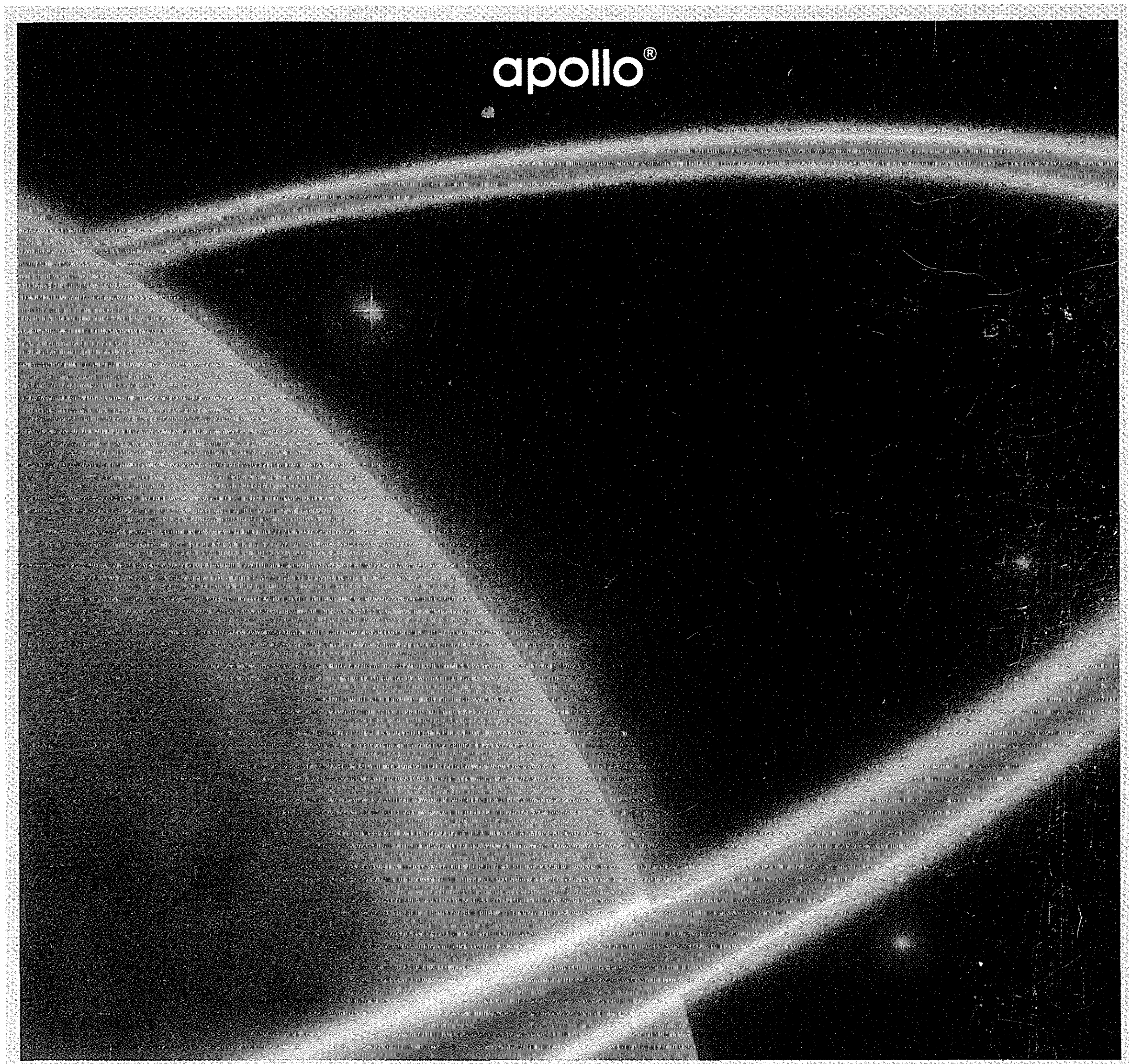
apollo®



D O M A I N

Order No. 009413
Revision 00

# DOMAIN/IX Support Tools Guide

# DOMAIN/IX Support Tools Guide

Order No. 009413
Revision 00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

# Preface

## Audience

The *DOMAIN/IX Support Tools Guide* consists of papers normally included in certain volumes of the *UNIX Programmer's Manual* as supplied by AT&T and the University of California at Berkeley. The papers were revised as necessary to reflect the DOMAIN® system environment. However, to help maintain the history of the UNIX® product as a multiuser system, we've included the more important references to operations conducted at terminals.

The *Support Tools Guide* is intended for users who are already familiar with UNIX software, AEGIS™ software, and DOMAIN networks.

The best introduction for those who want to use UNIX software on a DOMAIN node is *Getting Started With Your DOMAIN/IX System* (Order No. 008017). It explains how to use your keyboard and display, read and edit text, create and execute programs, and request DOMAIN system services interactively. Consult the *DOMAIN/IX User's Guide* (Order No. 005803) for detailed information on user interfaces and the various shells available for use.

