo 'unexpected argument $i' ;;
        esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
        -x | -y)        ...
esac
```

is equivalent to

```
case $i in
        -[xy])          ...
esac
```

The usual quoting conventions apply so that

```
case $i in
        \?)             ...
```

will match the character ?.

C. Here Documents

The **shell** procedure *tel* described in subpart "A. Control Flow—**for**" uses the file */usr/lib/telnos* to supply the data for **grep**(1). An alternative is to include this data within the **shell** procedure as a *here* document, as in,

```
for i
do
        grep $i <<!
        ...
        fred mh0123
        bert mh0789
        ...
!
done
```

In this example, the **shell** takes the lines between <<! and ! as the standard input for **grep**(1). The string "!" is arbitrary. The document is being terminated by a line that consists of the string following <<.

Parameters are substituted in the document before it is made available to **grep**(1) as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of "string1" in *file* to "string2". Substitution can be prevented using \ to quote the special character **$** as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

[This version of *edg* is equivalent to the first except that **ed**(1) will print a ? if there are no occurrences of the string **$1**.] Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a *here* document, this latter form is more efficient.

D. Shell Variables

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables *user*, *box*, and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```

will move the *file* from the current directory to the directory */usr/fred/bin.* A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of **ps**(1) to the file */tmp/psa,* whereas,

```
ps a >$tmpa
```

would cause the value of the variable *tmpa* to be substituted.

Except for **$?**, the following are set initially by the **shell**. The **$?** is set after executing each command.

$?	The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.
$#	The number of positional parameters (in decimal). The **$#** is used, for example, in the **append** command to check the number of parameters.
$$	The process number of this **shell** (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example, `ps a >/tmp/ps$$` ... `rm /tmp/ps$$`
$!	The process number of the last process run in the background (in decimal).
$–	The current **shell** flags, such as **–x** and **–v**.

Some variables have a special meaning to the **shell** and should be avoided for general use.

$*MAIL* When used interactively, the **shell** looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the **shell** prints the message "you have mail" before prompting for the next command. This variable is typically set in the file *.profile* in the user's login directory. For example:

```
MAIL=/usr/mail/fred
```

$*HOME* The default argument for the **cd**(1) command. The current directory is used to resolve file name references that do not begin with a / and is changed using the **cd** command. For example,

```
cd /usr/fred/bin
```

makes the current directory */usr/fred/bin*. Then

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command **cd**(1) with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the user's login profile.

$*PATH* A list of directories containing commands (the *search path*). Each time a command is executed by the **shell**, a list of directories is searched for an executable file. If $*PATH* is not set, the current directory, */bin*, and */usr/bin* are searched by default. Otherwise, $*PATH* consists of directory names separated by a colon (:). For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :), */usr/fred/bin*, */bin*, and */usr/bin* are to be searched in that order. In this way, individual users can have their own "private" commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used; a single attempt is made to execute the command.

$*PS1* The primary **shell** prompt string, by default, "$ ".

$*PS2* The **shell** prompt when further input is needed, by default, "> ".

$*IFS* The set of characters used by *blank interpretation* (See "D. Evaluation and Quoting" in part "KEYWORD PARAMETERS".).

E. The "test" Command

The **test** command is intended for use by **shell** programs. For example,

```
test −f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) for a complete specification].

test s	true if the argument *s* is not the null string
test −f file	true if *file* exists
test −r file	true if *file* is readable
test −w file	true if *file* is writable
test −d file	true if *file* is a directory

F. Control Flow—"while"

The actions of the **for** loop and the **case** branch are determined by data available to the **shell**. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

```
while command-list1
do
        command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time around the loop, *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do
        ...
        shift
done
```

is equivalent to

```
for i
do
        ...
done
```

The **shift** command is a **shell** command that renames the positional parameters **$2**, **$3**, ... as **$1**, **$2**, ... and loses **$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test -f file
do
        sleep 300
done
commands
```

will loop until *file* exists. Each time around the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

G. Control Flow—"if"

Also available is a general conditional branch of the form,

```
if command-list
then
   command-list
else
   command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in

```
if test -f file
then
            process file
else
            do something else
fi
```

An example of the use of **if**, **case**, and **for** constructions is given in "I. The Man Command" in part "SHELL PROCEDURES".

A multiple test **if** command of the form

```
if ...
then
        ...
else
        if ...
        then
                ...
        else
                if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

The **touch** command changes the "last modified" time for a list of files. The command may be used in conjunction with **make**(1) to force recompilation of a list of files. The following example is the **touch** command:

```
flag=
for i
do
      case $i in
          -c)     flag=N ;;
          *)      if test -f $i
                      then
                          ln $i junk$$
                          rm junk$$
                  elif test $flag
                  then
                          echo file \'$i\' does not exist
                  else
                          >$i

                  fi ;;
      esac
done
```

The −c flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The **shell** variable *flag* is set to some non-null string if the −c argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it.

The sequence

```
if command1
then    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

The first form, *command-list,* is simply executed. The second form executes *command-list* as a separate process. For example,

```
( cd x; rm junk )
```

executes ***rm junk*** in the directory *x* without changing the current directory of the invoking **shell**.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking **shell** in the directory *x*.

H. Debugging Shell Procedures

The **shell** provides two tracing mechanisms to help when debugging **shell** procedures. The first is invoked within the procedure as

```
set −v
```

(*v* for verbose) and causes linesof the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

```
sh −v proc ...
```

where *proc* is the name of the **shell** procedure. This flag may be used in conjunction with the **−n** flag which prevents execution of subsequent commands. (Note that typing "**set −n**" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set −x
```

will produce an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see what effect they have.) Both flags may be turned off by typing

```
set −
```

and the current setting of the **shell** flags is available as **$−**.

I. The "man" Command

The following is the **man** command which is used to print sections of the UNIX System User's Manual. It is called by entering

```
man sh
man −t ed
man 2 fork
```

In the first call, the manual section for **sh** is printed. Since no section is specified, Section 1 is used. The second call will typeset (**−t** option) the manual section for **ed.** The last call prints the **fork** manual page from Section 2 of the manual.

A version of the **man** command follows:

```
cd /usr/man
: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1
for i
do
      case $i in
            [1-9]*)   s=$i ;;
            −t)     N=t ;;
            −n)     N=n ;;
            −*)     echo unknown flag \'$i\' ;;
            *)      if test −f man$s/$i.$s
                    then
                          ${N}roff man0/${N}aa man$s/$i.$s
                    else
                          : 'look through all manual sections'
```

```
                                    found=no
                                    for j in 1 2 3 4 5 6 7 8 9
                                    do
                                        if test -f man$j/$i.$j
                                        then man $j $i
                                               found=yes
                                        fi
                                    done
                                    case $found in
                                        no) echo '$i: manual page not found'
                                    esac
                            fi ;;
          esac
     done
```

KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a **shell** procedure is invoked. An argument to a **shell** procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking **shell** is not affected. For example,

```
user=fred command
```

will execute **command** with *user* set to *fred.* The **−k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **$1, $2, ...**

The **set** command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set **$1** to the first file name in the current directory, **$2** to the next, etc. Note that the first argument, −, ensures correct treatment when the first file name begins with a −.

A. Parameter Transmission

When a **shell** procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a **shell** procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables *user* and *box* for export. When a **shell** procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking **shell**. It is generally true of a **shell** procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared readonly. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

B. Parameter Substitution

If a **shell** parameter is not set, then the null string is substituted for it. For example, if the variable *d* is not set,

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable *d* if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo * if the variable *d* is not set. Similarly,

```
echo ${d-$1}
```

will echo the value of *d* if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if *d* were not previously set, it will be set to the string ".". (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable *d* if it has one; otherwise, *message* is printed by the **shell** and execution of the **shell** procedure is abandoned. If *message* is absent, a standard message is printed. A **shell** procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command built in to the **shell** and does nothing once its arguments have been evaluated. If any of the variables *user, acct,* or *bin* are not set, the **shell** will abandon execution of the procedure.

C. Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command **pwd**(1) prints on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin*, the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between (\`...\`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a \` must be escaped using a \\. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within **shell** procedures. An example of such a command is **basename** which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main". Its use is illustrated by the following fragment from a **cc(1)** command.

```
case $A in
        ...
        *.c)        B=`basename $A .c`
        ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- for i in \`ls −t\`; do ...

  The variable *i* is set
  to the names of files in time order,
  most recent first.

- set \`date\`; echo $6 $2 $3, $4

  will print, e.g.,
  1977 Nov 1, 23:59:59

D. Evaluation and Quoting

The **shell** is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 2.A. Before a command is executed, the following substitutions occur:

1. parameter substitution, e.g., **$user**

2. command substitution, e.g., \`**pwd**\`

   Only one evaluation occurs so that if, for example, the value of the variable *X* is the string "$y" then

   ```
   echo $X
   ```

   will echo "$y".

3. blank interpretation

   Following the above substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, "blanks" are the characters of the string "*$IFS*". By default, this string consists of blank, tab, and new-line. The null string is not regarded as a word unless it is quoted. For example,

   ```
   echo ' '
   ```

   will pass on the null string as the first argument to **echo**, whereas

   ```
   echo $null
   ```

   will call **echo** with no arguments if the variable *null* is not set or set to the null string.

4. file name generation

   Each word is then scanned for the file pattern characters *, ?, and [...]; and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and '...', a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \\.

$	parameter substitution
\`	command substitution
"	ends the quoted string
\\	quotes the special characters $ \` " \\

For example,

```
echo " $x"
```

will pass the value of the variable *x* as a single argument to **echo**. Similarly,

```
echo " $*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo " $1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted. Inputting

```
echo " $@"
```

will pass the positional parameters, unevaluated, to **echo** and is equivalent to

```
echo " $1" " $2" ...
```

The following illustration gives, for each quoting mechanism, the **shell** metacharacters that are evaluated.

metacharacter

	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t	=	terminator
y	=	interpreted
n	=	not interpreted

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable *X* has the value "$y" and if *y* has the value "pqr", then

```
eval echo $X
```

will echo the string "pqr".

In general, the **eval** command evaluates its arguments (as do all commands) and treats the result as input to the **shell**. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as |, following substitution.

E. Error Handling

The treatment of errors detected by the **shell** depends on the type of error and on whether the **shell** is being used interactively. An interactive **shell** is one whose input and output are connected to a terminal [as determined by **gtty**(2)]. A **shell** invoked with the **-i** flag is also interactive.

Execution of a command (see also "G. Command Execution") may fail for any of the following reasons:

- Input/output redirection may fail, e.g., if a file does not exist or cannot be created.

- The command itself does not exist or cannot be executed.
- The command terminates abnormally, e.g., with a "bus error" or "memory fault" signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the **shell** will go on to execute the next command. Except for the last case, an error message will be printed by the **shell**. All remaining errors cause the **shell** to exit from a command procedure. An interactive **shell** will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., if ... then ... done
- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**(1).

The **shell** flag **−e** causes the **shell** to terminate if any error is detected. The following is a list of the UNIX operating system signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination [from **kill**(1)]

The UNIX operating system signals marked with an asterisk "*" as shown in the list produce a core dump if not caught. However, the **shell** itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to **shell** programs are 1, 2, 3, 14, and 15.

F. Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```

The **exit** is another built-in command that terminates execution of a **shell** procedure. The **exit** is required; otherwise, after the trap has been taken, the **shell** will resume executing the procedure at the place where it was interrupted.

UNIX operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "G. Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
    case  $i in
    -c)    flag=N ;;
    *)     if test -f $i
           then
                  ln $i junk$$; rm junk$$
           elif test $flag
           then
                  echo file \'$i\' does not exist
           else
                  >$i
           fi ;;
      esac
done
```

The cleanup action is to remove the file *junk*$$. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the UNIX operating system, it is used by the **shell** to indicate the commands to be executed on exit from the **shell** procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following:

```
trap '' 1 2 3 15
```

is a fragment taken from the **nohup(1)** command which causes the UNIX operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d=`pwd`
for i in *
do
      if test -d $d/$i
      then
            cd $d/$i
            while echo " $i:"  && trap exit 2 && read x
            do
                  trap : 2
                  eval $x
            done
      fi
done
```

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

G. Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork**(2). The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap '' 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and **exec** replaces the **shell** by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is **.c*. Input/output specifications are evaluated left to right as they appear in the command. Some input/output specifications are as follows:

> *word*	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of **shell** input that follow up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to

quote the characters \, $, `, and the first character of *word*. In the latter case, **\new-line** is ignored (e.g., quoted strings).

>& *digit*	The file descriptor *digit* is duplicated using the system call **dup(2)**, and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.
<&−	The standard input is closed.
>&−	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*. Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the **shell** and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the **shell** to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UNIX operating system convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the **shell** command **trap** has no effect for an ignored signal.

H. Invoking the Shell

The following flags are interpreted by the **shell** when it is invoked. If the first character of argument zero is a minus, commands are read from the file *.profile*.

−c *string*	If the **−c** flag is present, then commands are read from *string*.
−s	If the **−s** flag is present or if no arguments remain, commands are read from the standard input. **Shell** output is written to file descriptor 2.
−i	If the **−i** flag is present or if the **shell** input and output are attached to a terminal [as told by **getty(8)**], this **shell** is <u>interactive.</u> In this case, TERMINATE is ignored (so that **kill 0** does not kill an interactive **shell**, and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the **shell**.

TABLE 2.A
GRAMMAR

item:	*word*
	input-output
	name = value
simple-command:	item
	simple-command item
command:	simple-command
	(*command-list*)
	{ *command-list* }
	for *name* **do** *command-list* **done**
	for *name* **in** *word* ... **do** *command-list* **done**
	while *command-list* **do** *command-list* **done**
	until *command-list* **do** *command-list* **done**
	case *word* **in** *case-part* ... **esac**
	if *command-list* **then** *command-list else-part* **fi**
pipeline:	*command*
	pipeline | *command*
andor:	*pipeline*
	andor && *pipeline*
	andor || *pipeline*
command-list:	andor
	command-list ;
	command-list &
	command-list ; *andor*
	command-list & *andor*
input-output:	> *file*
	< *file*
	>> *word*
	<< *word*
file:	*word*
	& *digit*
	& –
case-part:	*pattern*) *command-list* ;;
pattern:	*word*
	pattern | *word*
else-part:	**elif** *command-list* **then** *command-list else-part*
	else *command-list*
empty:	*empty*
word:	a sequence of nonblank characters
name:	a sequence of letters, digits, or underscores starting with a letter
digit:	0 1 2 3 4 5 6 7 8 9

TABLE 2.B

METACHARACTERS AND RESERVED WORDS

(a) *syntactic:*

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

(b) *patterns:*

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

(c) *substitution:*

${...}	substitute shell variable
\`...\`	substitute command output

(d) *quoting:*

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for the $, \` ,\, and "

(e) *reserved words:*

if then else elif fi
case in esac
for while until do done
{ } [ ] test

THE SHELL TUTORIAL

INTRODUCTION

In any programming project, some effort is used to build the end product. The remainder is consumed in building the supporting tools and procedures used to manage and maintain that end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool in such situations. If it is a flexible programming language, it can be used to solve many internal support problems without requiring compilable programs to be written, debugged, and maintained. The most important advantage of a good command language is the ability to get the job done now. For a perspective on the motivations for using a command language in this way, see [1,2,3,4]. Throughout this section references of the type [1 through 10] refer to documents listed in part "REFERENCES".

When users log into a UNIX system, they communicate with an instance of the **shell** that reads commands typed at the terminal and arranges for the execution of the commands entered. Thus, the **shell's** most important function is to provide a good interface for human beings. In addition, a sequence of commands may be preserved for repeated use by saving it in a file, called a *shell procedure, command file,* or *runcom* according to local preference.

Some UNIX system users need little knowledge of the **shell** to do their work while others make heavy use of its programming features. This section may be read in several different ways, depending on the reader's interests. A brief discussion of the UNIX system environment is found in part "OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT". The discussion in part "SHELL BASICS" covers aspects of the **shell** that are important for everyone, while all of part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES" and most of part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" are mainly of interest to those who write **shell** procedures. A group of annotated **shell** procedure examples is given in part "EXAMPLES OF SHELL PROCEDURES". Finally, a brief discussion of efficiency is offered in part "EFFECTIVE AND EFFICIENT SHELL PROGRAMMING". The discussion on efficiency is found in its proper place (at the end) and is intended for those who write especially time-consuming **shell** procedures.

Complete beginners should *not* be reading this section, but should work their way through other available tutorials first. See [10] for an appropriate plan of study.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT

Full understanding of what follows depends on familiarity with the UNIX system; [9] is useful for that, and it would be helpful to read [5] and at least one of [6,7]. For completeness, a short overview of the most relevant concepts are given below.

A. File System

The UNIX system file system's overall structure is that of a rooted tree composed of *directories* and other files. A simple *file name* is a sequence of characters other than a slash (/). A *pathname* is a sequence of directory names followed by a simple file name, each separated from the previous one by a /. If a pathname begins with a /, the search for the file begins at the *root* of the entire tree; otherwise, it begins at the user's *current directory* (also known as the *working directory*). The first kind of name is often called a *full* (or *absolute*) *pathname* because it is invariant with regard to the user's current directory. The latter is often called a *relative pathname,* because it specifies a path relative to the current directory. The user may change the current directory at any time by using the **cd**(1) command. In most cases, a file name and its corresponding pathname may be used interchangeably. Some sample names are:

/	absolute pathname of the root directory of the entire file structure.
/bin	directory containing most of the frequently used public commands.
/a1/tf/jtb/bin	a full (or absolute) pathname typical of multiperson programming projects. This one happens to be a private directory of commands belonging to person *jtb* in project *tf; a1* is the name of a *file system.*
bin/x	a relative pathname; it names file *x* in subdirectory *bin* of the current directory. If the current directory is */*, it names */bin/x*. If, on the other hand, the current directory is */a1/tf/jtb*, it names */a1/tf/jtb/bin/x*.
memox	name of a file in the current directory.

The file system for the UNIX operating system provides special shorthand notations for the current directory and the *parent* directory of the current directory:

. is the generic name of the current directory. A *./memox* names the same file as *memox* if such a file exists in the current directory.

.. is the generic name of the parent directory of the current directory. If the user types:

```
cd ..
```

then the parent directory of your current working directory will become your new current directory.

B. UNIX System Processes

An *image* is a computer execution environment, including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image. Most UNIX system commands execute as separate processes. One process may spawn another using the **fork**(2) system call, which duplicates the image of the original (*parent*) process. The new (*child*) process continues to execute the same program as the parent. The two images are identical, except that each program can determine whether it is executing as parent or child. Each program may continue execution of the image or may abandon it by issuing an **exec**(2) system call, thus initiating execution of another program. In any case, each process is free to proceed in parallel with the other, although the parent most commonly issues a **wait**(2) system call to suspend execution until a child terminates (**exits**).

Figure 2.1 illustrates these ideas. **Program** A is executing (as *process 1)* and wishes to run **program** B. It **forks** and spawns a child (*process 2*) that continues to run **program** A. The child abandons A by **execing** B, while the parent goes to sleep until the child **exits**.

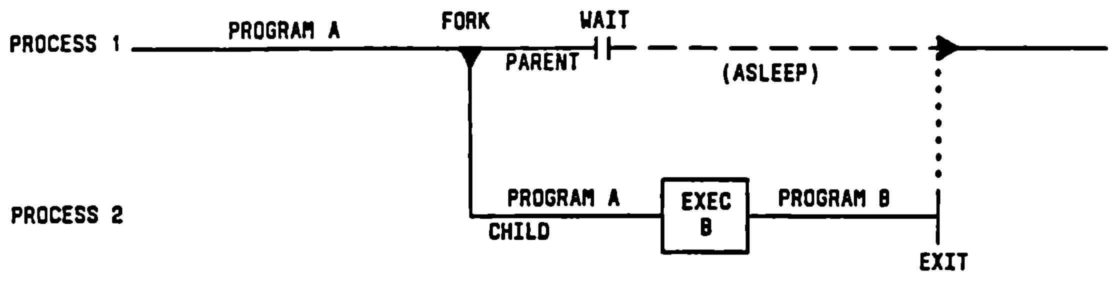

Fig. 2.1—The Shell Executing a Typical UNIX System Command

A child inherits its parent's *open files.* This mechanism permits processes to share common input streams in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations on the file. The **read**(2) and **write**(2) system calls copy a requested number of bytes from and to a file beginning at the position given by the current value of the pointer. As a side effect, the pointer is incremented by the number of bytes transferred yielding the effect of sequential I/O. The **lseek**(2) system call can be used to obtain random-access I/O by setting the pointer to an absolute position within the file or to a position offset either from the end of the file or from the current pointer position.

When a process terminates, it can set an 8-bit *exit status* (see **$?** in "Predefined Special Variables" in subpart "D. Shell Variables") that is available to its parent. This code is *usually* used to indicate success (zero) or failure (nonzero).

Signals indicate the occurrence of events that may have some impact on a process. A signal may be sent to a process by another process from the terminal or by the UNIX system itself. A child process inherits its parent's signals. For most signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to *catch* it and take appropriate action as described in "Interrupt Handling—**trap**" in subpart "D. Control Commands". For example, an INTERRUPT signal may be sent by depressing an appropriate key *(del, break,* or *rubout).* The action taken depends on the requirements of the specific program being executed:

- The **shell** invokes most commands in such a way that they immediately die when an interrupt is received. For example, the **pr**(1) (print) command normally dies allowing the user to terminate unwanted output.

- The **shell** *itself* ignores interrupts when reading from the terminal because it should continue execution even when the user terminates a command like **pr**.

- The editor **ed**(1) chooses to *catch* interrupts so that it can halt its current action (especially printing) without being terminated.

SHELL BASICS

The **shell** [i.e., the **sh**(1) command] implements the command language visible to most UNIX system users. The **shell** reads input from a terminal or a file and arranges for the execution of the requested commands. It is a program written in the C language [8]. The **shell** is *not* part of the operating system but is an ordinary user program.

A. Commands

A ***simple command*** is a sequence of nonblank arguments separated by blanks or tabs. The first argument (numbered *zero*) usually specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. A command may be as simple as:

```
who
```

which prints the login names of users who are currently logged into the system. The following line requests the **pr**(1) command to print files *a*, *b*, and *c*:

```
        pr a b c
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the **shell** (as parent) spawns a new (child) process that immediately executes that program. If the file is marked as being executable but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines, as well as possibly lines meant to be read by other programs. In this case, the **shell** spawns another instance of itself (a *subshell)* to read the file and execute the commands included in it. The **shell forks** to do this, but no **exec** call is made. The **man**(1) command requests that entries in the on-line UNIX System User's Manual be printed on the terminal. For example, the section that describes the **who** and **pr** commands can be printed by entering the following:

```
        man who pr
```

(Incidentally, the **man**(1) command itself is actually implemented as a **shell** procedure.) From the user's viewpoint, compiled programs and **shell** procedures are invoked in exactly the same way. The **shell** determines which implementation has been used rather than requiring the user to do so. This preserves the uniformity of invocation and the ease of changing the choice of implementation for a given command. The actions of the **shell** in executing any of these commands are illustrated in Fig. 2.1.

B. How the Shell Finds Commands

The **shell** normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. The **shell** first attempts to find the command (as given on the command line) in the current directory. If this fails, the **shell** prepends the string */bin* to the name and, finally, */usr/bin*. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example the **pr**(1) and **man**(1) commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex pathname may be given either to locate a file relative to the user's current directory or to access a command via an absolute pathname. If a command name *as given* begins with a /, ./, or ../ (e.g., */bin/sort* or ../cmd), the prepending is *not* performed. Instead, a single attempt is made to execute the command as given.

This mechanism gives the user a convenient way to execute public commands and commands in or *near* the current directory as well as the ability to execute *any* accessible command regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command will not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the *PATH* variable as described in "User-defined Variables" in subpart "D. Shell Variables".

C. Generation of Argument Lists

Command arguments are very often file names. A list of file names can be automatically generated as arguments on a command line by specifying a pattern that the **shell** matches against the file names in a directory.

Most characters in such a pattern match themselves, but there are also special *metacharacters* that may be included in a pattern. These special characters follow:

*	Matches any string *including* the null string
?	Matches *any one* character

[...]	Matches any sequence of characters enclosed within the square brackets. Be warned that square brackets are also used to indicate that the enclosed argument is optional. See "A. Conditional Evaluation—**test**" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" for more details.
[!...]	Any sequence of characters preceded by a ! and enclosed within [...] will match any one character *other* than one of the enclosed characters. Inside square brackets, a pair of characters separated by a – includes in the set all characters lexically within the inclusive range of that pair, so that **[a-de]** is equivalent to **[abcde]**.

For example, the * matches all file names in the current directory. The ***temp*** matches all file names containing string " temp" . A **[a-f]*** matches all file names that begin with *a* through *f*. The **[!0-9]** matches all single-character names other than the digits, and ***.c** matches all file names ending in **.c**. The **/a1/tf/bin/?** matches all single-character file names found in */a1/tf/bin*. This pattern matching capability saves much typing and, more importantly, makes it possible to organize information in large collections of small files that are named in disciplined ways.

Pattern matching has some restrictions. If the first character of a file name is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any file names, then the pattern itself is returned as the result of the match, for example:

```
echo *.c
```

will print:

```
*.c
```

if the current directory contains no files ending in **.c**.

Directory names should not contain the characters *, ?, [, or] because this may cause infinite recursion during pattern matching attempts. This may be changed in a future release.

D. Shell Variables

The **shell** has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*. *Parameters* are the variables which are normally set only on a command line. There are also *positional parameters* (see "Positional Parameters") and *keyword parameters* (see "A. A Command's Environment" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES"). Other variables are simply names to which the user or the **shell** itself may assign string values.

Positional Parameters: When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters*. The argument in position zero on the command line (the name of the **shell** procedure itself) is called **$0**, the first argument is called **$1**, etc. The **shift** command (see "Passing Arguments to the Shell—**shift**" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES") may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the **set** command:

```
set abc def ghi
```

which assigns string1(" abc") to the first positional parameter (**$1**), string2 to the second (**$2**), and string3 to the third (**$3**). For this example, **set** also *unsets* **$4**, **$5**, etc. even if they were previously set. The **$0** may not be assigned a value so that it always refers to the name of the **shell** procedure or to the name of the **shell** (in the login **shell**).

User-defined Variables: The **shell** also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement. Positional parameters can only be set as described in "Positional Parameters". A simple assignment is of the form:

```
name =string
```

Thereafter, **$***name* will yield the value " string". A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment may appear in an assignment statement, but beware since *the* **shell** *performs the assignments from right to left.* The following command line results in the variable *a* acquiring the value " abc":

```
a=$b b=abc
```

The following are examples of simple assignments. ***Double*** quotes around *the right-hand side* allow blanks, tabs, semicolons, and new-lines to be included in " string", while also allowing *variable substitution* (also known as *parameter substitution)* to occur. In *parameter substitution* references to positional parameters and other variable names that are prefaced by **$** are replaced by the corresponding values, if any. ***Single*** quotes inhibit variable substitution. Some examples follow:

```
MAIL=/usr/mail/gas
var=" echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable *var* has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of *, ?, [...]) does *not* apply in this context. Note that the value of **$asterisks** is the literal string " $stars", *not* the string " *****", because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in **$first** and **$second** having the same value:

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces {} to delimit the variable name from any following string. See "Command Grouping—Parentheses and Braces" in "D. Control Commands" and "G. Conditional Substitution" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" for other meanings of braces in the **shell**. In particular, if the character immediately following the name is a letter, digit, or underscore (digit only for positional parameters), then the braces are *required:*

```
a='This is a string'
echo" ${a}ent test"
```

The following variables are used by the **shell**. Some of them are set by the **shell**, and all of them can be set and reset by the user:

HOME is initialized by the **login**(1) program to the name of the user's *login directory,* i.e., the directory that becomes the current directory upon completion of a login. The **cd** command without arguments uses $*HOME* as the directory to switch to. Using this variable helps one to keep full pathnames out of **shell** procedures. This is a big help when the pathname of your login directory is changed (e.g., to balance disk loads).

MAIL is the pathname of a file where your mail is deposited. If *MAIL* is set, then the **shell** checks to see if anything has been added to the file it names and announces the arrival of new mail every time you return to command level (e.g., by leaving the editor). *MAIL* must be set by the user. (The presence of mail in the standard mail file is also announced at login, regardless of whether *MAIL* is set.)

PATH is the variable that specifies where the **shell** is to look when it is searching for commands. Its value is an ordered list of directory pathnames separated by colons. A null character anywhere in that list represents the current directory. The **shell** initializes *PATH* to the list *:/bin:/usr/bin* where, by convention, a null character appears in front of the first colon. Thus if you wish to search your current directory last, rather than first, you would type:

```
PATH=/bin:/usr/bin::
```

where the two colons together represent a colon followed by a null followed by a colon, thus naming the current directory. A user often has a personal directory of commands (say, *$HOME*) and causes it to be searched *before* the */bin* and *usr/bin* directories by using:

```
PATH=:$HOME/bin:/bin:/usr/bin
```

The setting of *PATH* to other than the default value is normally done in a user's *.profile* file (see "The *.profile* File" in subpart "I. Changing the State of the shell and the *.profile* File").

CDPATH is the variable that specifies where the **shell** is to look when searching for the argument of the **cd** command whenever that argument is not null and does not begin with /, ./, or ../ [see cd(1), "A. File System" in part "OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT", and "E. Special Shell Commands" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES"]. The value of *CDPATH* is an ordered list of directory pathnames separated by colons. A null character anywhere in that list represents the current directory. By convention, if the list begins with a colon, a null character is assumed to precede that colon. Initially, *CDPATH* is *unset*, resulting in only the current directory being searched. Thus if you wish the **cd** command to first search your current directory and then your home directory, you would type:

```
CDPATH=:$HOME
```

The setting of *CDPATH* to other than the default value is normally done in a user's *.profile* file (see "The *.profile* File" in subpart "I. Changing the State of the shell and the *.profile* File"). Note that if the **cd** command changes to a directory that is *not* a descendent of the current directory, it writes the full name of the new directory on the diagnostic output (see "Standard Input and Standard Output" and "Diagnostic and Other Outputs" in subpart "F. Redirection of Input and Output").

PS1 is the variable that specifies what string is to be used as the primary *prompt* string. If the **shell** is interactive, it prompts with the value of *PS1* when it expects input. The default value of *PS1* is "$" (a $ followed by a blank).

PS2 is the variable that specifies the secondary prompt string. If the **shell** expects more input when it encounters a new-line in its input, it will prompt with the value of *PS2* The default value of *PS2* is ">" (a > followed by a blank).

IFS	is the variable that specifies which characters are *internal field separators.* These are the characters the **shell** uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set *IFS* to include that delimiter.) The **shell** initially sets *IFS* to include the blank, tab, and new-line characters.

Command Substitution: Any command line can be placed within grave accents (`...`) to capture the output of the command. This concept is known as *command substitution.* The command or commands enclosed between grave accents are first executed by the **shell** and then their output replaces the whole expression, grave accents and all. This feature is often combined with **shell** variables so that

```
today=`date`
```

assigns the string representing the current date to the variable *today* (e.g., **Tue Nov 27 16:01:09 EST 1979**). The command

```
users=`who | wc -l`
```

saves the number of logged-in users in the variable *users.* Any command that writes to the standard output can be enclosed in grave accents. Grave accents (see "E. Quoting Mechanisms") may be nested. The inside sets must be escaped with \. For example:

```
logmsg=`echo Your login directory is\`pwd\``
```

Shell variables can also be given values indirectly by using the **read**(2) command. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named:

```
read first init last
```

will take an input line of the form:

```
G. A. Snyder
```

and have the same effect as if you had typed:

```
first=G.    init=A.    last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

Predefined Special Variables: Several variables have special meanings. The following are set *only* by the **shell**:

$# records the number of *positional* arguments passed to the **shell**, not counting the name of the **shell** procedure itself. The variable **$#** yields the number of the highest-numbered positional parameter that is set. Thus, **sh x a b c** sets **$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$? is the exit status (also referred to as *return code, exit code,* or *value*) of the last command executed. Its value is a decimal string. Most UNIX system commands return 0 to indicate successful completion. The **shell** itself returns the current value of **$?** as *its* exit status.

$$ is the process number of the current process. Since process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names for temporary files. The UNIX system provides no mechanism for the automatic creation and deletion of temporary files. A file exists until it is explicitly removed. Temporary files are generally undesirable. The UNIX system pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
temp=$HOME/temp/$$              # use current process number
ls > $temp                      # to form unique temp file
     commands, some of which use $temp, go here
rm $temp                        # clean up at end
```

$! is the process number of the last process run in the background (using &—see "D. Control Commands" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES"). Again, this is a string of up to five digits.

$− is a string consisting of names of execution flags (see "Execution Flags—**set**" in subpart "F. Redirection of Input and Output" and "G. More about Execution Flags" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES") currently turned on in the **shell**. The **$−** variable might have the value **xv** if you are tracing your output.

E. Quoting Mechanisms

Many characters have a special meaning to the **shell** which is sometimes necessary to conceal. Single quotes (' ') and double quotes (" ") surrounding a string or backslash (\) before a single character provide this function in somewhat different ways. (Grave accents [\` \`] are sometimes called *back quotes* but are used only for command substitution [see "Command Substitution" in subpart "D. Shell Variables"] in the **shell** and do not hide special meanings of any characters.)

Within right single quotes, all characters (except ' itself) are taken literally with any special meaning removed. Thus:

```
stuff='echo $? $*; ls * | wc'
```

results only in the string **echo $? $*; ls * | wc** being assigned to the variable *stuff* but *not* in any other commands being executed.

Within double quotes, the special meaning of certain characters does persist while all other characters are taken literally. The characters that retain their special meaning are **$**, \`, and " itself. Thus, within double quotes, variables are expanded and command substitution takes place. However, any commands in a command substitution are not affected by double quotes outside of the grave accents, so that characters such as * retain their special meaning.

To hide the special meaning of **$**, \`, and " within double quotes, you can precede these characters with a backslash (\). Outside of double quotes, preceding a character with \ is equivalent to placing single quotes around that character. A \ followed by a new-line causes that new-line to be ignored, thus allowing continuation of long command lines.

F. Redirection of Input and Output

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline (see subpart

"G. Command Lines and Pipelines"). Depending on the nature of a command's input or output, the actions taken by a few commands can be varied either for efficiency's sake or to avoid useless actions (such as attempting random access I/O on a terminal).

Standard Input/Output: When a command begins execution, it usually expects that three files are already open—a *standard input,* a *standard output,* and a *diagnostic (error) output.* A number called a *file descriptor* is associated with each of these files. By convention, the file descriptor 0 is associated with standard input, file descriptor 1 with standard output, and file descriptor 2 with diagnostic output. A child process normally inherits these files from its parent. All three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the printer or screen). The **shell** permits them to be redirected elsewhere before control is passed to an invoked command. An argument to the **shell** of the form < *file* or > *file* opens the specified file as the standard input or output, respectively (in the case of output, destroying the previous contents of *file*, if any). An argument of the form >> *file* directs the standard output to the end of *file*, thus providing a way to *append* data to it without destroying its existing contents. In either of the two output cases, the **shell** creates *file* if it does not already exist (thus > **output** alone on a line creates a zero-length file). The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution. Neither blank interpretation nor pattern matching of file names occurs after these substitutions. Thus:

```
echo 'this is a test' > *.ggg
```

and:

```
cat < ?
```

will produce, respectively, a 1-line file named **.ggg* (a rather disastrous name for a file) and an error message (unless you have a file named *?*, which is also *not* a wise choice for a file name—see end of part "C. Generation of Argument Lists").

Diagnostic & Other Outputs: Diagnostic output from UNIX system commands is traditionally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines—see subpart "G. Command Lines and Pipelines".) One can redirect this error output to a file by immediately prepending the number of the file descriptor (i.e., 2 in this case) to either output redirection symbol (> or >>). The following line will append error messages from the **cc**(1) command to file *ERRORS:*

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening blanks or tabs. Otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow one to redirect output associated with any of the first ten file descriptors (numbered 0 through 9) so that, for instance, if **cmd** puts output on file descriptor 9, the following line will capture that output in file *savedata:*

```
cmd 9> savedata
```

A command often generates standard output and error output and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose that **cmd** directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd > standard    2> error    9> data
```

Other forms of input/output redirection are described in "Input/Output Redirection and Control Commands" and "In-line Input Documents" in subpart "D. Control Commands" and "F. Input/Output Redirection Using File Descriptors" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES".

G. Command Lines and Pipelines

A sequence of one or more commands separated by | (or ^) make up a ***pipeline***. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbor(s) by *pipes*, i.e., the *output* of each command (except the last one) becomes the *input* of the next command in line. A ***filter*** is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read larger amounts of data before producing output. The **sort**(1) command is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline: **nroff** (see **troff** in Section 1) is a text formatter whose output may contain reverse line motions; **col**(1) converts these motions to a form that can be printed on a terminal lacking reverse-motion capability; **greek**(1) is used to adapt the output to a specific terminal, here specified by **−Thp**. The flag **−cm** indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted:

```
nroff −cm text | col | greek −Thp
```

H. Examples

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

who
Prints (on the terminal) the list of logged-in users.

who >> log
Appends the list of logged-in users to the end of file *log*.

who | wc −l
Prints the number of logged-in users. (The argument to **wc** is *minus ell.)*

who | pr
Prints a paginated list of logged-in users.

who | sort
Prints an alphabetized list of logged-in users.

who | grep pw
Prints the list of logged-in users whose login names contain the string "pw".

who | grep pw | sort | pr
Prints an alphabetized, paginated list of logged-in users whose login names contain the string "pw".

{ date; who | wc −l; } >> log
Appends (to file *log*) the current date followed by the count of logged-in users (see "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands" for the meaning of {...} in this context).

who | sed 's/ .*//' | sort | uniq −d
Prints only the login names of all users who are logged in more than once.

The **who** command does not *by itself* provide options to yield all of the results which can be obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace **who |** by < */etc/passwd* in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line.

I. Changing of the Shell and .profile State

The state of a given instance of the **shell** includes the values of positional parameters (see "Positional Parameters" in subpart "D. Shell Variables"), user-defined variables (see "User-defined Variables" in subpart "D. Shell Variables"), environmental variables (see "A. A Command's Environment"), modes of execution (see "G. More about Execution Flags"), and the current working directory.

The state of a **shell** may be altered in various ways. These include the **cd** command, several flags that can be set by the user, and a file in one's login directory called *.profile* that is treated specially by the **shell.**

"CD": The **cd**(1) command changes the current directory to the one specified as its argument. This can (and should) be used to change to a convenient place in the directory structure. The **cd** command is often combined with () to cause a subshell to change to a different directory and execute a group of commands without affecting the original **shell.** The first sequence below extracts the component files of the archive file */a1/tf/q.a* and places them in whatever directory is the current one. The second sequence of commands places them in directory */a1/tf.*

```
ar x /a1/tf/q.a
(cd /a1/tf; ar x q.a)
```

The .profile File: When you log in, the **shell** is invoked to read your commands. First, however, the **shell** checks to see if a file name */etc/profile* exists on your UNIX system and, if it does, commands are read from it. The */etc/profile* is used by system administrators to set up variables needed by *all* users. Type

```
cat /etc/profile
```

to see what your system administrator has already done for you. After this, the **shell** proceeds to see if you have a file named *.profile* in your login directory. If so, commands are read and executed from it. For a sample *.profile,* see **profile**(5). Finally, the **shell** is ready to read commands from your standard input—usually the terminal.

The Execution Flags—"set": The **set** command provides the capability of altering several aspects of the behavior of the **shell** by setting certain *shell flags.* In particular, the **x** and v flags may be useful from the terminal. Flags may be **set** by typing, for example:

```
set −xv
```

(to turn on flags **x** and v). The same flags may be turned *off* by typing

```
set +xv
```

These two flags have the following meaning:

−v Input lines are printed as they are read by the **shell.** This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

−x Commands and their arguments are printed as they are executed. (Shell control commands, such as **for**, **while**, etc., are not printed, however.) Note that **−x** causes a trace of *only* those commands that are actually executed, whereas **−v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within **shell** procedures (see "G. More about Execution Flags").

USING THE SHELL AS A COMMAND: SHELL PROCEDURES

A. A Command's Environment

All the variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a **shell** passes to its child processes are those that have been named as arguments to the **export** (see **sh**) command. The **export** command places the named variables in the environments of both the **shell** *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line (see also **−k** flag in "G. More about Execution Flags"). Such variables are placed in the environment of the procedure being invoked. For example:

```
# key_command
echo $a $b
```

is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 key_command
```

then the output is:

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count **$#** (see "Predefined Special Variables" in "D. Shell Variables").

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are *not* reflected in the environment. The changes are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure (see "B. Invoking the Shell"). To obtain a list of variables that have been made **exportable** from the current **shell**, type:

```
export
```

(You will also get a list of variables that have been made **readonly**—see "E. Special Shell Commands".) To get a list of name-value pairs in the current environment, type:

```
env
```

B. Invoking the Shell

The **shell** is an ordinary command and may be invoked in the same way as other commands:

sh proc [arg...] A *new instance of the* **shell** is explicity invoked to read ***proc***. Arguments, if any, can b manipulated as described in "C. Passing Arguments to the Shell; **shift**".

sh —v proc [arg...] This is equivalent to putting **set—v** at the beginning of *proc*. Similarly for the **x**, **e**, **u**, ar **n** flags (see "Execution Flags—**set**" in subpart "I. Changing the State of the Shell and th *profile* File" and "G. More about Execution Flags").

proc [arg...] If *proc* is marked executable and is not a compiled, executable program, the effect is similar to that of ***sh proc [args...]***. An advantage of this form is that *proc* may be found by the search procedure described in "B. How the Shell Finds Commands" and "User-defined Variables" in subpart "D. Shell Variables". Also, variables that have been **exported** in the **shell** will still be **exported** from *proc* when this form is used (because the **shell** only **forks** to read commands from *proc*). Thus any changes made within *proc* to the values of **exported** variables will be passed on to subsequent commands invoked from within *proc*.

C. Passing Arguments to the Shell—"shift"

When a command line is scanned, any character sequence of the form **$*n*** is replaced by the ***n*th** argument to the **shell** counting the name of the **shell** procedure itself as **$0**. This notation permits direct reference to the procedure name and to as many as nine positional parameters (see "Positional Parameters" in subpart "D. Shell Variables"). Additional arguments can be processed using the **shift** command or by using a **for** loop (see "Looping over a List—**for**" in subpart "D. Control Commands").

The **shift** command shifts arguments to the left; i.e., the value of **$1** is thrown away, **$2** replaces **$1**, **$3** replaces **$2**, etc.. The highest-numbered positional parameter becomes *unset*. (**$0** is *never* **shifted**.) The command **shift *n*** is a shorthand notation for ***n*** consecutive **shifts**. A **shift 0** does nothing. For example, consider the **shell** procedure **ripple** below. The **echo** command writes its arguments to the standard output. The **while** command is discussed in "Conditional Looping—**while** and **until**" in subpart "D. Control Commands". The lines that begin with # are comments.

```
#           ripple command
while test $# != 0
do
            echo $1 $2 $3 $4 $5 $6 $7 $8 $9
            shift
done
```

If the procedure were invoked by

```
ripple a b c
```

it would print

```
a b c
b c
c
```

The notation **$*** causes substitution of *all* positional parameters except **$0**. Thus, the **echo** line in the **ripple** example above could be written more compactly as:

```
echo $*
```

These two **echo** commands are *not* equivalent. The first prints at most nine positional parameters. The second prints *all* of the current positional parameters. The **$*** notation is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command such as the **nroff** text formatter:

```
nroff −h −rW120 −T450 −cm $*
```

It is important to understand the sequence of actions used by the **shell** in scanning command lines and substituting arguments. The **shell** first reads input up to a new-line or semicolon and then parses that much of the input. Variables are replaced by their values and then command substitution (via *grave accents*) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next the **shell** scans the resulting command line for *internal field separators,* that is, for any characters specified by *IFS* to break the command line into distinct arguments. Any *explicit* null arguments (specified by "" or ' ') are retained, while *implicit* null arguments resulting from evaluation of variables that are null or not set are removed. Then file name generation occurs, with all metacharacters being expanded. The resulting command line is executed by the **shell**.

Sometimes, one builds command lines inside a **shell** procedure. In this case one might want to have the **shell** rescan the command line after all the initial substitutions and expansions are done. The special command **eval** is available for this purpose. The **eval** command takes a command line as its argument and simply rescans the line performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=' | wc −l'
eval $command $output
```

This segment of code results in the pipeline **who | wc −l** being executed.

The output of **eval** cannot be redirected. The uses of **eval** can, however, be nested.

D. Control Commands

The **shell** provides several flow-of-control commands that are useful in creating **shell** procedures. To explain them, we first need a few definitions.

A ***simple command*** is defined in "A. Commands". Input/Output redirection arguments can appear in a simple command line and are passed to the **shell**, *not* to the command.

A ***command*** is a simple command or any of the **shell** control commands described below. A ***pipeline*** is a sequence of one or more commands separated by |. (For historical reasons, ^ is a synonym for | in this context.) The standard output of each command but the last in a pipeline is connected [by a **pipe(2)**] to the standard input of the next command. Each command in a pipeline is run separately. The **shell** waits for the last command to finish. The exit status of a pipeline is nonzero if the exit status of either the first or last process in the pipeline is nonzero. (This is a bit weird and may be changed in the future.)

A ***command list*** is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. A semicolon (;) causes sequential execution of the previous pipeline (i.e., the **shell** waits for the pipeline to finish before reading the next pipeline), while & causes asynchronous execution of the preceding pipeline. Both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily or until its processes are killed. Figure 2.2 shows the actions of the **shell** involved in executing these two command lists:

```
who >log; date
who >log& date&
```

For the first command list in Fig. 2.2, the **shell** executes **who**, waits for it to terminate, then executes **date** and waits for it to terminate. For the second command list in Fig. 2.2, the **shell** invokes both commands in order but does not wait for either one to finish.

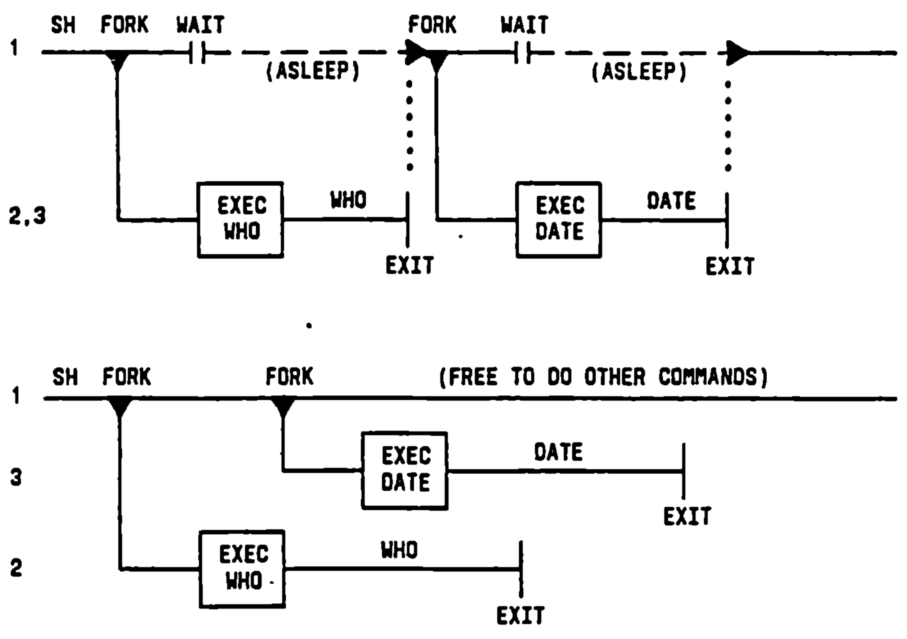

Fig. 2.2—The Shell Executing Typical Command Lists

More typical uses of & include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with & is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The **nohup** command is used for this purpose. Without **nohup**, if you hang up while **cc** in the above example is still executing, **cc** will be killed and your output will disappear.

Note: The & operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of simultaneous, asynchronous processes without a compelling reason for doing so.

The && and || operators, which are of equal precedence (but lower than & and |), cause conditional execution of pipelines. In **cmd1 || cmd2**, **cmd1** is executed and its exit status examined. Only if **cmd1** fails (i.e., has a nonzero exit status) is **cmd2** executed. This is thus a more terse notation for:

```
if    cmd1
          test $? != 0
then
      cmd2
fi
```

See **writemail** in part "EXAMPLES OF SHELL PROCEDURES" for an example of use of ||.

The && operator yields the complementary test: in **cmd1 && cmd2**, the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints *two* separate documents in a way similar to that shown in a previous example (see "G. Command Lines and Pipelines"):

```
{ nroff −cm text1; nroff −cm text2; } | col | greek −Thp
```

See "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands" for further details on command grouping.

All of the following commands are formally described in **sh**(1).

Structured Conditional—"if": The **shell** provides an **if** command. The simplest form of the **if** command is:

```
if command list
then
      command list
fi
```

The *command list* following **if** is executed. If the last command in this list has a *zero* exit status, then the *command list* that follows **then** is executed. The **fi** indicates the end of the **if** command.

In order to cause an alternative set of commands to be executed in the case where the *command list* following **if** has a *nonzero* exit status, one may add an **else** clause to the form given above. This results in the following structure:

```
if command list
then
      command list
else
      command list
fi
```

Multiple tests can be achieved in an **if** command by using the **elif** clause. For example:

```
if test −f " $1"             # is $1 a file?
then
                 pr $1
elif test −d " $1"           # else, is $1 a directory?
then
                 (cd $1; pr *)
else
                 echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows. If the value of the first positional parameter is a file name, then print that file. If not, then check to see if it is the name of a directory. If so, change to that directory and print all the files there. Otherwise, **echo** the error message.

The **if** command may be nested (but be sure to end each one with an **fi**). The new-lines in the above examples of **if** may be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Multiway Branch—"case": A multiple way branch is provided by the **case** command. The basic format of **case** is:

```
case string in
    pattern) command list ;;
        .
        .
        .
    pattern) command list ;;
esac
```

The **shell** tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in file-name generation ("C. Generation of Argument Lists"). If a match is found, the *command list* following the matched pattern is executed. The ;; serves as a break out of the **case** and is required after each command list except the last. Note that only one pattern is ever matched and that matches are attempted in order, so that if * is the first pattern in as **case**, no other patterns will ever be looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by |. For example:

```
case $i in
        *.c)            cc $i
                        ;;
        *.h|*.sh)       # do nothing
                        ;;
        *)              echo " $i of unknown type"
                        ;;
esac
```

In the above example, no action is taken for the second set of patterns because the *null* command is specified. The * is used as a default pattern because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands were executed, then **case** has a zero exit status.

Conditional Looping—"while" and "until": A **while** command has the general form:

```
while command list
do
        command list
done
```

The commands in the first *command list* are executed; and if the exit status of the last command in that list is zero, then the commands in the second list are executed. This sequence is repeated as long as the exit status of the first *command list* is zero. A loop can be executed as long as the first *command list* returns a nonzero exit status by replacing **while** with **until**.

Any new-line in the above example may be replaced by a semicolon. The exit status of a **while** (**until**) command is the exit status of the last command executed in the *second* command list. If no such command is executed, **while** (**until**) has exit status zero.

Looping over a List—"for": Often, one wishes to perform some set of operations for each in a set of files or execute some command once for each of several arguments. The **for** command can be used to accomplish this.

The **for** command has the format:

```
for variable in word list
do
          command list
done
```

where *word list* is a list of strings separated by blanks. The commands in the *command list* are executed once for each word in *word list*. *Variable* takes on as its value each word from *word list*; in turn, *word list* is fixed after it is evaluated the first time. For example, the following **for** loop will cause each of the C source files **xec.c**, **cmd.c**, and **word.c** in the current directory to be **diffed** with a file of the same name in the directory */usr/src/cmd/sh*:

```
for cfile in xec cmd word
do
            diff $cfile.c /usr/src/cmd/sh/$cfile.c
done
```

One can omit the "**in** *word list*" part of a **for** command. This will cause the current set of positional parameters to be used in place of *word list*. This is very convenient when one wishes to write a command that performs the same set of commands for each of an unknown number of arguments. See **null** in part "Examples of Shell Procedures" for an example of this feature.

Loop Control—"break" and "continue": The **break** command can be used to terminate execution of a **while**, **until**, or a **for** loop. The **continue** command requests the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break** *n*.

The **continue** command causes execution to resume at the nearest enclosing **while**, **until**, or **for**, i.e., the one that begins the innermost loop containing the **continue**. One can also specify an argument *n* to **continue** and execution will resume at the *nth* enclosing loop:

```
# This procedure is interactive; 'break' and 'continue'
# commands are used to allow the user to control data entry.
while true
do
            echo " Please enter data"
            read response
            case " $response"  in
                    " done" )          break          # no more data
                                       ;;
                    "" )               continue
                                       ;;
                    *)
                                       process the data here
                                       ;;
            esac
done
```

End-of-file and "exit": When the **shell** reaches end-of-file, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The **exit** command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated "normally" by using **exit 0**.

Command Grouping—Parentheses/Braces: There are two methods for grouping commands in the **shell**. As mentioned in "Cd" in subpart "I. Changing the State of the Shell and the *.profile File*", parentheses

() cause the **shell** to spawn a *subshell* that reads the enclosed commands. Both the right and left parentheses are recognized *wherever* they appear in a command line. The left and right parentheses can appear as literal parentheses *only* by being quoted. For example, if you type **garble(stuff)**, the **shell** interprets this as four separate words: **garble**, **(**, **stuff**, and **)**.

This subshell capability is useful if one wishes to perform some operations without affecting the values of variables in the current **shell** or to temporarily change directory and execute some commands in the new directory without having to explicitly return to the current directory. The current environment is passed to the subshell and variables that are **exported** in the current **shell** are also **exported** in the subshell. Thus:

```
current=`pwd`; cd /usr/docs/sh_tut;
nohup mm −Tlp sc_? | lpr& cd $current
```

and

```
(cd /usr/docs/sh_tut; nohup mm −Tlp sc_? | lpr&)
```

accomplish the same result. Both examples are used to print a copy of a document on the line printer. However, the second example automatically puts you back in your original working directory. In the second example above, blanks or new-lines surrounding the parentheses are allowed but not necessary. The **shell** will prompt with $*PS2* is expected. See also the example in "**Cd**" in subpart "I. Changing the State of the Shell and the *.profile* File".

Braces {} may also be used to group commands together. See "User-defined Variables" in subpart "D. Shell Variables" and "G. Conditional Substitution" for other meanings of braces in the **shell**. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace { may be followed by a new-line (in which case the **shell** will prompt for more input). Unlike the case involving parentheses, no subshell is spawned for braces. The enclosed commands are simply read by the **shell**. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command. See the last example in "D. Control Commands".

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

I/O Redirection and Control Commands: The **shell** normally does not *fork* when it recognizes the *control* commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input (output) to (from) each command. Also, when redirection of input/output is specified explicitly for a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, or **for** is used in a pipeline consisting of more than one command, the **shell** **forks** and a subshell runs the control command. This has certain implications. The most noticeable one is that *any changes made to variables within the control command are not effective once that control command finishes* (similar to the effect of using parentheses to group commands). The control commands run slightly slower when redirection is specified.

Beginners should skip to "E. Special Shell Commands" on first reading.

In-line Input Documents: Upon seeing a command line of the form:

command << *eofstring*

where *eofstring* is any arbitrary string, the **shell** will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring* (possibly preceded by one or more tab characters). By appending a minus (−) to <<, leading tab characters are deleted from each line of the input document before the **shell** passes the line to *command.*

The **shell** creates a temporary file containing the input document and performs variable and command substitution ("Command Substitution" in subpart "D. Shell Variables") on its contents before passing it to the command. Pattern matching on file names is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, one may quote any character of *eofstring*:

```
command << \eofstring
```

Typically *eofstring* consists of a single chararcter like ! which is often used for this purpose.

The in-line input document feature is especially useful for small amounts of input data (e.g., an editor "script"), where it is more convenient to place the data in the **shell** procedure than to keep it in a separate file. For instance, one could type:

```
cat <<- xyz
		This message will be printed on the
		terminal with leading tabs removed.
xyz
```

This in-line input document feature is most useful in **shell** procedures. See **edfind**, **edlast**, and **mmt** in part "EXAMPLES OF SHELL PROCEDURES". Note that in-line input documents may *not* appear within grave accents. This is an implementation bug that may be changed in the future.

Transfer to Another File and Back via Dot (.): A command line of the form

```
. proc
```

causes the **shell** to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the *dot* command finishes. This is thus a good way to gather a number of **shell** variable initializations into one file. Note that an **exit** command in a file executed in this manner will cause an exit from your current **shell**. If you are at login level, you will be logged out.

Interrupt Handling—"trap": As noted in "B. UNIX System Processes", a program may choose to *catch* an interrupt from the terminal, *ignore* it completely, or be terminated by it. Shell procedures can use the **trap** command to obtain the same effects.

```
trap arg signal-list
```

is the form of the **trap** command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers [as described in **signal**(2)]. The commands in *arg* are scanned at least once when the **shell** first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The single quotes inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the **shell**. The following procedure will print the name of the current directory on the file **errdirect** when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
            commands to be executed in directory $i here
done
```

while the same procedure with double (rather than single) quotes **trap** "echo`pwd`>errdirect" **2 3 15** will, instead, print the name of the directory from which the procedure was executed.

Signal 11 (SEGMENTATION VIOLATION) may never be trapped because the **shell** itself needs to catch it to deal with memory allocation. Zero is not a UNIX system signal but is effectively interpreted by the **trap** command as a signal generated by exiting from a **shell** (either via an **exit** command or by "falling through" the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string ('' or " "), then the signals in *signal-list* are *ignored* by the **shell**.