## Structure of This Manual

This manual is structured as follows:

| | |
|---|---|
| **Chapter 1** | Introduces **awk**, a pattern scanning and processing language designed to make many common information retrieval and text manipulation tasks easy to state and to perform. |
| **Chapter 2** | Describes **sed**, the UNIX stream editor. |
| **Chapter 3** | Explains how to use **lint**, a C program checker. (This chapter is based on a 1978 AT&T Bell Laboratories memo by S. C. Johnson.) |
| **Chapter 4** | Describes **make**, a program for maintaining, updating, and regenerating groups of computer programs. (This chapter is based on an original technical report by S. I. Feldman of AT&T Bell Laboratories.) |
| **Chapter 5** | Tells about the System V extensions to the **make** program, designed to handle problems within the original version of **make**. |
| **Chapter 6** | Details **lex**, a lexical analyzer which processes character input streams. |
| **Chapter 7** | Describes **yacc**, a general tool for imposing structure on the input to a computer program |
| **Chapter 8** | Explains how to use **sccs**, the UNIX source code control system. |
| **Chapter 9** | Contains C language reference material. (This chapter was based on a section of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1978.) |

| | |
|---|---|
| **Chapter 10** | Tells about **ratfor**, a preprocessor for a rational FORTRAN. (This chapter is based on a paper written by Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.) |
| **Chapter 11** | Describes the **m4** macro processor. (This chapter is based on a paper written by Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories, Murray Hill, New Jersey.) |
| **Chapter 12** | Details **bc**, a compiler for doing arbitrary precision arithmetic. |
| **Chapter 13** | Provides information about **dc**, an interactive desk calculator that does arbitrary precision integer arithmetic. |
| **Chapter 14** | Describes the **curses** screen package, which provides for movement optimization and optimal screen updating. (This chapter is based on a paper written by Kenneth C. R. C. Arnold, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California - Berkeley.) |
| **Chapter 15** | Describes *bsd4.2* interprocess communication facilities. (This chapter is based on a July 9, 1984 draft of *A 4.2BSD Interprocess Communication Primer*, by S. J. Leffler, R. S. Fabry, and W. N. Joy of the Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California - Berkeley. |

## Related Manuals

*Getting Started With Your DOMAIN/IX System* (Order No. 008017) is the first volume you should read. It explains how to log in and out, manage windows and pads, and execute simple commands.

The *DOMAIN/IX User's Guide* (Order No. 005803) describes how the DOMAIN/IX system works, and contains extensive material on the C Shell, both versions of the Bourne Shell, and the *bsd4.2* version of the **mail** program.

The *DOMAIN/IX Text Processing Guide* (Order No. 005802) describes the UNIX text editors (**ed**, **ex**, and **vi**) supported by DOMAIN/IX. It also contains material on the formatters **troff** and **nroff**, the macro packages **-ms**, **-me**, and **-mm**, and the preprocessors **eqn** and **tbl**.

The *DOMAIN/IX Command Reference for System V* (Order No. 005798) describes all the UNIX System V shell commands supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005799) describes all the UNIX System V system calls and library functions supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Command Reference for BSD4.2* (Order No. 005800) describes all the BSD4.2 UNIX shell commands supported by the *bsd4.2* version of DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005801) describes all the BSD4.2 UNIX system calls and library functions supported by the *bsd4.2* version of DOMAIN/IX.

*System Administration for DOMAIN/IX BSD4.2* (Order No. 009355) and *System Administration for DOMAIN/IX Sys5* (Order No. 009356) describe the tasks necessary to configure and maintain DOMAIN/IX system software services such as TCP/IP, the line printer spoolers, and UNIX–to–UNIX communications processing. Also explains how to maintain file system security, create user accounts, and manage various servers and daemons.

The *DOMAIN C Language Reference* (Order No. 002093) describes C program development on the DOMAIN system. It lists the features of C, describes the C library, and gives information about compiling, binding, and executing C programs.

The *DOMAIN System Command Reference* (Order No. 002547) gives information about using the DOMAIN system and describes the DOMAIN commands.

The *DOMAIN System Call Reference* (Order No. 007196) describes calls to operating system components that are accessible to user programs.

## Documentation Conventions

Unless otherwise noted in the text, we use the following symbolic conventions;

| | |
|---|---|
| **command** | Command names and command–line options are set in Classic font, bold type. These are commands, letters, or symbols that you must use literally. |
| `output` | Typewriter font is used to represent literal system output. |
| program line | Modern font is used to show lines that may be part of a program, non–literal characters or strings in an example, and all other sample information not attributed to being a literal command line or display of system output. |
| *filename* | Italicized terms or characters represent generic, or metanames in example command lines. They may also represent characters that stand for other characters, as in **d***x*, where *x* is a digit. In text, the names of files written or read by programs are set in italics. |
| [ ] | Square brackets enclose optional items in formats and command descriptions. |
| \| | A vertical bar separates items in a list of choices. |
| < > | Angle brackets enclose the name of a key on the keyboard. |
| ↑D | The notation ↑ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while typing the character. |
| ... | Horizontal ellipsis points indicate that the preceding item can be repeated one or more times. |
| ⋮ | Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted. |

## Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software–related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. You can view an on-line description of the command used to submit a UCR (**crucr**) by typing:

```
% /com/help crucr  <RETURN>
```

**Note:** Although we use a C Shell prompt in our example, you may type this command from any type of shell available to users of the DOMAIN/IX system.

For your documentation comments, we've included a Reader's Response form at the back of each manual.