The most frequent use of **trap** is to assure removal of temporary files upon termination of a procedure. The second example of "Predefined Special Variables" in subpart "D. Shell Variables" would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
        commands, some of which use $temp, go here
```

In this example whenever signals 1 (HANGUP), 2 (INTERRUPT), 3 (QUIT), or 15 (SOFTWARE TERMINATION) are received by the shell procedure or whenever the **shell** procedure is about to exit, the commands enclosed between the single quotes will be executed. The **exit** command must be included or else the **shell** continues reading commands where it left off when the signal was received. The **trap 0** turns off the original trap on exits from the **shell** so that the **exit** command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the **shell** continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, thus terminating the **while** loop and restarting the cycle for the next directory. The entire procedure is terminated if interrupted when waiting for input; but during the execution of a command, an interrupt terminates *only* that command:

```
dir=`pwd`
for i in *
do
        if test -d $dir/$i
        then
                cd $dir/$i
                while echo " $i:"
                        trap exit 2
                        read x
                do
                        trap : 2        # ignore interrupts
                        eval $x
                done
        fi
done
```

Several **traps** may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the **shell** implements the **trap** command in order not to be surprised. When a signal (other than 11) is received by the **shell**, it is passed on to whatever child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the **shell** *then* polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward except in the case of traps set at the command (outermost or login) level. In this case, it is possible that no child process is running, so the **shell** waits for the termination of the first process spawned *after* the signal is received before it polls the traps.

For internal commands, the **shell** normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

E. Special Shell Commands

There are several special commands that are *internal* to the shell (some of which have already been mentioned). These commands should be used in preference to other UNIX system commands whenever possible because they are faster and more efficient. The **shell** does not fork to execute these commands, so no additional processes are spawned. The trade-off for this efficiency is that redirection of input/output is not allowed for most of these special commands.

Several of the special commands have already been described in "D. Control Commands" because they affect the flow of control. They are **break, continue, exit,** dot (.), and **trap**. The **set** command described in "Positional Parameters" in subpart "D. Shell Variables" and "Execution Flags—**set**" in subpart "I. Changing the State of the Shell and the *.profile* File" is also a special command. Descriptions of the remaining special commands [see **sh**(1)] are given here:

:	The **null** command does nothing. The exit status is zero (*true*). *Beware:* any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place just as in other commands.
cd ***arg***	Make *arg* the current directory. If *arg* does not begin with /, ./, or ../, **cd** uses the *CDPATH* **shell** variable ("User-defined Variables" in subpart "D. Shell Variables") to locate a parent directory that contains the directory *arg*. If *arg* is not a directory or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *arg* is equivalent to typing **cd $HOME**.
exec ***arg*** ...	If *arg* is a command, then the **shell** executes it without forking. No new process is created. Input/output redirection arguments *are* allowed on the command line. If *only* input/output redirection arguments appear, then the input/output of the **shell** itself is modified accordingly. See **merge** in part "Examples of Shell Procedures" for an example of this use of **exec**.
newgrp ***arg*** ...	The **newgrp(1)** command is executed replacing the **shell**. The **newgrp** command in turn spawns a new **shell**. *Beware:* Only variables in the environment will be known in the shell that is spawned by the **newgrp** command. Any variables that were **exported** will no longer be marked as such.
read ***var*** ...	One line (up to a new-line) is read from standard input and the first word is assigned to the first variable, the second word to the second variable, etc. All leftover words are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.
readonly ***var*** ...	The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all **exported** variables is given.
test	A conditional expression is evaluated. More details are given in "A. Conditional Evaluation—**test**".

times	The accumulated user and system times for processes run from the current **shell** are printed.
umask ***nnn***	The user file creation mask is set to *nnn*. See **umask**(2) for details. If *nnn* is omitted, then the current value of the mask is printed.
ulimit ***n***	This command imposes a limit of *n* blocks on the size of files written by the **shell** and its child processes (files of any size may be read). If *n* is omitted, the current value of this limit is printed. The default value for *n* varies from one installation to another.
wait ***n***	The **shell** waits for the child process whose process number is *n* to terminate. The exit status of the **wait** command is that of the process waited on. If *n* is omitted or is not a child of the current **shell**, then *all* currently active processes are waited for and the return code of the **wait** command is zero.

F. Creation and Organization of Shell Procedures

A **shell** procedure can be created in two simple steps:

1. Build an ordinary text file.

2. Change the file's *mode* to make it *executable*.

Changing the *mode* allows a shell procedure to be invoked by *proc args* rather than by **sh** *proc args*. The second step may be omitted for a procedure to be used once or twice and then discarded but is recommended for longer-lived ones. Here is the entire input needed to set up a simple procedure (the executable part of **draft** in part "Examples of Shell Procedures"):

```
ed
a
nroff –rC3 –T450-12 –cm $*
.
w draft
q
chmod +x draft
```

It may then be invoked as **draft file1 file2**. Note that **shell** procedures must always be at least readable so that the **shell** itself can read commands from the file.

If **draft** were thus created in a directory whose name appears in the user's *PATH* variable, the user could change working directories and still invoke the **draft** command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the **shell** to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current **shell** read commands from the new file, allowing use of existing **shell** variables, and avoiding the spawning of an additional process for another **shell**.

Many users prefer to write **shell** procedures instead of C programs. First, it is easy to create and maintain a **shell** procedure because it is only a file of ordinary text. Second, it has no corresponding object program that must be generated and maintained. Third, it is easy to create a procedure on the fly, use it a few times, and then remove it. Finally, because **shell** procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and/or shell procedures are usually named *bin*. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures.

Some users have their *PATH* variable list several such directories. Although you can have a number of suc directories, it is unwise to go overboard. It may become difficult to keep track of your environment, and efficiency may suffer (see "C. Efficient Organization").

G. More about Execution Flags

There are several execution flags available in the **shell** that can be useful in **shell** procedures:

−e	The **shell** will exit immediately if any command that it executes exits with a nonzero exit status.
−u	When this flag is set, the **shell** treats the use of an unset variable as an error. This flag can be used to perform a global check on variables.
−t	The **shell** exits after reading and executing the commands on the remainder of the current input line.
−n	This is a *don't execute* flag. On occasion, one may want to check a procedure for syntax errors but not to execute the commands in the procedure. Writing **set −nv** at the beginning of the file will accomplish this.
−k	*All* arguments of the form *variable=value* are treated as keyword parameters. When this flag is *not* set, only such arguments that appear *before* the command name are treated as keyword parameters.

MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES

Shell procedures can make use of any UNIX system command. The commands described in this part are either used especially frequently in **shell** procedures or are explicitly designed for such use. More detailed descriptions of each of these commands can be found in Section 1 of the UNIX System User's Manual.

A. Conditional Evaluation—"test"

The **test**(1) command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. The **test** command also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the *command list* following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotes if there is any chance of their being null or not set.

On some UNIX systems, the square brackets ([]) may be used as an alias for **test**; e.g., [*expression*] has the same effect as **test** *expression*.

The following is a partial list of the primaries that can be used to construct a conditional expression:

−r ***file***	*true* if the named file exists and is readable by the user.
−w ***file***	*true* if the named file exists and is writable by the user.
−x ***file***	*true* if the named file exists and is executable by the user.
−s ***file***	*true* if the named file exists and has a size greater than zero.
−d ***file***	*true* if the named file exists and is a directory.
−f ***file***	*true* if the named file exists and is an ordinary file.

−p *file*	*true* if the named file exists and is a named pipe (*fifo*).
−z *s1*	*true* if the length of string " s1" is zero.
−n *s1*	*true* if the length of the string " s1" is nonzero.
−t *fildes*	*true* if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
s1 = *s2*	*true* if strings " s1" and " s2" are identical.
s1 != *s2*	*true* if strings " s1" and " s2" are *not* identical.
s1	*true* if " s1" is *not* the null string.
n1 −eq *n2*	*true* if the integers *n1* and *n2* are algebraically equal. Other algebraic comparisons are indicated by **−ne, −gt, −ge, −lt,** and **−le.**

These primaries may be combined with the following operators:

!	unary negation operator.
−a	binary logical *and* operator.
−o	binary logical *or* operator. The **−o** has lower precedence than **−a**.
(***expr***)	parentheses for grouping; they must be escaped to remove their significance to the **shell**. When parentheses are absent the evaluation proceeds from left to right.

Note that all primaries, operators, file names, etc. are separate arguments to **test**.

B. Reading a Line—"line"

The **line(1)** command takes one line from standard input and prints it on standard output. This is useful when you need to read a line from a file or capture the line in a variable. The functions of **line** and of the **read** command that is internal to the **shell** differ in that input/output redirection is possible only with **line**. If the user does not require input/output redirection, **read** is faster and more efficient. An example of a usage of **line** for which **read** would not suffice is:

```
firstline=`line < somefile`
```

C. Simple Output—"echo"

The **echo(1)** command, invoked as **echo** [*arg* ...], copies its arguments to the standard output, each followed by a single space except for the last argument which is normally followed by a new-line. Often, **echo** is used to prompt the user for input to issue diagnostics in **shell** procedures or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command **ls** is often replaced by **echo *** because the latter is faster and prints fewer lines of output.

The **echo** command recognizes several escape sequences. A **\n** yields a new-line character. A **\c** removes the new-line from the end of the echoed line. The following prompts the user, allowing one to type on the same line as the prompt:

```
echo 'enter name:\c'
read name
```

The **echo** command also recognizes octal escape sequences for *all* characters whether printable or not. An **echo** " \007" typed at a terminal will cause the bell on that terminal to ring.

D. Expression Evaluation—"expr"

The **expr(1)** command provides arithmetic and logical operations on integers and some pattern matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output. The **expr** command can be used inside grave accents to set a variable. Typical examples are:

```
#           increment $a
a=`expr $a + 1`
#           put third through last characters of
#           $1 into substring
substring=`expr " $1"  : '..\(.*\)'`
#           obtain length of $1
c=`expr " $1"  : '.*'`
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern matching capability to pick apart strings. See **expr(1)** for more details.

E. "true" and "false"

The **true(1)** and **false** [see **true(1)**] commands perform the obvious functions of exiting with zero and non-zero exit status, respectively. The **true** command is often used to implement an unconditional loop.

F. Input/Output Redirection Using File Descriptors

Beginners should skip this subpart on first reading. A command occasionally directs output to some file associated with a file descriptor other than 1 or 2 (see "Diagnostic and Other Outputs" in subpart "F. Redirection of Input and Standard Output"). In languages such as C, one can associate output with *any* file descriptor by using the **write(2)** system call. The **shell** provides its own mechanism for creating an output file associated with a particular file descriptor. By typing

fd1>&*fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* onto the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at execution time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing

command 2>&1

If one wanted to redirect both standard output and standard error output to the same file, one would type

command 1> *file* 2>&1

The order here is significant. First, file descriptor 1 is associated with *file*. Then file descriptor 2 is associated with the same file that is *currently* associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file* because at the time of the error output redirection file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard *input*. One could type

fda<&*fdb*

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for commands that use two or more input sources. Another use of this notation is for sequential reading and processing of a file. See **merge** in part "EXAMPLES OF SHELL PROCEDURES" for an example of use of this feature.

G. Conditional Substitution

Normally, the **shell** replaces occurrences of **$***variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution depending upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string which may be assigned to a variable in any one of the following ways:

```
A=
bcd=""
Ef_g=''
set '' ""
```

The first three of these examples assign the null string to each of the corresponding *shell variables*. The last example sets the first and second *positional parameters* to the null string and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null*. (Note that, in these expressions, *variable* refers to either a digit or a variable name and the meaning of braces differs from that described in "User-defined Variables" in subpart "D. Shell Variables" and "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands".)

${*variable* :−*string*}

If *variable* is set and is non-null, then substitute the value **$***variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

${*variable* :=*string*}

If *variable* is set and is non-null, then substitute the value **$***variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value **$***variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

${*variable* :?*string*}

If *variable* is set and is non-null, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable : *string*

and exit from the current **shell**. (If the **shell** is the login **shell**, it is not exited.) If *string* is omitted in this form, then the message

variable : parameter null or not set

is printed instead.

${*variable* :+*string*}

If *variable* is set and is non-null, then substitute *string* for this expression; otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon (:), in which case the **shell** does *not* check whether *variable* is null or not. It only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If *PATH* has ever been set and is not null, then keep its current value. Otherwise, set it to the string *:/bin:/usr/bin.* Note that one needs an explicit assignment to set *PATH* in this form:

```
PATH=${PATH:-':/bin:/usr/bin'}
```

2. If *HOME* is set and is not null, then change directory to it; otherwise, set it to the given value and change directory to it. Note that *HOME* is automatically assigned a value in this case:

```
cd ${HOME:='/usr/gas'}
```

H. Invocation Flags

There are four flags that may be specified on the command line invoking the **shell**. These flags may *not* be turned on via the **set** command:

−i If this flag is specified or if the **shell**'s input and output are both attached to a terminal, the **shell** is *interactive.* In such a **shell**, INTERRUPT (signal 2) is caught and ignored, while QUIT (signal 3) and SOFTWARE TERMINATION (signal 15) are ignored.

−s If this flag is specified or if no input/output redirection arguments are given, the **shell** reads commands from standard input. Shell output is written to file descriptor 2. The **shell** you get upon logging into the system effectively has the **−s** flag turned on.

−c When this flag is turned on, the **shell** reads commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multiword string in order to allow for variable substitution.

−r When this flag is specified on invocation, then the *restricted shell* is invoked. This is a version of the **shell** in which certain actions are disallowed. In particular, the **cd** command produces an error message, and the user cannot set *PATH.* See **sh**(1) for a more detailed description.

EXAMPLES OF SHELL PROCEDURES

Some examples in this subpart are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.

copypairs

```
#       usage: copypairs file1 file2 ...
#       copy file1 to file2, file3 to file4, ...
while test " $2" != ""
do
          cp $1 $2
          shift; shift
          done
          if test " $1" != ""
          then
                echo " $0: odd number of arguments"
          fi
```

Note: This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop because you can adjust the positional parameters via **shift** to handle related arguments.

copyto

```
#           usage: copyto dir file ...
#           copy argument files to 'dir', making sure that at least
#           two arguments exist and that 'dir' is a directory
if test $# -lt 2
then
                echo " $0: usage: copyto directory file ..."
elif test ! -d $1
then
                echo " $0: $1 is not a directory";
else
                dir=$1; shift
          for eachfile
          do
                    cp $eachfile $dir
          done
fi
```

Note: This procedure uses an **if** command with two tests in order to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first. The original **$1** is shifted off.

distinct

```
#           usage: distinct
#           reads standard input and reports list of alphanumeric strings
#           that differ only in case, giving lowercase form of each
tr -cs '[A-Z][a-z][0-9]' '[\012*]' | sort -u |
          tr '[A-Z]' '[a-z]' | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. It may not be immediately obvious how this works. [See **tr**(1), **sort**(1), and **uniq**(1) if you are completely unfamiliar with these commands.] The **tr** translates all characters except letters and digits into new-line characters and then squeezes out repeated new-line characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The **uniq -d** prints (once) only those lines that occur more than once yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged. The two lines below are equivalent assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3; rm temp[12]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic **distinct** with such a step-by-step process using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc.abc
```

Although pipelines can give a concise notation for complex processes exercise some restraint lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

draft

```
#         usage: draft file(s)
#         prints the draft (−rC3) of a document on a DASI 450
#         terminal in 12-pitch using memorandum macros (MM).
nroff −rC3 −T450-12 −cm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

edfind

```
#         usage: edfind file arg
#         find the last occurrence in 'file' of a line whose
#         beginning matches 'arg', then print 3 lines (the one
#         before, the line itself, and the one after)
ed − $1 <<!
H          ——> Help command.
?^$2?;−,+p
!
```

Note: This procedure illustrates the practice of using editor (**ed**) in-line input scripts into which the **shell** can substitute the values of variables. It is a good idea to turn on the **H** option of **ed** when embedding an **ed** script in a **shell** procedure [see **ed(1)**].

edlast

```
#         usage: edlast file
#         prints the last line of file, then deletes that line
ed − $1 <<−\eof          # no variable substitutions in "ed" script
          H
          $p
          $d
          w
          q
 eof
 echo Done.
```

Note: This procedure contains an in-line input document or script (see "In-line Input Documents" in subpart "D. Control Commands"); it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, **eof**) of the input redirection. If this had not been done, **$p** and **$d** would have been treated as **shell** variables.

fsplit

```
#           usage: fsplit file1 file2
#           read standard input and divide it into three parts:
#           append any line containing at least one letter
#           to file1, any line containing at least one digit
#           but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
        total=" `expr $total + 1` "
        case " $next"     in
        *[A-Za-z]*)
                    echo   " $next"    >> $1 ;;
        *[0-9]*)
                    echo   " $next"    >> $2 ;;
        *)
                    lost=" `expr $lost + 1` "
        esac
done
echo "$total lines read, $lost thrown away"
```

Note: In this procedure, each iteration of the **while** loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file.

Do not use the **shell** to read a line at a time unless you must—it can be grotesquely slow (see "Number of Processes Generated" in subpart "B. Approximate Measures of Resource Consumption").

initvars

```
#           usage: . initvars
#           use carriage return to indicate " no change"
echo " initializations? \c"
read response
if test " $response"  = y
then
        echo " PS1=\c" ; read temp
                PS1=${temp:-$PS1}
        echo " PS2=\c " ; read temp
                PS2=${temp:-$PS2}
        echo " PATH=\c " ; read temp
                PATH=${temp:-$PATH}
        echo " TERM=\c "  ; read temp
                TERM=${temp:-$TERM}
fi
```

Note: This procedure would be invoked by a user at the terminal or as part of a *.profile* file. The assignments are effective even when the procedure is finished because the **dot** command is used to invoke it. To better understand the **dot** command invoke **initvars** as indicated above and check the values of *PS1*, *PS2*, *PATH*, and *TERM*; then make **initvars** executable, type **initvars**, assigning different values to the three variables, and check again the values of these three **shell** variables after **initvars** terminates. It is assumed that *PS1*, *PS2*, *PATH*, and *TERM* have been **exported**, presumably by your *.profile* (see "The *.profile* File" in subpart "I. Changing the State of the Shell and the *.profile* File" and "A. A Command's Environment").

merge

```
#           usage:     merge src1 src2 [ dest ]
#           merge two files, every other line.
#           the first argument starts off the merge,
#           excess lines of the longer file are appended to
#           the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}               # default destination file is named $1.m
while true
do
                             # alternate reading from the files;
                             # 'more' represents the file descriptor
                             # of the longer file
        line <&4 >>$dest  || { more=5; break ;}
        line <&5 >>$dest  || { more=4; break ;}
done
                             # delete the last line of destination
                             # file, because it is blank.
ed - $dest <<\eof
        H
        $d
        w
        q
eof
while line <&$more >> $dest
do :; done                  # read the remainder of the longer
                            # file—the body of the 'while' loop
                            # does nothing; the work of the loop
                            # is done in the command list following
                            # 'while'
```

Note: This procedure illustrates a technique for reading sequential lines from a file or files without creating any subshells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used **src1** and **src2** explicitly rather than the associated file descriptors, this procedure would never terminate because the *first* line of each file would be read over and over again.

mkfiles

```
#           usage: mkfiles pref [ quantity ]
#           makes 'quantity' (default = 5) files, named pref1, pref2, ...
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
        > $1$i
        i="`expr $i + 1`"
done
```

Note: This procedure uses input/output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop. Compare this procedure with procedure **null** below.

mmt

```
if test "$#" = 0; then cat <<\!
Usage: "mmt [ options ] files" where "options" are:
-a          => output to terminal
-e          => preprocess input with eqn
-t          => preprocess input with tbl
-Tst        => output to STARE phototypesetter
               manufactured by Honeywell
-T4014      => output to 4014 manufactured by Tektronix
-Tvp        => output to printer manufactured by Versatec
-           => use instead of "files" when mmt used
               inside a pipeline.
Other options as required by TROFF and the MM macros.
!
          exit 1
fi
PATH='/bin:/usr/bin'; O='-g'; o='| gcat -ph';
#                       Assumes typesetter is accessed via gcat(1)
#                       If typesetter is on-line, use O=''; o=''
while test -n "$1" -a ! -r "$1"
do
   case "$1" in
          -a)            O='-a';           o='';;
          -Tst)          O='-g';           o='|gcat -st';;
#                       Above line for STARE only
          -T4014)        O='-t';           o='|tc';;
          -Tvp)          O='-t';           o='|vpr -t';;
          -e)            e='eqn';;
          -t)            f='tbl';;
          -)             break;;
          *)             a="$a $1";;
   esac
   shift
done
if test -z "$1"
then
                         echo 'mmt: no input file'
                         exit 1
fi
if test "$O" = '-g'
then
                         x="-f$1"
fi
d="$*"
if test "$d" = '-'
then
                         shift
                         x=''
                         d=''
```

```
        fi
        if test -n "$f"
        then
                                        f=" tbl $* |"
                                        d=''
        fi
        if test -n "$e"
        then
                              if test -n " $f"
                                        then e='eqn | '
                                        else e=" eqn $* | "
                                        d=''
                              fi
        fi
        eval " $f $e troff $O -cm $a $d $0 $x" ;            exit 0
```

Note: This is a slightly simplified version of an actual UNIX system command (although this is *not* the version included in UNIX system Release 4.0). It uses many of the features available in the **shell**. If you can follow through it without getting lost, you have a good understanding of **shell** programming. Pay particular attention to the process of building a command line from **shell** variables and then using **eval** to execute it.

null

```
        #         usage: null file
        #         create each of the named files as an empty file
        for eachfile
        do
                  > $eachfile
        done
```

Note: This procedure uses the fact that output redirection creates the (empty) output file if that file does not already exist. Compare this procedure with procedure **mkfiles** above.

phone

```
        #         usage: phone initials
        #         prints the phone number(s) of person with given initials
        echo 'inits          ext          home'
        grep " ^$1" <<\!
        abc       1234        999-2345
        def       2234        583-2245
        ghi       3342        988-1010
        xyz       4567        555-1234
        !
```

Note: This procedure is an example of using an in-line input document or *script* to maintain a *small* data base.

writemail

```
        #         usage: writemail message user
        #         if user is logged in, write message on terminal;
        #          otherwise, mail it to user
        echo "$1"  | { write "$2"  || mail "$2" ;}
```

Note: This procedure illustrates command grouping. The message specified by **$1** is piped to the **write** command and, if **write** fails, to the **mail** command.

EFFECTIVE AND EFFICIENT SHELL PROGRAMMING

A. Overall Approach

This subpart outlines strategies for writing *efficient* shell procedures, i.e., ones that do not waste resources unreasonably in accomplishing their purposes. The primary reason for choosing the **shell** procedure as the implementation method is to achieve a desired result at a minimum *human* cost. Emphasis should *always* be placed on simplicity, clarity, and readability; but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size and often increases its comprehensibility. In any case, one should not worry about optimizing **shell** procedures unless they are intolerably slow or are known to consume a lot of resources.

The same kind of iteration cycle should be applied to **shell** procedures as to other programs—write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs even when the style of programming is a familiar one. Each timing test should be run several times because the results are easily disturbed by, for instance, variations in system load.

B. Approximate Measures of Resource Consumption

Number of Processes Generated: When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another **shell**.

If you are worried about efficiency, it is important to know which commands are currently built into the **shell** and which are not. Here is the alphabetical list of those that are built-in:

break	**case**	**cd**	**continue**	**eva**	**exec**
exit	**export**	**for**	**if**	**newgrp**	**read**
readonly	**set**	**shift**	**test**	**times**	**trap**
ulimit	**umask**	**until**	**wait**	**while**	**.**
:	**{...}**				

The (...) command executes as a child process, i.e., the **shell** does a **fork**, but no **exec**. Any command *not* in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a **shell** procedure. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by

$$\text{processes} = k*n + c$$

where *k* and *c* are constants, and *n* is the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero. Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of procedure **fsplit** of part "EXAMPLES OF SHELL PROCEDURES". For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo** is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n + 1. On the other hand, the number of processes in the following (equivalent) procedure is 12 regardless of the number of lines of input:

```
#          faster fsplit
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$                # read standard input into temp file
                            # save original lengths of $1, $2
if test -s "$1" ; then start1=`wc -l < $1`; fi
if test -s "$2" ; then start2=`wc -l < $2`; fi
grep "$b"  temp$$ >> $1             # lines with letters onto $1
grep -v "$b"  temp$$ | grep '[0-9]' >>$2
                            # lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \( $end1 - $start1 \) - \ ($end2 -$start2\)`"
echo "$total lines read, $lost thrown away"
```

This version is often ten times faster than **fsplit**, and it is even faster for larger input files.

Some types of procedures should *not* be written using the **shell**. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C.

Note: Shell procedures should not be used to scan or build files a character at a time.

Number of Data Bytes Accessed: It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. Which of the following is likely to be faster?

```
sort file | grep pattern
grep pattern file | sort
```

Directory Searches: Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of cd can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system):

```
time sh -c 'ls -l /usr/bin/* >/dev/null'
time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

If you do not understand exactly what is going on in these examples, read Section 7 in the UNIX System User's Manual.

C. Efficient Organization

Directory-Search Order and PATH Variable: The *PATH* variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion; or the result may be a great increase in system overhead that occurs in a subtle, but avoidable, way.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current *PATH* variable. As an example, consider the effect of invoking nroff (i.e., */usr/bin/nroff*) when *$PATH* is *:/bin:/usr/bin*. The sequence of directories read is: ., /, */bin*, /, */usr*, and */usr/bin*, i.e., a total of six directories. A long path list assigned to *PATH* can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless *PATH* setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (but *only* with respect to the efficiency of command searches):

```
:/al/tf/jtb/bin:/usr/lbin:/bin:/usr/bin
:/bin:/al/tf/jtb/bin:/usr/lbin:/usr/bin
:/bin:/usr/bin:/al/tf/jtb/bin:/usr/lbin
/bin::/usr/bin:/al/tf/jtb/bin:/usr/lbin
```

The first one above should be avoided. The others are acceptable; the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the *PATH* variable inside the procedure such that the fewest possible directories are searched in an optimum order. The **mmt** example in part "EXAMPLES OF SHELL PROCEDURES" does this.

Setting Up Directories: It is wise to avoid directories that are larger than necessary. You should be aware of several *magic* sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a *small* file. Anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink.

REFERENCES

[1]—Bianchi, M. H., and Wood, J. L. A User's Viewpoint on the Programmer's Workbench. Proc. Second Int. Conf. on Software Engineering, pp. 193-99 (Oct. 13-15, 1976).

[2]—Dolotta, T. A., Haight, R. C., and Mashey, J. R. The Programmer's Workbench. The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 2177-200 (July-Aug. 1978).

[3]—Dolotta, T. A., and Mashey, J. R. An Introduction to the Programmer's Workbench. Proc. Second Int. Conf. on Software Engineering, pp. 164-68 (Oct. 13-15, 1976).

[4]—Dolotta, T. A., and Mashey, J. R. Using a Command Language as the Primary Programming Tool. In: Beech, D. (ed.), Command Language Directions (Proc. of the Second IFIP Working Conf. on Command Languages), pp. 35-55. Amsterdam: North Holland (1980).

[5]—Kernighan, B. W., and Mashey, J. R. The UNIX Programming Environment. *COMPUTER*, Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in *Software-Practice & Experience*, Vol. 9, No. 1, pp. 1-15 (Jan. 1979).

[6]—Kernighan, B. W., and Plauger, P. J. Software Tools. Proc. First Nat. Conf. on Software Engineering, pp. 8-13 (Sept. 11-12, 1975).

[7]—Kernighan, B. W., and Plauger, P. J. Software Tools. Reading, MA: Addison-Wesley (1976).

[8]—Kernighan, B. W., and Ritchie, D. M. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall (1978).

[9]—Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905-29 (July-Aug. 1978).

[10]—Snyder, G. A., and Mashey, J. R. UNIX System Documentation Road Map. Bell Laboratories (January 1981).

3. THE C PROGRAMMING LANGUAGE

INTRODUCTION

This section describes C language as it is implemented and supported on the 3B20S Processor, the PDP*-11, the VAX*-11/780, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, this section concentrates on the PDP-11 but tries to point out implementation-dependent details. With few exceptions, such dependencies follow directly from the properties of the hardware. The various compilers are generally quite compatible. This section contains the following subsections:

- C LANGUAGE—A summary of the grammar and rules of the C programming language.
- LIBRARIES—Descriptions of functions and declarations that support C language and how to use these functions.
- THE "cc" COMMAND—The command used to compile C language programs, assemble assembly language programs, and produce executable programs is briefly described in terms of usage.
- A C PROGRAM CHECKER—"lint"—A program that attempts to detect bugs in C programs during compilation.
- A SYMBOLIC DEBUGGER—"sdb"—A symbolic debugging program that is used to debug compiled C language programs.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

*Trademarks of the Digital Equipment Corporation.

C LANGUAGE

LEXICAL CONVENTIONS

There are six classes of tokens—identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

A. Comments

The characters /* introduce a comment which terminates with the characters */. Comments do not nest.

B. Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

PDP-11	7 characters, 2 cases
VAX-11	7 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases
WECo 3B20	8 characters, 2 cases

C. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```
auto      do      float  register  switch
break     double  for    return    typedef
case      else    goto   short     union
char      entry   if     sizeof    unsigned
continue  enum    int    static    void
default   extern  long   struct    while
```

The **entry** keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words **fortran** and **asm** .

D. Constants

There are several kinds of constants. Some of the more important constants are integer, long, character, floating, and enumeration. Hardware characteristics that affect sizes are summarized in "F. Hardware Characteristics" under part "LEXICAL CONVENTIONS".

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer.

The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by l (letter ell) or L is a long constant. As discussed below, on some machines integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	\ddd

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is \0 (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double precision.

Enumeration Constants

Names declared as enumerators (see "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") are constants of the corresponding enumeration type. They behave like **int** constants.

E. Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of characters" and storage class static (see part "NAMES") and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used. Finally, a \ and the immediately following new-line are ignored.

F. Hardware Characteristics

The following table summarizes certain hardware properties that vary from machine to machine.

TABLE 3.A

HARDWARE CHARACTERISTICS

	DEC PDP-11 ASCII	DEC VAX-11 ASCII	HONEYWELL 6000 ASCII	IBM 370 EBCDIC	INTERDATA 8/32 ASCII	WECO 3B ASCII
char	8 bits	8 bits	9 bits	8 bits	8 bits	8 bits
int	16	32	36	32	32	32
short	16	16	36	16	16	16
long	32	32	36	32	32	32
float	32	32	36	32	32	32
double	64	64	72	64	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 308}$

SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in part "SYNTAX SUMMARY".

NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier—its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes:

- automatic
- static
- external
- register.

Automatic variables are local to each invocation of a block (see "B. Compound Statement or Block" in part "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

Each enumeration (see "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") is conceptually a separate type with its own set of named constants. The properties of an **enum** type are identical to those of **int** type.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char**, **int** of all sizes, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The void type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *arrays* of objects of most types
- *functions* which return objects of a given type
- *pointers* to objects of a given type
- *structures* containing a sequence of objects of various types
- *unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

OBJECTS AND LVALUES

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name "lvalue" comes from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "F. Arithmetic Conversions". The summary will be supplemented as required by the discussion of each operator.

A. Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from −128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all positive. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value −1.

When a longer integer is converted to a shorter or to a **char**, it is truncated on the left. Excess bits are simply discarded.

B. Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length.

C. Floating and Integral

Conversions of floating values to integral type tend to be rather machine dependent. In particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

D. Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

E. Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

F. Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

- First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.

- Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.
- Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
- Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
- Otherwise, both operands must be **int**, and that is the type of the result.

G. Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "A. Expression Statement" in part "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" in part "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "D. Additive Operators") are those expressions defined under "A. Primary Expressions", "B. Unary Operators", and "C. Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of part "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine dependent. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

A. Primary Expressions

Primary expressions involving **.**, **->**, subscripting, and function calls group left to right.

primary-expression:
 identifier
 constant
 string
 (expression)
 primary-expression [*expression*]
 primary-expression (expression-list$_{opt}$ *)*
 primary-expression . identifier
 primary-expression -> identifier

expression-list:
 expression
 expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants are **double**.

A string is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see "F. Initialization" in part "DECLARATIONS".)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "B. Unary Operators" and "D. Additive Operators" on identifiers, ***** and **+**, respectively. The implications are summarized under "C. Arrays, Pointers, and Subscripting" in part "TYPES REVISITED".

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "B. Unary Operators" and "G. Type Names" in part "DECLARATIONS".

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from – and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS".

B. Unary Operators

Expressions with unary operators group right to left.

unary-expression:
> ** expression*
> *& lvalue*
> *− expression*
> *! expression*
> *˜ expression*
> *++ lvalue*
> *−− lvalue*
> *lvalue ++*
> *lvalue −−*
> *(type-name) expression*
> *sizeof expression*
> *sizeof (type-name)*

The unary * operator means *indirection:* the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where *n* is the number of bits in an **int**. There is no unary + operator.

The result of the logical negation operator ! is 1 if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ˜ operator yields the 1's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x+=1**. See the discussions "D. Additive Operators" and "N. Assignment Operators" for information on conversions.

The lvalue operand of prefix −− is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix −− is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix −− operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described under "G. Type Names" in part "Declarations".

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***)−2** is the same as **(sizeof(***type***))−2**.

C. Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
    *expression * expression*
    expression / expression
    expression % expression

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary / operator indicates division. When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be floating.

D. Additive Operators

The additive operators + and − group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
    expression + expression
    expression − expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the − operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

E. Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
    expression << expression
    expression >> expression

The value of E1<<E2 is E1 (interpreted as a bit pattern) left-shifted E2 bits. Vacated bits are 0 filled. The value of E1>>E2 is E1 right-shifted E2 bit positions. The right shift is guaranteed to be logical (0 fill) if E1 is **unsigned**; otherwise, it may be arithmetic (fill by a copy of the sign bit).

F. Relational Operators

The relational operators group left to right. This fact is not very useful; a<b<c does not mean what it seems to.

relational-expression:
    expression < expression
    expression > expression
    expression <= expression
    expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

G. Equality Operators

equality-expression:
    expression == expression
    expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

H. Bitwise AND Operator

and-expression:
    expression & expression

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

I. Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ expression

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

J. Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | expression

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

K. Logical AND Operator

logical-and-expression:
expression && expression

The && operator groups left to right. It returns 1 if both its operands are nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

L. Logical OR Operator

logical-or-expression:
expression || expression

The || operator groups left to right. It returns 1 if either of its operands is nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

M. Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

N. Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue −= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; and it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form E1 *op* = E2 may be inferred by taking it as equivalent to E1 = E1 *op* (E2); however, E1 is evaluated only once. In += and −=, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "D. Additive Operators". All right operands and all nonpointer left operands must have arithmetic type.

O. Comma Operator

comma-expression:
expression , expression

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions ("A. Primary Expressions") and lists of initializers ("F. Initialization" in part "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$ *;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

> *decl-specifiers:*
> *type-specifier decl-specifiers*$_{opt}$
> *sc-specifier decl-specifiers*$_{opt}$

The list must be self-consistent in a way described below.

A. Storage Class Specifiers

The sc-specifiers are:

> *sc-specifier:*
> **auto**
> **static**
> **extern**
> **register**
> **typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "H. Typedef" for more information. The meanings of the various storage classes were discussed in part "Names".

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see part "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

B. Type Specifiers

The type-specifiers are

> *type-specifier:*
> **char**
> **short**
> **int**
> **long**
> **unsigned**
> **float**
> **double**
> **void**
> *struct-or-union-specifier*
> *typedef-name*
> *enum-specifier*

The words **long**, **short**, and **unsigned** may be thought of as adjectives. The following combinations are acceptable.

```
short int
long int
unsigned int
unsigned char
long float
```

The meaning of the last is the same as **double**. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "E. Structure, Union, and Enumeration Declarations". Declarations with **typedef** names are discussed in "H. Typedef".

C. Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
 init-declarator
 init-declarator , declarator-list

init-declarator:
 declarator initializer$_{opt}$

Initializers are discussed in "F. Initialization". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
 identifier
 (declarator)
 ** declarator*
 declarator ()
 declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

D. Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... T," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

```
*D
```

the type of the contained identifier is "... pointer to T."

If D1 has the form

```
D()
```

then the contained identifier has the type "... function returning T."

If D1 has the form

D[*constant-expression*]

or

```
D[]
```

then the contained identifier has type "... array of T." In the first case, the constant expression is an expression whose value is determinable at compile time and whose type is **int**. (Constant expressions are defined precisely in part "Constant Expressions".) When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers to such things; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, ×3d is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array," the last has type **int**.

E. Structure, Union, and Enumeration Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

> *struct-or-union-specifier:*
> *struct-or-union { struct-decl-list }*
> *struct-or-union identifier { struct-decl-list }*
> *struct-or-union identifier*

> *struct-or-union:*
> *struct*
> *union*

The struct-decl-list is a sequence of declarations for the members of the structure or union:

> *struct-decl-list:*
> *struct-declaration*
> *struct-declaration struct-decl-list*

> *struct-declaration:*
> *type-specifier struct-declarator-list ;*

> *struct-declarator-list:*
> *struct-declarator*
> *struct-declarator , struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field;* its length is set off from the field name by a colon.

> *struct-declarator:*
> *declarator*
> *declarator : constant-expression*
> *: constant-expression*

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the

PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator & may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier { struct-decl-list }*
union *identifier { struct-decl-list }*

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple example of a structure declaration is

```
struct tnode
{
        char tword [20] ;
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which **sp** points;

```
s.left
```

refers to the left subtree pointer of the structure **s**; and

```
s.right->tword[0]
```

refers to the first character of the **tword** member of the right subtree of **s**.

Enumerations are unique types with named constants. However, the current language treats enumeration variables and constants as being of **int** type.

```
enum-specifier:
        enum { enum-list }
        enum identifier { enum-list }
        enum identifier
enum-list:
        enumerator
        enum-list , enumerator
enumerator:
        identifier
        identifier = constant-expression
```

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

F. Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

```
initializer:
        = expression
        = { initializer-list }
        = { initializer-list , }
initializer-list:
        expression
        initializer-list , initializer-list
        { initializer-list }
```

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in part "CONSTANT EXPRESSIONS", or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0. Automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x [ ] = { 1, 3, 5 };
```

declares and initializes **x** as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y [4] [3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y [4] [3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y [4] [3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }

};
```

initializes the first column of y (regarded as a 2-dimensional array) and leaves the rest 0.

Finally,

```
        char msg[ ] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

G. Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

```
        type-name:
                type-specifier abstract-declarator

        abstract-declarator:
                empty
                ( abstract-declarator )
                * abstract-declarator
                abstract-declarator ()
                abstract-declarator [ constant-expression_opt ]
```

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
        int
        int *
        int * [3]
        int (*) [3]
        int *()
        int (*)()
```

name respectively the types "integer", "pointer to integer", "array of 3 pointers to integers", "pointer to an array of 3 integers", "function returning pointer to integer", and "pointer to function returning an integer".

H. Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

```
        typedef-name:
                identifier
```

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "D. Meaning of Declarators". For example, after

```
        typedef int MILES, *KLICKSP;
        typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**," and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

STATEMENTS

Except as indicated, statements are executed in sequence.

A. Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

B. Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
    { declaration-list$_{opt}$ statement-list$_{opt}$ }

declaration-list:
    declaration
    declaration declaration-list

statement-list:
    statement
    statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

C. Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

D. While Statement

The **while** statement has the form

> **while** (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

E. Do Statement

The do statement has the form

> **do** *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

F. For Statement

The **for** statement has the form:

> **for** *(expression-1$_{opt}$; expression-2$_{opt}$; expression-3$_{opt}$) statement*

This statement is equivalent to

```
expression-1 ;
while ( expression-2 )
{
        statement
        expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

G. Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

> **switch** (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

> **case** *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in part "CONSTANT EXPRESSIONS".

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default**, prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "H. Break Statement".

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

H. Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

I. Continue Statement

The statement

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...)         do                  for (...)
{                   {                   {
      ...                 ...                 ...
contin: ;           contin: ;           contin: ;
}                   } while (...);      }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see "M. Null Statement".)

J. Return Statement

A function returns to its caller by means of the **return** statement, which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

K. Goto Statement

Control may be transferred unconditionally by means of the statement

> **goto** *identifier* ;

The identifier must be a label (see "L. Labeled Statement") located in the current function.

L Labeled Statement

Any statement may be preceded by label prefixes of the form

> *identifier* :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See part "SCOPE RULES".

M. Null Statement

The null statement has the form

> ;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

EXTERNAL DEFINITIONS

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "B. Type Specifiers" in part "DECLARATIONS") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

A. External Function Definitions

Function definitions have the form

> *function-definition:*
>
> *decl-specifiers*$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "B. Scope of Externals" in part "SCOPE RULES" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

> *function-declarator:*
>
> *declarator (parameter-list*$_{opt}$ *)*

> *parameter-list:*
>
> *identifier*
>
> *identifier , parameter-list*

The function-body has the form

> *function-body:*
>
> declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
          int a, b, c;
{
          int m;
          m = (a > b) ? a : b;
          return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; { ... } is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...".

B. External Data Definitions

An external data definition has the form

data-definition:
    declaration

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

SCOPE RULES

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

A. Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also ("E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
     auto int distance;
     ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

B. Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the **extern** keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multifile program, an external data definition without the **extern** specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the **extern** in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

A. Token Replacement

A compiler-control line of the form

#define *identifier token-string*

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier , ... , identifier) token-string*

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100
int table [TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition to be forgotten.

B. File Inclusion

A compiler control line of the form

#include " *filename* "

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include *<filename>*

searches only the specified or standard places and not the directory of the source file. (How the places are specified is not part of the language.)

#include's may be nested.

C. Conditional Compilation

A compiler control line of the form

#if *constant-expression*

checks whether the constant expression evaluates to nonzero. (Constant expressions are discussed in part "CONSTANT EXPRESSIONS"; the following additional restriction applies here: the constant expression may not contain **sizeof** or an enumeration constant.) A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

D. Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant* "*filename*"

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent, the remembered file name does not change.

IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...", it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**".

TYPES REVISITED

This part summarizes the operations which can be performed on objects of certain types.

A. Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int         type;
        } n;
        struct
        {
                int         type;
                int         intnode;
        } ni;
        struct
        {
                int         type;
                float       floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

B. Functions

There are only two things that can be done with a function—call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

C. Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules which apply to **+**, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2**−th member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If E is an *n*−dimensional array of rank i×j×···×k, then E appearing in an expression is converted to a pointer to an (n−1)-dimensional array with rank j×···×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n−1)-dimensional array, which itself is immediately converted into a pointer.

If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n−1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];;
```

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then i is converted to the type of **x**, which involves multiplying i by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

D. Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "B. Unary Operators" in part "EXPRESSIONS" and "G. Type Names" in part "DECLARATIONS".

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The **chars** have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word lie just to their right. Thus **char** pointers measure units of 2^{16} bytes; everything else is measured in units of 2^{18} machine words. The **double** quantities and aggregates containing them must lie on an even word address (0 mod 2^{19}).

The IBM 370 and the Interdata 8/32 are similar. On each, pointers are 32-bit quantities that measure bytes; elementary objects are aligned on a boundary equal to their length, so pointers to **short** must be 0 mod 2, to **int** and **float** 0 mod 4, and to **double** 0 mod 8. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B20 and 3B5 Processors characteristics are the same. On each, pointers are 24-bit quantities. Most objects are aligned on 4-byte boundaries. **Shorts** are aligned in all cases on 2-byte boundries. Arrays of characters, all structures, **inits**, **longs**, **floats**, and **doubles** are aligned on 4-byte boundries; but structure members may be packed tighter.

CONSTANT EXPRESSIONS

In several places C requires expressions which evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   <   >   <=   >=
```

or by the unary operators

```
-   ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Less latitude is allowed for constant expressions after #if; **sizeof** expressions and enumeration constants are not permitted.

PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11 and VAX-11; left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on the PDP-11 and VAX-11 and left to right on other machines. These differences are invisible to isolated programs which do not indulge in type running (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit fields and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

ANACHRONISMS

Because C is an evolving language, certain obsolete constructions may be found in older programs. Although some versions of the compiler support such anachronisms, they have by and large disappeared leaving only a portability problem behind.

Earlier versions of C used the form *=op* instead of *op=* for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which assigns **−1** to **x**, but previously decremented **x**.

The syntax of initializers has changed. Previously, the equals sign that introduces an initializer was not present, so instead of

```
int     x = 1;
```

one used

```
int     x 1;
```

The change was made because the initialization

```
int     f (1)
```

resembles a function declaration closely enough to confuse the compilers.

A structure or union member reference is a chain of member references (qualifications) that are prefixed by either a pointer to a structure or union or a structure or union proper. Because each qualification implies the addition of an offset within an address computation, older compilers (which failed to check for membership in the appropriate structure or union) allowed omission of those qualifications with an offset of zero. Complete qualification is now required.

Previous versions of the compiler were lax in detecting mixed assignments involving pointers and arithmetic quantities. These are now remarked upon.

SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

A. Expressions

The basic expressions are:

```
expression:
          primary
          * expression
          & lvalue
          - expression
          ! expression
          ~ expression
          ++ lvalue
          -- lvalue
          lvalue ++
          lvalue --
          sizeof expression
          ( type-name ) expression
          expression binop expression
          expression ? expression : expression
          lvalue asgnop expression
          expression , expression
primary:
          identifier
          constant
          string
          ( expression )
          primary ( expression-list_opt )
          primary [ expression ]
          primary . identifier
          primary -> identifier
lvalue:
          identifier
          primary [ expression ]
          lvalue . identifier
          primary -> identifier
          * expression
          ( lvalue )
```

The primary-expression operators

```
( ) [ ] . ->
```

have highest priority and group left to right. The unary operators

```
        *   &   -   !   ~   ++   --   sizeof    ( type-name )
```

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```
            *       /       %
            +       -
            >>      <<
            <       >       <=      >=
            ==      !=
            &
            ^
            |
            &&
            ||
            ?:
```

Assignment operators all have the same priority and all group right to left.

asgnop:

```
            =   +=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

The comma operator has the lowest priority and groups left to right.

B. Declarations

declaration:
  decl-specifiers init-declarator-list$_{opt}$ *;*

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

sc-specifier:
  auto
  static
  extern
  register
  typedef

type-specifier:
  char
  short
  int
  long
  unsigned
  float
  double
  void
  struct-or-union-specifier

typedef-name
enum-specifier

enum-specifier:
  enum *{ enum-list }*
  enum *identifier { enum-list }*
  enum *identifier*

enum-list:
  enumerator
  enum-list , enumerator

enumerator:
  identifier
  identifier = constant-expression

init-declarator-list:
  init-declarator
  init-declarator , init-declarator-list

init-declarator:
  declarator initializer$_{opt}$

declarator:
  identifier
  (declarator)
  ** declarator*
  declarator ( )
  declarator [constant-expression$_{opt}$]

struct-or-union-specifier:
  struct *{ struct-decl-list }*
  struct *identifier { struct-decl-list }*
  struct *identifier*
  union *{ struct-decl-list }*
  union *identifier { struct-decl-list }*
  union *identifier*

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , struct-declarator-list

struct-declarator:
  declarator
  declarator : constant-expression
  : constant-expression

initializer:

> *= expression*
> *= { initializer-list }*
> *= { initializer-list , }*

initializer-list:

> *expression*
> *initializer-list , initializer-list*
> *{ initializer-list }*

type-name:

> *type-specifier abstract-declarator*

abstract-declarator:

> *empty*
> *(abstract-declarator)*
> ** abstract-declarator*
> *abstract-declarator ( )*
> *abstract-declarator [constant-expression*$_{opt}$ *]*

typedef-name:

> *identifier*

C. Statements

compound-statement:

> *{ declaration-list*$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:

> *declaration*
> *declaration declaration-list*

statement-list:

> *statement*
> *statement statement-list*

statement:

> *compound-statement*
> *expression ;*
> **if** *(expression) statement*
> **if** *(expression) statement* **else** *statement*
> **while** *(expression) statement*
> **do** *statement* **while** *(expression) ;*
> **for** *(expression-1*$_{opt}$ *; expression-2*$_{opt}$ *; expression-3*$_{opt}$ *) statement*
> **switch** *(expression) statement*
> **case** *constant-expression : statement*
> **default** *: statement*
> **break ;**
> **continue ;**
> **return ;**
> **return** *expression ;*
> **goto** *identifier ;*
> *identifier : statement*
> *;*

D. External Definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ function-declarator function-body

function-declarator:
declarator (parameter-list$_{opt}$)

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list compound-statement

data-definition:
extern$_{opt}$ *type-specifier$_{opt}$ init-declarator-list$_{opt}$;*
static$_{opt}$ *type-specifier$_{opt}$ init-declarator-list$_{opt}$;*

E. Preprocessor

#define *identifier token-string*
#define *identifier (identifier , ... , identifier) token-string*
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant* "*filename*"

LIBRARIES

A. General

This part describes the libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Part 3 of the UNIX System User's Manual. Most of the declarations described are in Part of the UNIX System User's Manual. The three main libraries on the UNIX system are:

C library	This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions.
Object file	This library provides functions for the access and manipulation of object files.
Math library	This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions.

Some libraries consist of two portions—functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being complied. In other cases, the functions (and/or declarations) are included automatically.

Including Functions

When a program is being complied, the complier will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the complier to locate and include functions from other libraries, the user must specifiy these libraries on the command line for the complier. For example, when using functions of the math library, the user must request that the math library be searched by including the argument −lm on the command line, such as:

```
        cc file.c −lm
```

This method should be used for all functions that are not part of the C language library.

Including Declarations

Some functions require a set of declarations in order to operate properly. The declarations of all libraries must be included by request of the user. A set of declarations is stored in a file under the */usr/include* directory. These files are referred to as ***header files***. In order to include a certain header file, the user must specifiy this request within the C language program. The request is in the form:

```
        #include <file.h>
```

where *file* is the name of the file. Since this request is handled by the preprocessor, header files should appear at the beginning of the (first) file being complied.

The remainder of this part decribes the functions and header files of the various libraries. The description of each library begins with the actions required by the user to include the functions and/or header files in a program being complied (if any). Following the description of the actions required, is information in three column format of the form:

function	**reference(N)**	Brief description.

The functions are grouped by type while the reference refers to section 'N' in the UNIX System User's Manual Following this, are descriptions of the header files associated with these functions (if any).

B. The C Library

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the complier. Various declarations must be included by the user as required. The functions of the C library are divided into the following types:

- input/output control
- string manipulation
- character manipulation
- time functions
- miscellaneous functions.

Input/Output Control

These functions of the C library are included as needed during the compiling of a C language program automatically. No command line request is needed.

The header file required by the input/output functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <stdio.h>
```

near the beginning of the (first) file being compiled.

The input/output functions are grouped into the following categories:

- file access
- file status
- input
- output
- miscellaneous.

File Access Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fclose	fclose(3S)	Close an open stream.
fdopen	fopen(3S)	Associate stream with an open(2) ed file.
fileno	ferror(3S)	Integer associated with an open stream.
fopen	fopen(3S)	Open a stream with specified permissions. A "stream" is defined to be what fopen returns.
freopen	fopen(3S)	Substitute named file in place of open stream.

fseek	**fseek**(3S)	Reposition stream pointer.
pclose	**popen**(3S)	Close a stream opened by **popen**.
popen	**popen**(3S)	Create pipe as a stream between calling process and command.
rewind	**fseek**(3S)	Reposition stream pointer at beginning of file.
setbuf	**setbuf**(3S)	Assign buffering to stream.

File Status Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
clearerr	**ferror**(3S)	Reset error condition on stream.
feof	**ferror**(3S)	Test for "end of file" on stream.
ferror	**ferror**(3S)	Test for error condition on stream.
ftell	**fseek**(3S)	Return current stream pointer.

Input Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fgetc	**getc**(3S)	True function for **getc** (3S).
fgets	**gets**(3S)	Read string from stream.
fread	**fread**(3S)	General buffered read from stream.
fscanf	**scanf**(3S)	Read using format from stream.
getc	**getc**(3S)	Return next character from stream.
getchar	**getc**(3S)	Return next character from **stdin**.
gets	**gets**(3S)	Read string from **stdin**.
getw	**getc**(3S)	Read word from stream.
scanf	**scanf**(3S)	Read using format from **stdin**.
sscanf	**scanf**(3S)	Read using format from string.
ungetc	**ungetc**(3S)	Put back one character on stream.

Output Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fflush	**fclose**(3S)	Write all currently buffered characters from stream.

fprintf	**printf(3S)**	Print using format to stream.
fputc	**putc(3S)**	True function for **putc** (3S).
fputs	**puts(3S)**	Write string to stream.
fwrite	**fread(3S)**	General buffered write to stream.
printf	**printf(3S)**	Print using format to **stdout**.
putc	**putc(3S)**	Write next character to stream.
putchar	**putc(3S)**	Write next character to **stdout**.
puts	**puts(3S)**	Write string to **stdout**.
putw	**putc(3S)**	Write word to stream.
sprintf	**printf(3S)**	Write using format to string.

Miscellaneous Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ctermid	**ctermid(3S)**	Return file name for controlling terminal.
cuserid	**cuserid(3S)**	Return login name for owner of current process.
system	**system(3S)**	Execute system command.
tempnam	**tmpnam(3S)**	Create temporary file name using directory and prefix.
tmpnam	**tmpnam(3S)**	Create temporary file name.
tmpfile	**tmpfile(3S)**	Create temporary file.

Input/Output Header File (stdio.h)

The following listing is the contents of *stdio.h* for the 3B20S Processor. Note that along with parameters used by the stdio functions several macros are defined. The following files are open automatically by the system:

stdin	standard input file
stdout	standard output file
stderr	standard error file

Also, note that the functions **fopen**, **fdopen**, and **freopen** are declared as returning a structure of type *FILE;* **fgets** and **gets** are declared as returning character pointers; and **ftell** is declared as returning a long integer.

```
/* this example is included as */
/* a typical illustration of the */
/* header file stdio.h */
#ifndef _NFILE
#define _NFILE          20
```

```
#define BUFSIZ          512
typedef struct {
unsigned char *_ptr;
int _cnt;
unsigned char *_base;
char _flag;
char _file;
} FILE;
#define _IOREAD 01
#define _IOWRT 02
#define _IONBF 04
#define _IOMYBUF 010
#define _IOEOF 020
#define _IOERR 040
#define _IOUNK 0100 /* indicates buffering status unknown */
#define _IORW 0200
#ifndef NULL
#define NULL 0
#endif
#ifndef EOF
#define EOF (-1)
#endif
#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
#ifndef lint
#define getc(p) ((--(p)->_cnt) >= 0)?((int) *((p)->_ptr)++): _filbuf(p))
#endif
#define getchar() getc(stdin)
#ifndef lint
#define putc(x,p) ((--((p)->_cnt) >= 0)?((int)) (*((p)->_ptr)++ = (unsigned char)(x))): \
_flsbuf((unsigned char)(x),p))
#endif
#define putchar(x) putc(x,stdout)
#define feof(p) (((p)->_flag & _IOEOF) != 0)
#define ferror(p) (((p)->_flag & _IOERR) != 0)
#define fileno(p) p->file
extern FILE _iob[_NFILE];
extern FILE *fopen( );
extern FILE *fdopen( );
extern FILE *freopen( );
extern long ftell( );
extern char *fgets( );
extern char *gets( );
#define L_ctermid 9
#define L_cuserid 9
#define P_tmpdir "/usr/tmp/"
#define L_tmpnam 10+sizeof(P_tmpdir)
#endif
```

String Manipulation Functions

These functions are used to locate characters within a string, copy, concatenate, and compare strings. These functions are located and loaded during the compling of a C language program automatically. No command line request is needed since these functions are part of the C library. There are no declarations associated with these functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
strcat	**string**(3C)	Concatenate two strings.
strchr	**string**(3C)	Search string for character.
strcmp	**string**(3C)	Compares two strings.
strcpy	**string**(3C)	Copy string over string.
strcspn	**string**(3C)	Length of initial string not containing set of characters.
strlen	**string**(3C)	Length of string.
strncat	**string**(3C)	Concatenate two strings with fixed length.
strncmp	**string**(3C)	Compares two strings with fixed length.
strncpy	**string**(3C)	Copy string over string with fixed length.
strpbrk	**string**(3C)	Search string for any set of characters.
strrchr	**string**(3C)	Search string backwards for character.
strspn	**string**(3C)	Length of initial string containing set of characters.
strtok	**string**(3C)	Search string for token separated by any of a set of characters.

Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <ctype.h>
```

near the beginning of the (first) file being compiled.

Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
isalnum	**ctype**(3C)	Is character alphanumeric?
isalpha	**ctype**(3C)	Is character alphabetic?

isascii	ctype(3C)	Is integer ASCII character?
iscntrl	ctype(3C)	Is character a control character?
isdigit	ctype(3C)	Is character a digit?
isgraph	ctype(3C)	Is character a printing character?
islower	ctype(3C)	Is character a lowercase letter?
isprint	ctype(3C)	Is character a printing character including space?
ispunct	ctype(3C)	Is character a punctuation character?
isspace	ctype(3C)	Is character a white space character?
isupper	ctype(3C)	Is character an uppercase letter?
isxdigit	ctype(3C)	Is character a hex digit?

Character Translation Functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
toascii	conv(3C)	Convert integer to ASCII character.
tolower	conv(3C)	Convert character to lowercase.
toupper	conv(3C)	Convert character to uppercase.

Character Header File (ctype.h)

The following listing is the *ctype.h* file which is located in the */usr/include* directory. This file is included in a program with the line:

```
        #include <ctype.h>
```

This file provides a few data declarations and defines the macros.