# Contents

Chapter 1

# Awk: A Pattern Scanning and Processing Language

## 1.1 Introduction

Awk is a programming language that lets you prepare programs that search a file or set of files for patterns, then perform actions on lines or parts of lines that contain instances of those patterns. **Awk** makes certain data selection and transformation operations easy to express; for example, the following very simple **awk** program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary Boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern–matching constructions as in patterns, as well as arithmetic and string expressions and assignments, if–else, while, and for statements, and multiple output streams.

This chapter explains how to write **awk** programs. It also includes a discussion of the design and implementation of **awk**, for insight into the way UNIX software development tools can be combined to produce programs for specific tasks.

## 1.2 Overview

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

When invoked, **awk** scans a set of input lines (usually from a specified *file*) in order, searching for instances of *patterns* specified in the *program*. For any pattern, you can specify an *action* to be performed on each line that matches the pattern.

Readers familiar with the UNIX program **grep**(1) recognize the approach, although patterns in **awk** may be more general than in **grep**. Also, while **grep** allows only one action (print the line), **awk** provides you with a variety of actions that may be taken on all or part of a line in which the matching pattern occurs. For example, in **awk**,

```
{print $3, $2}
```

prints the third and second fields of an input line in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev      { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

### 1.2.1 Usage

The command line

```
% awk 'program' [input_file(s)] <RETURN>
```

executes the **awk** commands in the *program* string on the named *input_file(s)*.

**Note:** When you include the **awk** program in the command line, it must be delimited by single quotes (as shown above) so that the shell knows that the entire program is the first argument to **awk**.

As is the case with other UNIX programs, **awk** reads the standard input if no *file* is specified, or if a dash (–) is specified in place of a filename, as shown here:

```
% awk 'program' – <RETURN>
```

If *program* is more than a few statements long, you may want to place it in a file and execute it by including the –f option on the **awk** command line, as shown here:

```
% awk –f program_file input_file(s) <RETURN>
```

### 1.2.2 Program Structure

An **awk** program is a sequence of statements of the form:

```
pattern   { action }
pattern   { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all specified patterns have been tested against the contents of the first input line, the next line is fetched and the matching process starts again.

Either the pattern or the action may be left out of an **awk** program line, but not both. If there is no action for a pattern, **awk** simply copies all matching input line(s) to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

### 1.2.3 Records and Fields

Awk divides each input file into records terminated by a record separator. The default record separator is the newline, so by default **awk** processes its input a line at a time. The number of the current record is available in a variable named *NR*.

Each input record is considered to be divided into fields. These are normally separated by white space (blanks or tabs), although the input field separator may be changed to any other character by resetting the *FS* variable as described below. Fields are referred to as $1, $2, and so forth, where $1 is the first field, $2 is the second field, and $0 is the entire input record. Fields may be assigned to a numeric or string value. The number of fields in the current record is available in a variable named *NF*.

The variables *FS* and *RS* refer to the input field and record separators. These may be changed to another (single) character at any time. The optional command–line argument –F*c* may also be used to set *FS* to a character represented here by *c*.

If the record separator is empty, an empty input line is taken as the record separator. Blanks, tabs, and newlines are then treated as field separators.

The variable *FILENAME* contains the name of the current input file.

### 1.2.4 Printing

If an action has no pattern, the action is executed for all input lines (records). The simplest action is provided by the **awk** command **print**, which prints some or all of a record. The simple **awk** program

```
{ print }
```

prints each input record. It merely copies the input to the output – something to which **cat**(1) is far better suited. A more useful **awk** program might print a field or possibly selected fields from each record. For instance, the program

```
{ print $2, $1 }
```

prints the first two fields of each input record (since no pattern has been specified) in reverse order. Items separated by a comma in the **print** statement are separated by the current output field separator when printed. Items not separated by commas are concatenated, so this runs the first and second fields together:

```
print $1 $2
```

The predefined numeric variables, *NF* (Number of Fields) and *NR* (Number of Records), have many uses. For example, the program

```
{ print NR, NF, $0 }
```

prints each record preceded by its record number and the number of fields it contains.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field on the file *foo1* and the second field on file *foo2*. The >> notation can be used to append **awk** output to a file. Thus, the program

```
{ print $1 >>"foo" }
```

appends the first field of every input record to the file *foo*.

**Note:** When printing or appending output to a file, **awk** creates the specified output file if it does not already exist.

The filename can be derived from a variable or a field, as well as a constant. Thus, this uses the contents of field 2 of the current input record as the output filename:

```
{ print $1 >$2 }
```

**Note:** You may not specify more than 10 output files in an **awk** program.

**Awk** output can also be piped into another process; for instance, this mails the current input record to mail user *bob*:

```
{ print | "mail bob" }
```

The variables *OFS* and *ORS* may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

**Awk** also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in *format* and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8–digits wide, with two after the decimal point, and $2 as a 10–digit long decimal number followed by a newline. Output separators are not produced automatically; you must add them yourself, as in this example. The **awk** version of **printf** is identical to that used in the C programming language.

## 1.3 Patterns

A pattern to the left of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string–valued expressions, and arbitrary Boolean combinations of all three.

### 1.3.1 The BEGIN and END Patterns

The special pattern *BEGIN* matches the beginning of the input, before the first record is read. The special pattern *END* matches the end of the input, after the last record has been processed. *BEGIN* and *END* provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN     { FS = ":" }
... body of program ...
```

This line finishes an **awk** program by displaying a count of input lines:

```
END  { print NR }
```

**Note:** If *BEGIN* is used, it must be the first pattern; *END*, is used, must be the last.

## 1.3.2 Regular Expressions

The simplest regular expression is a literal string of characters delimited by slashes:

```
/smith/
```

This is actually a complete **awk** program which prints all lines which contain any occurrence of the name *smith*. Lines that contain *smith* as part of a larger word (e.g., blacksmithing) are also printed.

**Awk** regular expressions include the regular expression forms found in the UNIX text editor **ed**(1) as well as those used by **grep**(1) (without back-referencing). In addition, **awk** allows parentheses for grouping, the vertical line (|) to separate alternatives, a plus sign (+) for "one or more", and a question mark (?) for "zero or one". All of these usages should be familiar to **lex**(1) users. Character classes may be abbreviated:

```
[a-zA-Z0-9]
```

matches the set of all letters and digits. As an example, the brief **awk** program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

prints all lines which contain any of the names *Aho*, *Weinberger*, or *Kernighan*, whether or not the first letter of the name is capitalized.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in **ed**(1) and **sed**(1). Within a regular expression, blanks and the regular expression metacharacters are significant. To escape a regular expression character and restore its "real" meaning, precede it with a backslash. The pattern

```
/\/.*\//
```

matches any string of characters enclosed in slashes.

You can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. This program prints all lines where the first field matches *john* or *John*:

```
$1 ~ /[jJ]ohn/
```

Note that this also matches *Johnson*, *St. Johnsbury*, and so on. To restrict it to exactly *[jJ]ohn*, use

```
$1 ~ /^[jJ]ohn$/
```

The caret (^) refers to the beginning of a line or field, and the dollar sign ($) the end.

### 1.3.3 Relational Expressions

An **awk** pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. For example,

```
$2 > $1 + 100
```

selects lines where the second field is at least 100 greater than the first field. Likewise,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise, a numeric comparison is made. Thus,

```
$1 >= "s"
```

selects lines that begin with an *s*, *t*, *u*, etc. In the absence of any other information, fields are treated as strings, so this program performs a string comparison:

```
$1 > $2
```

### 1.3.4 Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with *s*, but is not *smith*. The && and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

### 1.3.5 Pattern Ranges

The pattern that selects an action may also comprise two patterns separated by a comma, as in

```
pat1, pat2     { ... }
```

In this case, the action is performed for each line starting at an occurrence of *pat1* and ending at the first subsequent occurrence of *pat2* (inclusive). For example,

```
/start/, /stop/
```

prints all lines between *start* and *stop*, while this does the action for lines 100 through 200 of the input:

```
NR == 100, NR == 200 { ... }
```

## 1.4 Actions

An **awk** action is a sequence of one or more action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks, many of which are described in this section.

### 1.4.1 Built-in Functions

**Awk** provides a length function to compute the length of a string of characters. The program below prints each record preceded by its length:

```
{ print length, $0 }
```

By itself, **length** is a "pseudo-variable" that yields the length of the current record. With an argument (any expression), it becomes a function that yields the length of its argument, as in the following:

```
{ print length($0), $0 }
```

**Awk** also provides the arithmetic functions **sqrt** (square root, base *e*), **log** (logarithm), **exp** (exponential), and **int** (integer).

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. This program prints lines whose length is less than 10 or greater than 20:

```
length < 10 || length > 20
```

The function **substr(s, m, n)** produces the substring of *s* that begins at position *m* (origin 1) and is at most *n* characters long. If *n* is omitted, the substring goes to the end of *s*. The function **index(s1, s2)** returns the position where the string *s2* occurs in *s1*, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions *e1*, *e2*, etc., in the **printf** format specified by *f*. Thus,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets *x* to the string produced by formatting the values of *$1* and *$2*.

### 1.4.2 Variables, Expressions, and Assignments

**Awk** variables assume numeric (floating point) or string values according to context. Thus, in

```
x = 1
```

*x* is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns the value 7 to *x*. Strings which can't be interpreted as numbers in a numerical context generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most *BEGIN* sections. For example, the sums of the first two fields can be computed by

```
        { s1 += $1; s2 += $2 }
END     { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are: +, -, *, /, and % (mod). The C increment (++) and decrement (--) operators are also available, as are the assignment operators +=, -=, *=, /=, and %=. You may use all these operators in expressions.

### 1.4.3 Field Variables

Fields in **awk** share essentially all of the properties of variables; they may be used in arithmetic or string operations, and may be assigned to a numeric or string value. **Awk** lets you, for example, replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if     ($3 > 1000)
          $3 = "too big"
  print
}
```

which replaces the third field by the string "too big" when the field exceeds an arbitrary size (in this case, 1000 characters), then prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context. In ambiguous cases such as this,

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string *s* into *array(1), ..., array(n)*. **Awk** returns a value indicating the number of elements found. If the *sep* argument is provided, it is used as the field separator; otherwise *FS* is used as the separator.

### 1.4.4 String Concatenation

Strings may be concatenated, e.g., this returns the length of the first three fields:

```
length($1 $2 $3)
```

In a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by *is*. Variables and numeric expressions may also appear in concatenations.