```
/* this example is included as */
/* a typical illustration of the */
/* header file ctype.h */
#define   _U   01
#define   _L   02
#define   _N   04
#define   _S   010
#define   _P   020
#define   _C   040
#define   _B   0100
#define   _X   0200
extern char   _ctype[];
```

```
#define  isalpha(c)    ((_ctype+1)[c]&(_U|_L))
#define  isupper(c)    ((_ctype+1)[c]&_U)
#define  islower(c)    ((_ctype+1)[c]&_L)
#define  isdigit(c)    ((_ctype+1)[c]&_N)
#define  isxdigit(c)   ((_ctype+1)[c]&_X)
#define  isspace(c)    ((_ctype+1)[c]&_S)
#define  ispunct(c)    ((_ctype+1)[c]&_P)
#define  isalnum(c)    ((_ctype+1)[c]&(_U|_L|_N))
#define  isprint(c)    ((_ctype+1)[c]&(_P|_U|_L|_N|_B))
#define  isgraph(c)    ((_ctype+1)[c]&(_P|_U|_L|_N))
#define  iscntrl(c)    ((_ctype+1)[c]&_C)
#define  isascii(c)    ((unsigned char)(c)<=0177)
#define  _toupper(c)   ((c)-'a'+'A')
#define  _tolower(c)   ((c)-'A'+'a')
#define  toascii(c)    ((c)&0177)
```

Time Functions

These functions are used for accessing and reformatting the systems idea of the current date and time. These functions are located and loaded automatically during the compling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <time.h>
```

near the beginning of the (first) file being compiled.

These functions (except **tzset**) convert a time such as returned by **time**(2).

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
asctime	**ctime**(3C)	Return string representation of date and time.
ctime	**ctime**(3C)	Return string representation of date and time, given integer form.
gmtime	**ctime**(3C)	Return Greenwich Mean Time.
localtime	**ctime**(3C)	Return local time.
tzset	**ctime**(3C)	Set time zone field from environment variable.

Time Header File (time.h)

The following listing is the *time.h* file which is located in the */usr/include* directory. The *tm* structure is the type of structure returned by **gmtime** and **localtime**. Note that the **gmtime** and **localtime** functions are declared as returning pointers to structures of type *tm* and **ctime** and **asctime** are declared as returning character pointers.

The long *timezone* variable contains the difference (in seconds) between GMT and local standard time. For EST, this difference is 18,000 seconds. The *daylight* variable is nonzero if Daylight Savings Time conversion should be applied. The *tzname* variable defines the time zone names. For EST, the declaration would be **char *tzname [2] = " EST" ," EDT" ;**

If the environment variable *TZ* is present, **asctime** uses the contents of the variable to override the default time zone. The value of *TZ* must be a 3-letter time zone name, followed by a number representing the difference between GMT and local time (in hours). Following the time difference is an optional 3-letter name for a daylight time zone. For example, for users in EST, the value of *TZ* would be **EST5EDT**. By setting the *TZ* variable, the values of *timezone, daylight,* and *tzname* are changed accordingly.

```
/* this example is included as */
/* a typical illustration of the */
/* header file time.h */
struct tm {
int tm_sec;
int tm_min;
int tm_hour; /* hour of day (0 to 24) */
int tm_mday; /* day of month (1 to 31) */
int tm_mon; /* month of year (0 to 11) */
int tm_year; /* last two digits of current year */
int tm_wday; /* day of week (Sunday = 0) */
int tm_yday; /* day of year (0 to 365) */
int tm_isdst;   /* nonzero if DST in effect */
};
extern struct tm *gmtime( ), *localtime( );
extern char *ctime( ), *asctime( );
extern void tzset( );
extern long timezone;
extern int daylight;
extern char *tzname[ ];
```

Miscellaneous Functions

These functions support a wide variety of operations. Some of these are numerical conversion, password file and group file access, memory allocation, random number generation, and table management. These functions are located and included in a program being complied automatically. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

Numerical Conversion

The following functions perform numerical conversion.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
a641	**a641**(3C)	Convert string to base 64 ASCII.
atof	**atof**(3C)	Convert string to floating.
atoi	**atof**(3C)	Convert string to integer.
atol	**atof**(3C)	Convert string to long.
frexp	**frexp**(3C)	Split floating into mantissa and exponent.

l3tol	l3tol(3C)	Convert 3-byte integer to long.
ltol3	l3tol(3C)	Convert long to 3-byte integer.
ldexp	frexp(3C)	Combine mantissa and exponent.
l64a	a64l(3C)	Convert base 64 ASCII to string.
modf	frexp(3C)	Split mantissa into integer and fraction.

DES Algorithm Access

The following functions allow access to the DES algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
crypt	crypt(3C)	Encode string using salt.
encrypt	crypt(3C)	Encode/decode string of 0's and 1's.
setkey	crypt(3C)	Initialize for subsequent use of **encrypt**.

Group File Access

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

```
#include <grp.h>
```

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
endgrent	getgrent(3C)	Close group file being processed.
getgrent	getgrent(3C)	Get next group file entry.
getgrgid	getgrent(3C)	Return next group with matching gid.
getgrnam	getgrent(3C)	Return next group with matching name.
setgrent	getgrent(3C)	Rewind group file being processed.

Password File Access

These functions are used to search and access information stored in the password file (/etc/passwd). Some functions require declarations that can be included in the program being compiled by adding the line:

```
#include <pwd.h>
```

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
endpwent	getpwent(3C)	Close password file being processed.
getpw	getpw(3C)	Search password file for uid.

getpwent	getpwent(3C)	Get next password file entry.
getpwnam	getpwent(3C)	Return next entry with matching name.
getpwuid	getpwent(3C)	Return next entry with matching uid.
putpwent	putpwent(3C)	Write entry on stream.
setpwent	getpwent(3C)	Rewind password file being accessed.

Parameter Access

The following functions provide access to several different types of paramenters. None require any declarations.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
getopt	getopt(3C)	Get next option from option list.
getcwd	getcwd(3C)	Return string representation of current working directory.
getenv	getenv(3C)	Return string value associated with environment variable.
getpass	getpass(3C)	Read string from terminal without echoing.

Hash Table Management

The following functions are used to manage hash search tables.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
hcreate	hsearch(3C)	Create hash table.
hdestroy	hsearch(3C)	Destroy hash table.
hsearch	hsearch(3C)	Search hash table for entry.

Binary Tree Management

The following functions are used to manage a binary tree.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
tdelete	tsearch(3C)	Deletes nodes from binary tree.
tsearch	tsearch(3C)	Search binary tree.
twalk	tsearch(3C)	Walk binary tree.

Table Management

The following functions are used to manage a table. The "table" is basically a 2-dimensional character array. The first subscript defines the maximum number of entries in the table. The second subscript defines the width (or length) of a single entry. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
bsearch	**bsearch**(3C)	Search table using binary search.
lsearch	**lsearch**(3C)	Search table using linear search.
qsort	**qsort**(3C)	Sort table using quicker-sort algorithm.

Memory Allocation

The following functions provide a means by which memory can be dynamically allocated or freed.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
calloc	**malloc**(3C)	Allocate zeroed storage.
free	**malloc**(3C)	Free previously allocated storage.
malloc	**malloc**(3C)	Allocate storage.
realloc	**malloc**(3C)	Change size of allocated storage.

Pseudorandom Number Generation

The following functions are used to generate pseudorandom numbers. The functions that end with 48 are a family of interfaces to a pseudorandom number generator based upon the linear congruential algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
drand48	**drand48**(3C)	Random double over the interval [0 to 1).
lcong48	**drand48**(3C)	Set parameters for **drand48**, **lrand48**,and **mrand48**.
lrand48	**drand48**(3C)	Random long over the interval [0 to 2^{31}).
mrand48	**drand48**(3C)	Random long over the interval [-2^{31} to 2^{31}).
rand	**rand**(3C)	Random integer over the interval [0 to 214).
seed48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48**.
srand	**rand**(3C)	Seed the generator for rand.
srand48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48** using a long.

Signal Handling Functions

The functions **gsignal** and **ssignal** implement a software facility similar to **signal(2)** in the UNIX System User's Manual. This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions can be included in the program being complied by the line

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
gsinal	**ssignal**(3C)	Send a software signal.
ssignal	**ssignal**(3C)	Arrange for handling of software signals.

Miscellaneous

The following functions do not fall into any previously described category.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
abort	**abort**(3C)	Cause an IOT signal to be sent to the process.
abs	**abs**(3C)	Return the absolute integer value.
ecvt	**ecvt**(3C)	Convert double to string.
fcvt	**ecvt**(3C)	Convert double to string using Fortran format.
gcvt	**ecvt**(3C)	Convert double to string using Fortran F or E format.
isatty	**ttyname**(3C)	Test whether integer file descriptor is associated with a terminal.
mktemp	**mktemp**(3C)	Create file using template.
monitor	**monitor**(3C)	Cause process to record a histogram of program counter location.
swab	**swab**(3C)	Swap and copy bytes.
ttyname	**ttyname**(3C)	Return pathname of terminal associated with integer file descriptor.

Group File Header File (grp.h)

The *grp.h* file provides the structure used by several group file functions. This file can be included in a program by adding the line:

```
#include <grp.h>
```

```
/* this example is included as */
/* a typical illustration of the */
/* header file grp.h */
struct group {
char *gr_name; /* group name */
char *gr_passwd;   /* group password */
int gr_gid;   /* group id */
char **gr_mem;   /* list of pointers to group members */
};
```

Password File Header File (pwd.h)

The following listing describes the contents of *pwd.h* which is used by several of the password file functions. This file can be included in the program with the line:

```
#include <pwd.h>
```

The **pw_comment** field is actually a structure of type *comment*.

```
/* this example is included as */
/* a typical illustration of the */
/* header file pwd.h */
struct passwd {
char *pw_name; /* login name */
char *pw_passwd; /* password */
int pw_uid; /* user id */
int pw_gid; /* group id */
char *pw_age; /* age of password */
char *pw_comment; /* comments */
char *pw_gecos; /* optional GCOS user id */
char *pw_dir; /* login directory */
char *pw_shell; /*shellused by this login */
};
struct comment {
char *c_dept; /* user's department */
char *c_name; /* user's name */
char *c_acct; /* user's account number */
char *c_bin; /* user's mail bin */
};
```

Signal Handling Header File (signal.h)

The following listing describes the *signal.h* file which is under the */usr/include* directory. This file can be included by adding the line:

```
#include <signal.h>
```

The *signal.h* file contains the declarations used by the functions that handle signals. Most of this file defines *names* to each numerical signal.

```
/* this example is included as */
/* a typical illustration of the */
/* header file signal.h */
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction (not reset when caught)*/
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT 6 /* IOT instruction */
#define SIGEMT 7 /* EMT instruction */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD 18 /* death of a child */
#define SIGPWR 19 /* power-fail restart */
#define NSIG 20
#define SIG_DFL (int (*)( ))0
#if lint
#define SIG_IGN (int (*)( ))0
#else
#define SIG_IGN (int (*)( ))1
#endif
extern (*signal( ))( );
```

C. The Object File Library

The object file library provides functions to access object files. The functions allow access closing to single object files or object files that are part of an archive. Some functions locate portions of an object file such as the symbol table, the file header, sections, and lines within functions. Other functions read these types of entries into memory.

This library consists of several portions. The functions reside in */usr/lib/libld.a* and are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

```
cc file -lld
```

which causes the link editor to search the object file library.

In addition, various header files must be included. This is accomplished by including the line:

```
#include <sgs/file>
```

where *file* is the name of the appropriate header file. The header files required for each function are defined at the beginning of each function description. Following the descriptions of the functions is a description of the header files. The **HEADER** macro returns a pointer to the *HEADER* field of the *LDFILE* structure pointed to by *ldptr.* The *HEADER* field is the file header structure of the object file. The **IOPTR** macro returns the contents of the *IOPTR* field of the *LDFILE* structure pointed to by *ldptr.* The *IOPTR* field contains a file pointer returned by **fopen** and used by the input/output functions of the C library. The **TYPE** macro returns the *TYPE* field of the *LDFILE* structure pointed to by *ldptr*. The *TYPE* field contains the file magic number which is used to distinguish between archive members and simple object files.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ldaclose	ldclose(3X)	Close object file being processed.
ldahread	ldahread(3X)	Read archive header.
ldaopen	ldopen(3X)	Open object file for reading.
ldclose	ldclose(3X)	Close object file being processed.
ldfhread	ldfhread(3X)	Read file header of object file being processed.
ldlinit	ldlread(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	ldlread(3X)	Read line number entry from object file after **ldlinit**.
ldlread	ldlread(3X)	Read line number entry from object file.
ldlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed given name.
ldnrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed given name.
ldnshread	ldshread(3X)	Read section header of the named section of the object file being processed.
ldnsseek	ldsseek(3X)	Seeks to the section of the object file being processed given name.
ldohseek	ldohseek(3X)	Seeks to the optional file header of the object file being processed.
ldopen	ldopen(3X)	Open object file for reading.
ldrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	ldshread(3X)	Read section header of an object file being processed.
ldsseek	ldsseek(3X)	Seeks to the section of the object file being processed.
ldtbindex	ldtbindex(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	ldtbread(3X)	Reads the symbol table entry specified by *symindex* of the object file being processed.
ldtbseek	ldtbseek(3X)	Seeks to the symbol table of the object file being processed.

D. The Math Library

The math library consists of functions and a header file. The functions are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

```
cc file -lm
```

which causes the link editor to search the math library. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of the (first) file being compiled.

The functions are grouped into the following categories:

- trigonometric functions
- bessel functions
- hyperbolic functions
- miscellaneous functions.

Trigonometric Functions

These functions are used to compute angles (in decimal radian measure), sines, cosines, and tangents. All of these values are expressed in double precision and should always be declared as such.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
acos	**trig(3M)**	Return arc cosine.
asin	**trig(3M)**	Return arc sine.
atan	**trig(3M)**	Return arc tangent.
atan2	**trig(3M)**	Return arc tangent of a ratio.
cos	**trig(3M)**	Return cosine.
hypot	**hypot(3M)**	Return the square root of the sum of the squares of two numbers.
sin	**trig(3M)**	Return sine.
tan	**trig(3M)**	Return tangent.

Bessel Functions

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel(3M)**.

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the decimal portion of double precision numbers.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of gamma as a function of the absolute value of a given value.
log	**exp**(3M)	Returns the natural logarithm of a given value.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

Math Header File (math.h)

The following listing is the *math.h* file which is located in the */usr/include* directory. Note that all the math functions are declared as returning double-precision values. Also note that *HUGE* is defined as

$$1.701411733192644270 \times 10^{38}$$

```
/* this example is included as */
/* a typical illustration of the */
/* math header file (math.h) */
extern double fabs( ), floor( ), ceil( ), fmod( ), ldexp( );
extern double sqrt( ), hypot( ), atof( );
extern double sin( ), cos( ), tan( ), asin( ), acos( ), atan( ), atan2( );
extern double exp( ), log( ), log10( ), pow( );
extern double sinh( ), cosh( ), tanh( );
extern double gamma( );
extern double j0( ), j1( ), jn( ), y0( ), y1( ), yn( );
#define HUGE          1.701411733192644270e38
```

THE "cc" COMMAND

A. General

The C compiler cc(1) is used to compile C language programs or assembly language programs into machine language. This document briefly describes the usage of the C compiler. Most of this information is in the UNIX System User's Manual.

B. Usage

The cc command is invoked as:

```
cc options files
```

where *options* control the compiling; *files* are the files to be compiled.

The following options are interpreted by cc. See ld(1) for link editor options.

−c	Suppresses the link edit phase of the compilation and forces an object file to be produced even if only one program is compiled.
−D*name*=*def*	
−D*name*	Defines the *name* to the preprocessor, as if by **#define**. If no definition is given, the name is defined as 1.
−E	Runs only the macro preprocessor on the named C language programs and sends the result to the standard output.
−g	Arranges for the compiler to produce additional information needed for the use of sdb(1).
−I*dir*	Changes the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in " " will be searched for first in the directory of the *file* argument, then in directories named in the −I options, and last in directories on a standard list. For **#include** files whose names are enclosed by <...>, the directory of the *file* argument is not searched.
−O	Invokes an object-code optimizer. The optimizer will move, merge, and delete code so symbolic debugging with line numbers could be confusing when the optimizer is used.
−p	Arranges for the compiler to produce code which counts the number of times each routine is called. Also, if link editing takes place, replaces the standard startoff routine by one which automatically calls **monitor(3C)** at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can be generated by use of **prof(1)**.
−P	Runs only the macro preprocessor on the named C language programs and leaves the result on corresponding files suffixed .i.

−S	Compiles the named C language programs and leaves the assembler-language output on corresponding files suffixed **.s**.
−U*name*	Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes the operating system [unix (this reserved symbol refers to the UNIX operating system), gcos, os, tss, or u370], the hardware (ibm, interdata, pdp11, u3b, or VAX), and the UNIX system variant (RES, RT, or mert).

Other options are taken to be either link editor options or C-compatible object programs.

Arguments that end with **.c** are taken to be C language source programs; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**. The **.o** file is normally deleted.

In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled producing a **.o** file.

These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name **a.out**.

A C PROGRAM CHECKER—"lint"

A. General

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions which nevertheless are legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint**(1) command has the form:

```
lint [options] files ... library-descriptors ...
```

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

−a	Suppress messages about assignments of long values to variables that are not long.
−b	Suppress messages about break statements that cannot be reached.
−c	Suppress messages about casts that have questionable portability.
−h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
−n	Do not check for compatibility with either the standard or the portable **lint** library.
−p	Attempt to check portability to other dialects of C language (IBM and Honeywell).
−u	Suppress messages about function and external variables used and not defined or defined and not used.

−v	Suppress messages about unused arguments in functions.
−x	Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, **−ab** or **−xha**.

The names of files that contain C language programs should end with the suffix **.c** which is mandatory or **lint** and the C complier.

The **lint** program accepts certain arguments, such as:

−ly

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The lint program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the **−p** option is used, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

B. Types of Messages

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** program prints messages about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern  float  sin();
```

will evoke no comment if **sin** is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest and can be discovered by using the **−x** option with the **lint** command.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The −v option is available to suppress the printing of messages about unused arguments. When v is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the −v option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked, such as:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The −u option may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

The **lint** program attempts to detect cases where a variable is used before it is set. The **lint** program detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables which are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom and recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never return. Thus, a call to **exit** may cause an unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

```
        /* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The −O option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **−b** option.

Function Values

Sometimes functions return values which are never used. Sometimes programs incorrectly use function "values" which have never been returned. The **lint** program addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
        return(   expr   );
```

and

```
        return ;
```

statements is cause for alarm; the **lint** program will give the message

```
        function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
        f ( a ) {
                if ( a ) return ( 3 );
                g ( );
                }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a message from **lint**. If *g*, like **exit**, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial portion of the redundant messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value; and this value is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- Across certain binary operators and implied assignments

- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the −> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his/her intentions. It seems harsh for **lint** to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **−c** flag controls the printing of comments about casts. When **−c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On some systems, characters are signed quantities with a range from −128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
  ...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to ints, which is truncated. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **−a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **−h** option is used to enable these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting making such bugs extremely hard to find. For example, the statement

```
if( x&077 == 0 ) ...
```

or

```
        x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Finally, when the **−h** option has been used, **lint** prints messages about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes—assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =−, ...) could cause ambiguous expressions, such as:

```
        a =−1 ;
```

which could be taken as either

```
        a =− 1 ;
```

or

```
        a = −1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, −=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
        int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
        int x ( −1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
        int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
        int x = −1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers since double-precision values may begin on any integer boundary. On the Honeywell 6000, double-precision values must begin on even word boundaries. Thus, not all such assignments make sense. The **lint** program tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **−p** or **−h** options are used.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

C. Portability

C language on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C language tends to follow local conventions rather than adhere strictly to UNIX system conventions. Despite these differences, many C language programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations and discusses the **lint** features which encourage portability.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters with the uppercase and lowercase distinction kept. On the IBM systems, there are eight significant characters; but the case distinction is lost. On GCOS, there are only six characters of a single case. This leads to situations where programs run on the UNIX system but encounter loader problems on the IBM or GCOS systems. The **−p** option causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling. Characters in the UNIX system are 8-bit ASCII, while they are 8-bit EBCDIC on the IBM and 9-bit ASCII on GCOS. Also, character strings go from high- to low-bit positions "left to right" on GCOS and IBM and low to high "right to left" on the PDP-11. This means that code attempting to construct strings out of character constants or attempting to use characters as indices into arrays must be looked at with great suspicion. The **lint** program is of little help here except to flag multicharacter character constants.