### 1.4.5 Arrays

Array elements are not declared; they are created as necessary. Subscripts may have any non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the *NR*-th element of the array *x*. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the **awk** program

```
      { x[NR] = $0 }
END   { ... program ... }
```

The first action merely records each input line in the array *x*.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like *apple*, *orange*, etc. Then the program

```
/apple/      { x["apple"]++ }
/orange/     { x["orange"]++ }
END          { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

Any expression can be used as a subscript in an array reference. Thus,

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array *x*.

Suppose each line of input contains two fields, a name and a non-zero value. Names may be repeated. To print a list of each unique name followed by the sum of all the values for that name, you could use this program:

```
      { amount[$1] += $2 }
END   { for (name in amount)
      { print name, amount[name] }
```

To sort the output, replace the last line by

```
print name, amount[name] | "sort"
```

### 1.4.6 Flow-of-Control Statements

**Awk** provides these flow-of-control statements: **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement earlier without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line, specify this:

```
i = 1
while (i <= NF) {
     print $i
     ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
     print $i
```

does the same job as the **while** statement above.

An alternate form of the **for** statement is suited for accessing the elements of an associative array. Thus,

```
for (i in array)
     statement
```

does *statement* with **i** set in turn to each element of *array*. The elements are accessed in an apparently random order. Problems develop if the variable *i* is altered, or any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while**, or **for** can include relational operators such as <, <=, >, >=, == ("is equal to"), and != ("not equal to"); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and, of course, parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes **awk** to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in **awk** programs: they begin with the pound sign (#) and end with the end of the line, as in

```
print x, y       # this is a comment
```

## 1.5 Design

The UNIX system provides several programs that operate by passing input through a selection mechanism. The **grep**(1) program, one of the simplest, merely prints all lines which match a single specified pattern. **Egrep**(1) provides more general patterns, i.e., regular expressions in full generality; and **fgrep**(1) searches for a set of keywords with a particularly fast algorithm. The stream editor **sed**(1) applies most of the editing facilities of the editor **ed**(1) to a stream of input. None of these programs provide numeric capabilities, logical relations, or variables.

**Lex**(1) provides general regular expression recognition capabilities. By serving as a C program generator, it is essentially open-ended in its capabilities. To use **lex**, however, you need a knowledge of C programming. Furthermore, you have to compile and load a **lex** program before using it; this discourages its use for one-time applications.

**Awk** provides general regular expression capabilities and an implicit input/output loop. It also supplies convenient numeric processing, variables, more general selection, and control flow in the actions. **Awk** doesn't require compilation, nor does it presuppose extensive knowledge of C. Finally, it provides a convenient way to access fields within lines; it is unique in this respect.

**Awk** also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases, you can simply ignore the differences.

Most of the development effort applied to **awk** went into deciding what it should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation), rather than on writing or debugging the code. The authors of the program (A. V. Aho, P. J. Weinberger, and B. W. Kernighan) tried to make the syntax powerful, easy to use, and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, **awk** usage seems to fall into two broad categories. One area of use is "report generation", or processing of an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is data transformation, that is, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

The actual implementation of **awk** uses several of the UNIX language development tools discussed in other chapters of this manual. The grammar is specified with **yacc**(1), and the lexical analysis is done by **lex**(1). The regular expression recognizers are deterministic, finite automata constructed directly from the expressions. An **awk** program is translated into a parse tree, which is then directly executed by a simple interpreter.

Chapter 2

# Sed: The Stream Editor

## 2.1 Introduction

**Sed**(1) is a non–interactive context editor designed to be especially useful for:

- Editing files too large for comfortable interactive editing
- Editing a file of any size where the sequence of editing commands is too complicated to be comfortably typed in interactive mode
- Doing multiple "global" editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in real memory at one time, and no temporary files are used, the effective size of a file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to **sed** as a command file. This often saves considerable typing, and provides a way to make special–purpose filters based on **sed**.

The principal losses of functionality in **sed**, as compared with an interactive editor, are lack of relative addressing (because of the line–at–a–time operation), and lack of immediate verification that a command has done what was intended.

**Sed** is a lineal descendant of the UNIX line editor, **ed**(1). Because of differences between interactive and non–interactive operation, **sed** represents considerable advancement over **ed**. Even experienced **ed** users should read this chapter before trying to use **sed**. The most striking family resemblance between the two editors is in the class of patterns ("regular expressions") that they recognize. The code for matching patterns that **sed** uses is copied almost verbatim from the code for **ed**, so the two programs behave identically in this respect.

## 2.2 Normal Operation

By default, **sed** copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line.

The general format of an editing command is:

```
[address1,address2] [function] [arguments]
```

You may omit one or both addresses. We show the format of addresses in the next section of this chapter. Any number of blanks or tabs may separate the addresses from the function. The function must be present. The arguments may be required or optional, depending on the function. Functions and arguments are discussed later.

Tab characters and spaces at the beginning of lines are ignored.

### 2.2.1 Command Line Flags

**Sed** recognizes three command line flags:

| | |
|---|---|
| **–n** | Copy only those lines specified by *p functions* or *p flags* after *s functions.* |
| **–e** | Take the next argument as an editing command. |
| **–f** *name* | Get commands to be used by **sed** from file *name*. *Name* must be a file that contains editing commands, one to a line. |

### 2.2.2 Order of Application of Editing Commands

Before any editing is done or any input file opened, all the editing commands given to **sed** are compiled into a form that is moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they are attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands **t** and **b**. However, the input line to any command is always the output of any previously applied command.

## 2.3 The Pattern Space

The range of pattern matches is called the pattern space (usually, one line of the input text). You must use the N command to request that more than one line to be read into the pattern space. Beginning with this section, we supply several examples. Except where otherwise noted, all the examples assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

The command

**2q** <RETURN>

quits after copying the first two lines of the input. The output is:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

## 2.4 Addresses – Selecting Lines For Editing

You can select lines in the input file(s) to which editing commands are to be applied by addresses. Addresses may be either line numbers or context addresses.

To control the application of a group of commands by one address (or address-pair), group the commands in braces as shown here:

{*commands*}

### 2.4.1 Line–Number Addresses

A line number is a decimal integer. As each line is read from the input, a line–number counter is incremented; a line–number address matches (selects) the input line that causes the internal counter to equal the address line–number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. As a special case, a dollar sign ($) matches the last line of the last input file.

### 2.4.2 Context Addresses

A context address is a pattern ("regular expression") enclosed in slashes (/). The regular expressions recognized by **sed** are constructed as follows:

[1] An ordinary character (not one of those discussed below) is a regular expression and matches that character.

[2] A caret (^) at the beginning of a regular expression matches the null character at the beginning of a line.

[3] A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.

[4] The characters \n match an imbedded newline character, but not the newline at the end of the pattern space.

[5] A period (.) matches any character except the terminal newline of the pattern space.

[6] A regular expression followed by an asterisk (*) matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[7] A string of characters in square brackets ( [ ] ) matches any character in the string, and no others. If, however, the first character of the string is a caret (as in ^l), the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.

[8] A concatenation of regular expressions is a regular expression that matches the concatenation of strings matched by the components of the regular expression.

[9] A regular expression between the sequences \( and \) is identical in effect to the unadorned regular expression, but has side-effects described under the s command below and specification [10] immediately below.

[10] The expression *d* means the same string of characters matched by an expression enclosed in \( and \) earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of \( counting from the left. For example, the expression

```
^\(.*\)\1
```

matches a line beginning with two repeated occurrences of the same string.

[11] The null regular expression standing alone (e.g., //) is equivalent to the last regular expression compiled.

To use one of the special characters

(^ $ . * [ ] \ /)

as a literal (to match an occurrence of itself in the input), you must precede the special character with a backslash (\).

For a context address to "match" the input, the whole pattern within the address must match some portion of the pattern space.

### 2.4.3 Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command, we list its maximum number of allowable addresses (an error occurs if this number is exceeded).

If a command has no addresses, it is applied to every line in the input. If a command has one address, it is applied to all lines that match that address. If a command has two addresses, it is applied to the first line that matches the first address, and to all subsequent lines until (and including) the first subsequent line that matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses must be separated by a comma.

Here are some examples:

[1] **/an/** matches lines 1, 3, 4 in our sample text

[2] **/an.*an/** matches line 1

[3] **/^an/** matches no lines

[4] **/./** matches all lines

[5] **/\./** matches line 5

[6] **/r*an/** matches lines 1,3, 4 (number = zero)

[7] **/\(an\).*\1/** matches line 1

## 2.5 Functions

All functions are named by a single character. For each function, the following summary provides the maximum number of allowable addresses (in parentheses), the single character function name, possible arguments (in angles), an expanded English translation of the single-character name, and a description of what each function does.

**Note:** The angles around the arguments are not part of the argument, and should not be typed in actual editing commands.

### 2.5.1 Whole-Line-Oriented Functions

*(2)***d** Delete from the file (but don't write to the output) all lines matched by its address(es). As a side effect, attempt no further commands on the deleted line. Once this function is executed, a newline is read from the input, and the list of editing commands is restarted from the beginning of the newline.

*(2)***n** Read the next line from the input, replacing the current line. Write the current line to the output if it should be, and continue the list of editing commands.

*(1)***a***text* Append *text* to the output after the line matched by its address. This command is inherently multi-line. It must appear at the end of a line, and *text* may contain any number of lines. To preserve the one-command-to-a-line convention, interior newlines in *text* must be hidden by a backslash (\) immediately preceding the newline. The *text* argument is terminated by the first unhidden newline. Once this function is successfully executed, *text* is written to the output regardless of what later commands do to the line that triggered it. The triggering line may be deleted entirely; *text* is still written to the output. The *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

*(1)***i***text* Insert *text* in the output before (not after, as in **a**) the matched line.

*(2)***c***text* Delete the lines selected by its address(es), and replace them with *text*. Like **a** and **e**, lines in **c** must be followed by a newline hidden by a backslash; and interior new lines in *text* must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of *text* is written to the output, not one copy per line deleted. As with **a** and **i**, *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter. After a line has been deleted by a **c** function, no further commands are attempted on the line. If text is appended after a line by **a** or **r** functions, and the line is subsequently changed, the text inserted by the **c** function is placed before the text of the **a** or **r** functions.

**Note:** Within the text put in the output by these functions, leading blanks and tabs disappear, as always in **sed** commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash doesn't appear in the output.

To illustrate the use of some of the editing commands just described, consider this:

```
n
a\
XXXX
d
```

If applied to our standard input, these lines produce:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

For that matter, either of the two following command lists produce the same results:

```
n        n
i\        c\
XXXX    XXXX
d
```

## 2.5.2 The Substitute Function

The s (substitute) function changes parts of lines selected by a context search within the line. In prototype, it looks like this:

*(2)spattern replacement flags*

The part of a line matched by *pattern* is replaced with the text of *replacement*. *Pattern* contains a pattern like those in addresses. The only difference between *pattern* and a context address is that the context address must be delimited by slash ("/") characters; *pattern* may be delimited by any character other than space or newline. By default, only the first string matched by *pattern* is replaced. See the **g** flag below.

The *replacement* argument begins immediately after the second delimiting character of *pattern*, and must be followed immediately by another instance of the delimiting character. (Thus, there are exactly three instances of the delimiting character.)

The *replacement* is not a pattern, and the characters that are special in patterns do not have special meaning in *replacement*. Instead, other characters are special:

**&** Replaced by the string matched by *pattern*.

\*d* (where *d is a single digit)* Replaced by the *d*th substring matched by parts of *pattern* enclosed in \\( and \\). If nested substrings occur in *pattern*, the *d*th is determined by counting opening delimiters (\\( ). As in patterns, special characters may be made literal by preceding them with backslash (\\).

The *flags* argument may contain the following flags:

| | |
|---|---|
| **g** | Substitute *replacement* for all (non–overlapping) instances of *pattern* in the line. After a successful substitution, begin the scan for the next instance of *pattern* just after the end of the inserted characters; don't rescan characters put into the line from *replacement*. |
| **p** | Print the line if a successful replacement was done. In other words, cause the line to be written to the output if and only if the s function actually made a substitution. (Notice that if several s functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line is written to the output, one for each successful substitution.) |
| **w** *filename* | Write the line to *filename* if a successful replacement was done. That is, cause lines actually substituted by the **s** function to be written to a file named by *filename*. If *filename* exists before **sed** is run, overwrite it; other- |

wise, create it. Note that a single space must separate **w** and *filename*. The possibilities of multiple, somewhat different copies of one input line being written are the same as for **p**. A maximum of 10 different files may be mentioned after **w** flags and functions.

To illustrate how substitutions are done, consider the following command line:

**s/to/by/w changes** <RETURN>

When applied to our standard input, this line produces, on the standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file *changes*:

```
Through caverns measureless by man
Down by a sunless sea.
```

If the **nocopy** option is in effect, the command:

**s/[.,;?:]/*P&*/gp** <RETURN>

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the **g** flag, the command:

**/X/s/an/AN/p** <RETURN>

produces (in **nocopy** mode):

```
In XANadu did Kubla Khan
```

and the command:

**/X/s/an/AN/gp** <RETURN>

produces:

```
In XANadu did Kubla KhAN
```

## 2.5.3 Input/Output Functions

*(2)***p** Write the addressed lines to the standard output file. (The lines are printed at the time the **p** function is encountered, regardless of what succeeding editing commands may do to the lines.)

*(2)***w** *filename* Write the addressed lines to *filename*. If *filename* previously existed, overwrite it; if non-existent, create it. (The lines are written exactly as they exist when the **w** function is encountered for each line, regardless of what subsequent editing commands may do to them.) Exactly one space must separate **w** from *filename*. A maximum of ten different files may be mentioned in write functions and **w** flags after **s** functions, combined.

*(1)*r *filename* Read the contents of *filename* and append it after the line matched by the address (regardless of what subsequent editing commands do to line that matched its address). If e and a functions are executed on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed. Exactly one space must separate r from *filename*. If a file mentioned by r cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

**Note:** Only a limited number of files can be opened simultaneously. Make sure that you mention no more than ten files in w functions or flags. This is reduced by one if any r functions are present. (Only one read file is open at one time.)

To illustrate the functions mentioned above, assume that the file *note1* contains:

```
Note: Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
```

Then the following command:

```
/Kubla/r note1  <RETURN>
```

produces:

```
In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in
China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

### 2.5.4 Multiple Input-Line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines. They mainly provide pattern matches across lines in input:

*(2)*N Append next input line to the current line in the pattern space; separate the two input lines by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

*(2)*D Delete first part of the pattern space, up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

*(2)*P Print first part of the pattern space, up to and including the first newline. (The P, I, and D functions are equivalent to their lowercase counterparts if there are no imbedded newlines in the pattern space.)

## 2.5.5 Hold and Get Functions

Several functions save and retrieve part of the input for possible later use:

| | |
|---|---|
| *(2)***h** | Copy the contents of the pattern space into a hold area (destroying the previous contents of the hold area). |
| *(2)***H** | Hold and append to pattern space. Specifically, append the contents of the pattern space to the contents of the hold area. The former and new contents are separated by a newline. |
| *(2)***g** | Only get contents of hold area. Copy the contents of the hold area into the pattern space (destroying the previous contents of the pattern space). |
| *(2)***G** | Get (and append) contents of hold area. In particular, append the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline. |
| *(2)***x** | Exchange the contents of the pattern space and the hold area. |

To help illustrate the use of **sed** functions for saving and retrieving input, consider:

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

When applied to our standard example, these lines produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