Of course, the word size differs from machine to machine. This causes less trouble than might be expected at least when moving from the UNIX system (16-bit words) to the IBM (32-bits) or GCOS (36-bits). The main problems are likely to arise in shifting or masking. The C language now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
        x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11 but fails badly on GCOS and IBM. If the bit-field feature cannot be used, the same effect can be obtained by writing

```
        x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11 and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11 and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, the C language could be changed. In any case, **lint** is no help here.

The previous discussion may have made the problem of portability seem bigger than it is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file or to establish a pipe between processes has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, **lint** has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

A SYMBOLIC DEBUGGING PROGRAM—"sdb"

A. General

This part describes the symbolic debugger **sdb(1)** as implemented for C language and Fortran 77 programs on the UNIX operating system. The **sdb** program is useful both for examining "core images" of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

Breakpoints may be placed at selected statements or the program may be single stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provided formatted printout of structured data.

B. Usage

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **−g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **−g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of **shell** commands for debugging a core image is

```
$ cc -g prgm.c -o prgm
$ prgm
Bus error - core dumped
$ sdb prgm
main:25:        x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function *main* at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an * indicating that it awaits a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to *main* and "25", respectively.

In the above example, **sdb** was called with one argument, *prgm*. In general, it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; it defaults to *a.out* when not specified. The second is the name of the core file, defaulting to *core;* and the third is the name of the directory containing the source of the program being debugged. The **sdb** program currently requires all source to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function which was not compiled with the **-g** option. In this case, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in *main*. The **sdb** program will print an error message if *main* was not compiled with the **-g** option, but debugging can continue for those routines compiled with the **-g** option. Figure 3.1 shows a typical example of **sdb** usage.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls which led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)        [prgm.c:25]
inter(i=16012)       [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c)        [prgm.c:15]
```

This indicates that the error occurred within the function *sub* at line 25 in file *prgm.c*. The *sub* function was called with the arguments x=2 and y=3 from *inter* at line 96. The *inter* function was called from *main* at line 15. The *main* function is always called by the **shell** with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable *errflg*. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable *i* in function *sub*. F77 users can specify a common block variable in the same manner.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character. Consider the following commands:

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with *x*, the second prints the values of all two letter variables in function *sub* beginning with *y*, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**:*/
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b	One byte
h	Two bytes (half word)
l	Four bytes (long word).

The lengths are only effective with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed. There are a number of format specifiers available:

c	Character
d	Decimal
u	Decimal unsigned
o	Octal
x	Hexadecimal
f	32-bit single-precision floating point
g	64-bit double-precision floating point
s	Assume variable is a string pointer and print characters until a null is reached
a	Print characters starting at the variable's address until a null is reached
p	Pointer to function
i	Interpret as a machine-language with addresses printed symbolically
I	Interpret as a machine-language with addresses printed symbolically.

For example, the variable *i* can be displayed with

```
*i/x
```

which prints out the value of *i* in hexadecimal.

The **sdb** program also knows about structures, arrays, and pointers so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Depending on your machine, accessing arrays may be limited to 1-dimensional arrays. Note that as a special case:

```
*psym->/d
```

displays the location pointed to by *psym* in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both

```
*02000/
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000->x/, the sdb program uses the structure template of the last structured referenced.

The address of a variable is printed with the = p, so

```
*i=
```

displays the address of *i*. Another feature whose usefulness will become apparent later is the command

```
*./
```

which displays the last variable typed again.

C. Source File Display and Manipulation

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those of the UNIX system text editor **ed**(1). Like the editor, **sdb** has a notion of current file and line within the file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

p	Prints the current line.
w	Window; prints a window of ten lines around the current line.
z	Prints ten lines starting at the current line. Advances the current line by ten.
control–d	Scrolls; prints the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file but is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The e command is used to change the current source file. Either of the forms

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an e command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The z and **control–d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing/ and ? may be omitted from these commands. Regular expression matching is identical to that of **ed**(1).

The + and – commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints 10 lines.

D. A Controlled Environment for Program Testing

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; and then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. The **sdb** program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. Note that if an attempt is made to single step through a function which has not been compiled with the **−g** option execution proceeds until a statement in a function compiled with the **−g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **−g** option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function *proc*, and the third sets a breakpoint at the first line of *proc*. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the commands

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The a command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called. and the second prints the source line each time it is about to be executed. For both forms of the a command, execution continues after the function name or source line is printed.

Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the **shell**. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the **shell** command line. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to **sdb**.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command which continues but passes the signal which stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

continues at line 17 of the current function. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a function different than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal which stopped the program. Its uses are similar to the **s** command. There is also an **I** command which causes the program to execute one machine level instruction at a time, but also passes the signal which stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a function which prints structured data in a nice way. There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or values of variables which are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function which formats data from a dump.

E. Machine Language Debugging

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function *main*, use the command

```
*main:25?
```

The ? command is identical to the / command except that it displays from text space. The default format for printing text space is the i format which interprets the machine language instruction. The **control-d** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a : to them so that

```
*0x1024:?
```

displays the contents of address *0x1024* in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line *0x1024* in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address *0x1024*.

Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named instead of variables by appending a % to their name so that

```
*r3%
```

displays the value of register *r3*.

F. Other Commands

To exit **sdb**, use the **q** command.

The ! command is identical to that in **ed(1)** and is used to have the **shell** execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
```

which sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type float or double, the value can also be a floating-point constant.

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
    int i;
    i = div2(-1);
    printf( " -1/2 = %d\n" , i);
}
div2(i) {
    int j;
    j = i>>1;
    return(j);
}
$ cc -g testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image          # Warning message from sdb
*/^div2                # Search for function " div2"
7: div2(i) {           # It starts on line 7
*z                     # Print the next few lines
7: div2(i) {
8:     int j;
9:     j = i>>1;
10:     return(j);
11: }
*div2:b                # Place a breakpoint at the beginning of " div2"
div2:9 b               # Sdb echoes proc name and line number
*r                     # Run the function
a.out                  # Sdb echoes command line executed
Breakpoint at          # Executions stops just before line 9
div2:9:     j = i>>1;
*t                     # Print trace of subroutine calls
div2(i=-1)    [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)     [testdiv2.c:4]
*i/                    # Print i
-1
*s                     # Single step
div2:10:    return(j);    # Execution stops just before line 10
*j/                    # Print j
-1
*9d                    # Delete the breakpoint
*div2(1)/              # Try running " div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

Fig. 3.1—Example of sdb Usage

4. FORTRAN

INTRODUCTION

This section describes the implementation of the Fortran programming language on the UNIX operating system. This section contains the following parts:

- **FORTRAN 77**—Describes the implementation of Fortran 77 on the system in terms of the variations from the American National Standard.
- **A RATIONAL FORTRAN PREPROCESSOR—"ratfor"**—A preprocessor which provides a means for writing Fortran in a fashion similar to C language. This preprocessor provides (among other things) simplified control-flow statements.

Both parts assume that the user is already familiar with Fortran 77.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

FORTRAN 77

A. General

This part describes the compiler and run-time system for Fortran 77 as implemented on the UNIX operating system. Fortran 77 became an official American National Standard on April 3, 1978. The implementation of Fortran 77 on the UNIX operating system varies from the American National Standard. This document describes the difference between the American National Standard and the current implementation. Also, this document describes the interfaces between procedures and the file formats assumed by the I/O system.

Usage

The command to run the compiler is

```
f77 options file
```

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files. Ratfor source files will be preprocessed before being presented to the Fortran compiler. C language and assembler source files will be compiled by the appropriate programs. Object files will be loaded. [The **f77**(1) and **cc**(1) commands cause slightly different loading sequences to be generated since Fortran programs need a few extra libraries and a different startup routine than do C language programs.] The following file name suffixes are understood:

.f	Fortran source file
.r	Ratfor source file
.c	C language source file
.s	Assembler source file
.o	Object file

The following flags are understood:

−S	Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f**, **x.r**, or **x.c** is put on file **x.s**.
−c	Compile but do not load. Output for **x.f**, **x.r**, **x.c**, or **x.s** is put on file **x.o**.
−m	Apply the M4 macro preprocessor to each Ratfor source file before using the appropriate compiler.
−f	Apply the Ratfor processor to all relevant files and leave the output from **x.r** on **x.f**. Do not compile the resulting Fortran program.
−p	Generate code to produce usage profiles.
−of	Put executable module on file f. (Default is **a.out**).
−w	Suppress all warning messages.
−w66	Suppress warnings about Fortran 66 features used.
−O	Invoke the C language object code optimizer.
−C	Compile code the checks that subscripts are within array bounds.
−onetrip	Compile code that performs every **do** loop at least once.
−U	Do not convert uppercase letters to lowercase. The default is to convert Fortran programs to lowercase.
−u	Make the default type of a variable **undefined**.
−I2	On machines which support short integers, make the default integer constants and variables short. (−I4 is the standard value of this option.) All logical quantities will be short.
−R	The remaining characters in the argument are used as a Ratfor flag argument.
−F	Ratfor and source programs are preprocessed into Fortran files, but those files are not compiled or removed.

All library names (arguments beginning —l) and other options not ending with one of the special suffixes are passed to the loader. See f77(1) for additional options.

B. Language Extensions

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.

Double Complex Data Type

The data type double complex is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every complex built-in function is provided. The specific function names begin with z instead of c.

Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

Implicit Undefined Statement

Fortran has a fixed rule that the type of a variable that does not appear in a type statement is integer if its first letter is *i, j, k, l, m* or *n*. Otherwise, it is real. Fortran 77 has an implicit statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **−u** compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures may call themselves directly or through a chain of other procedures.

Automatic Storage

Two new keywords recognized are **static** and **automatic**. These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence**, **data**, or **save** statements.

Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (&) in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters—Fortran is a 1-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the −U compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

Include Statement

The statement

```
        include "stuff"
```

is replaced by the contents of the file *stuff.* Includes may be nested to a reasonable depth, currently ten.

Binary Initialization Constants

A **logical**, **real**, or **integer** variable may be initialized in a data statement by a binary constant denoted by a letter followed by a quoted string. If the letter is *b*, the string is binary, and only zeroes and ones are permitted. If the letter is *o*, the string is octal with digits *zero* through *seven.* If the letter is z or x, the string is hexadecimal with digits *zero* through *nine*, *a* through *f.* Thus, the statements

```
        integer a(3)
        data a/b'1010',o'12',z'a'/
```

initialize all three elements of a to ten.

Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

\n	new-line
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
\x	where x is any other character.

Fortran 77 only has one quoting character—the apostrophe ('). This compiler and I/O system recognize both the apostrophe and the double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an integer word boundary. Each character string constant appearing outside a data statement is followed by a null character to ease communication with C language routines.

Hollerith

Fortran 77 does not have the old Hollerith (**nh**) notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants and may also be used to initialize noncharacter variables in data statements.

Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **−onetrip** compiler option causes nonstandard loops to be generated.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of noncharacter variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
−345,.05e−3,12
```

correctly.

Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer*2**. (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C language type **long int;** half word integers are of C language type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **−I2** flag, all small integer constants will be of type **integer*2**. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **−I2** command flag is in effect). When the **−I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the command arguments (**getarg** and **iargc**).

C. Violations of the Standard

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.

Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370) run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. This requirement arises because of the way character string arguments are represented and of the 1-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external**. Code is correct if there are no **character** arguments.

T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

D. Interprocedure Interface

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representations

The following is a table of corresponding Fortran and C language declarations:

Fortran	**C Language**
integer*2 x	short int x;

integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r,i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

By the rules of Fortran, **integer**, **logical**, and **real** data occupy the same amount of memory.

Return Values

A function of type **integer**, **logical**, **real**, or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
  . . .
```

A character-valued function is equivalent to a C language routine with two extra initial arguments—a data address and a length. Thus,

```
character*15 function g( . . : )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
  . . .
```

and could be invoked in C language by

```
char chars[15];
  . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
        goto (1, 2, 3),   nret( )
```

Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

> Extra arguments for complex and character functions
> Address for each datum or function
> A **long int** for each character or procedure argument

Thus, the call in

```
        external f
        character*7 s
        integer b(3)
          . . .
        call sam(f, b(2), s)
```

is equivalent to that in

```
        int f( );
        char s[7];
        long int b[3];
          . . .
        sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

E. File Formats

Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records". When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77

American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist nor will they be created unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

A RATIONAL FORTRAN PREPROCESSOR—"ratfor"

A. General

This part describes the **ratfor(1)** preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX operating system.

The **ratfor** language allows users to write Fortran programs in a fashion similar to C language. The **ratfor** program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by **ratfor** are:

- statement grouping
- if-else and switch for decision making
- while, for, do, and repeat-until for looping
- break and next for controlling loop exits
- free form input such as multiple statements/lines and automatic continuation
- simple comment convention
- translation of >, >=, etc., into .gt., .ge., etc.
- return statement for functions
- define statement for symbolic parameters
- include statement for including source files.

B. Usage

The **ratfor** program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **−6x**, which uses x as a continuation character in column 6 (the UNIX system uses *&* in column 1), and −C, which causes ratfor comments to be copied into the generated Fortran.

The program rc(1M) provides an interface to the **ratfor(1)** command. This command is similar to cc(1). Thus:

```
rc options files
```

compiles the files specified by *files*. Files with names ending in **.r** are **ratfor** source; other files are assumed to be for the loader. The options −C and **−6x** described above are recognized, as are

−c	Compile only; don't load
−f	Save intermediate Fortran .f files
−r	Ratfor only; implies −c and **−f**
−2	Use big Fortran compiler (for large programs)
−U	Flag undeclared variables (not universally available).

Other options are passed on to the loader.

C. Statement Grouping

Fortran provides no way to group statements together short of making them into a subroutine. The **ratfor** language does provide a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the **ratfor** code

```
if (x > 100)
    { call error(" x>100" ); err = 1; return }
```

will be translated by the **ratfor** preprocessor into **Fortran** equivalent to

```
        if (x .le. 100) goto 10
            call error(5hx>100)
            err = 1
            return
10          ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous **ratfor** example that the character > was used instead of **.GT.** in the **if** statement. The **ratfor** preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like "*x> 100*"). Quotes are not allowed in ANSI Fortran, so **ratfor** converts it into the right number of *H*s.

The **ratfor** language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error("x>100" )
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because **ratfor** assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

D. The "if-else" Construction

The **ratfor** language provides an **else** statement. The syntax of the **if-else** construction is:

```
if (legal Fortran condition)
        ratfor statement
else
        ratfor statement
```

where the **else** part is optional. The legal Fortran condition is anything that can legally go into a Fortran Logical IF statement. The **ratfor** preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The ratfor statement is any **ratfor** or Fortran statement or any collection of them in braces. For example,

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

is a valid **ratfor if-else** construction. This writes out the smaller of a and b, then the larger, and sets sw appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

Nested "if" Statements

The statement that follows an **if** or an **else** can be any **ratfor** statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in **ratfor**. (The **ratfor** language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In **ratfor** when there is more if statements than else statements, else statements are associated with the closest previous if statement that currently does not have an associated else statement. For example:

```
if (x > 0)
if(y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the **ratfor** preprocessor as

```
if (x > 0) {
      if (y > 0)
            write(6, 1) x, y
      else
            write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired, it must be written as

```
if (x > 0) {
      if (y > 0)
            write(6, 1) x, y
}
else
      write(6, 2) y
```

with the braces specified.

E. The "switch" Statement

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

```
switch (expression) {
      case expr1 :
      statements
      case expr2, expr3 :
      statements
      ...
      default:
      statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match *expression*, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** is executed, the entire **switch** is exited immediately. This is somewhat different than C language.

F. The "do" Statement

The do statement in **ratfor** is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor** do statement is

```
        do legal-Fortran-DO-text {
            ratfor statements

        }
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a **ratfor** do statement. The *ratfor statements* are enclosed in braces; but as with the if, a single statement need not have braces around it. For example, the following code sets an array to 0:

```
        do i = 1, n
           x(i) = 0.0
```

and the code

```
        do i = 1, n
           do j = 1, n
              m(i, j) = 0
```

sets the entire array *m* to zero.

G. The "break" and "next" Statements

The **ratfor break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
        do i = 1, n {
            if (x(i) < 0.0)
                next
            process postive element
        }
```

The **break** and **next** statements will also work in the other **ratfor** looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
        break 2
```

exits from two levels of enclosing loops, and

```
        break 1
```

is equivalent to **break**. The

```
        next 2
```

iterates the second enclosing loop.

H. The "while" Statement

The **ratfor** language provides a **while** statement. The syntax of the **while** statement is

```
while (legal-Fortran-condition)
    ratfor statement
```

As with the if, legal-Fortran-condition is something that can go into a Fortran Logical IF, and ratfor statement is a single statement which may be multiple statements enclosed in braces.

For example, suppose nextch is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
        ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, ich contains the first nonblank.

I. The "for" Statement

The **for** statement is another **ratfor** loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
    ratfor statement
```

where *init* is any single Fortran statement which is executed once before the loop begins. The increment is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a Fortran Logical IF. Any of init, condition, and increment may be omitted although the semicolons must always be present. A nonexistent condition is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran DO loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for** and **while** versions have the advantage that they will be done zero times if *n* is less than 1. This is not true of the **do**. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a for need not be an arithmetic progression. The program

```
        sum = 0.0
        for (i = first; i > 0; i = ptr(i))
                sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

J. The "repeat-until" Statement

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

> **repeat**
>   *ratfor statement*
> **until** (*legal-Fortran-condition*)

where *ratfor-statement* is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a READ statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

K. The "return" Statement

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
        equal = 0
        return
```

cause *equal* to return zero.

The **ratfor** language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

> **return** (*expression*)

were *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

L. The "define" Statement

The **ratfor** language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

> **define** name value

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
      if (i > ROWS   |   j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

M. The "include" Statement

The **ratfor** language provides an **include** statement similar to the **#include** <...> statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the **ratfor** input file in place of the **include** statement. The standard usage is to place COMMON blocks on a file and use the **include** statement to include the common code whenever needed.

N. Free-Form Input

In **ratfor**, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if **ratfor** can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
=  +  -  *  ,  |  &  (  _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

```
write(6, 100); 100 format("hello" )
```

is converted into

```
          write(6, 100)
100       format(5hhello)
```

O. Translations

Text enclosed in matching single or double quotes is converted to *nH*... but is otherwise unaltered (except for formatting—it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
"\\\""
```

is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

==	.eq.
!=	.ne.
>	.gt.
>=	.ge.
<	.lt.
<=	.le.
&	.and.
|	.or.
!	.not.
¬	.not.

In addition, the following translations are provided for input devices with restricted character sets:

[	{
]	}
$(	{
$)	}

P. Warnings

The **ratfor** preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the **ratfor** code. This makes it difficult to locate **ratfor** statements that contain errors.

The keywords are deserved. Using **if**, **else**, **while**, etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic IF will cause problems.

The Fortran *n*H convention is not recognized by **ratfor**. Use quotes instead.

NOTES