## 2.5.6 Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part:

| | |
|---|---|
| *(2)***!** | Cause the next command (written on the same line) to be applied to all and only those input lines not selected by the address part. |
| *(2)*{*group*} | Cause the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of *group*. The first of the commands under control of the grouping may appear on the same line as the left brace ({) or on the next line. The *group* of commands is terminated by a matching right brace (}) on a line by itself. Groups can be nested. |
| *(0)*:*label* | Mark a place in the list of editing commands that may be referred to by **b** and **t** functions. The *label* may be any sequence of eight or fewer characters; if two different colon functions have identical labels, generate a compile time diagnostic, and attempt no execution. |
| *(2)***b***label* | Branch to *label*. Cause the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon (:) function with the same *label* is encountered. If no colon func- |

tion with the same label is found after all the editing commands are compiled, produce a compile time diagnostic, and attempt no execution. Take a **b** function with no *label* to be a branch to the end of the list of editing commands; do whatever should be done with the current input line, and read another input line; restart the list of editing commands from the beginning on the new line.

*(2)***t***label* Test to see whether any successful substitutions have been made on the current input line; if so, branch to *label*; if not, do nothing. The flag that indicates the execution of a successful substitution is reset by reading a new input line or executing a **t** function.

### 2.5.7 Miscellaneous Functions

*(1)*= Write, to the standard output, the number of the line matched by its address.

*(1)***q** Cause the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Chapter 3

# Lint: A C Program Checker

## 3.1 Introduction

**Lint**(1) examines C source code, detecting any bugs or obscurities. It enforces the type rules of C more strictly than the C compilers do. It may also be used to enforce many portability restrictions involved in moving programs between different machines and/or operating systems. Furthermore, it detects certain constructions which, although technically "legal," are nonetheless wasteful, error-prone, or otherwise best avoided. **Lint** accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between **lint** and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible, in part, because the compilers don't do sophisticated type checking, especially between separately-compiled programs. **Lint** takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This chapter discusses the use of **lint**, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

### 3.1.1 Usage

Suppose there are two C source files, *file1*.c and *file2.c*, that are ordinarily compiled and loaded together. Then the command

```
% lint file1.c file2.c <RETURN>
```

produces messages describing inconsistencies and inefficiencies in the programs. This

```
% lint -p file1.c file2.c <RETURN>
```

also produces these messages, as well as other messages that relate to the "portability" (to other operating systems and machines) of the programs. Replacing the **-p** by **-h** produces messages about constructions that, athough legal, demonstate poor programming style (according to **lint**). You may use both options

```
% lint -hp file1.c file2.c <RETURN>
```

to get both types of messages.

Many of the facts that **lint** needs to establish may, in reality, be impossible to discover. For example, it may not be possibe to know whether a given function in a program ever gets called without also knowing the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the **lint** algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, **lint** assumes it can be called.

**Lint** tries to give only relevant information. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, **lint** loses credibility, and its "error" messages merely clutter up the output, obscuring other, possibly more important messages.

## 3.1.2 Unused Variables and Functions

As sets of programs evolve, previously used variables and arguments to functions may become unused. It isn't uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally help you to discover bugs; if a function does a necessary job and is never called, something is probably wrong.

**Lint** complains about variables and functions that are defined but not otherwise mentioned. An exception is variables that are declared through explicit **extern** statements but are never referenced; thus, the statement

```
extern float sin();
```

evokes no comment if *sin* is never used. Note that this agrees with the semantics of the DOMAIN C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **-x** option to the **lint** invocation.

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 suppresses the printing of complaints about unused arguments. When **-v** is in effect, **lint** produces no messages about unused arguments except for those arguments that are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

In one particular case, information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some, but not all, files in a collection that is normally loaded together. Here, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. Use the **-u** option to suppress the spurious messages that might otherwise appear.

## 3.1.3 Set/Used Information

**Lint** attempts to detect cases where a variable is used before it is set. This isn't easy to do. Many algorithms take a good deal of time and space, and still produce "error" messages about perfectly valid programs. **Lint** 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 later, 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. This genre of complaint has its roots in stylistic, rather than actual, error. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables used in the expression that first sets them.

The set/used information also permits recognition of those local variables that are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

## 3.1.4 Flow of Control

**Lint** attempts to detect unreachable portions of the programs which it processes. It complains about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. **Lint** also complains about loops that can't be entered at the top. As is often true when **lint** makes false accusations, this condition may not be a bug, but an complaint about programming style.

**Lint** has an important area of blindness in the flow of control algorithm: it can't detect functions that are called and never return. Thus, a call to *exit* may cause unreachable code that **lint** doesn't detect; the most serious effects of this are in the determination of returned function values (see the next section).

A **break** statement that can't be reached causes no message. Programs generated by **yacc**(1) and **lex**(1) may have hundreds of unreachable **break** statements. The **-O** option in the C compiler often eliminates the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing you can do about them, and the resulting messages would clutter up **lint**'s output. If you want to see these messages, invoke **lint** with the **-b** option.

## 3.1.5 Function Values

Sometimes functions return values that are never used; sometimes programs incorrectly use function "values" that have never been returned. **Lint** 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; **lint** gives 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. For example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* calls *g* and then returns with no defined return value; this triggers a complaint from **lint**. If *g*, like *exit*, never returns, the message is produced even though nothing is actually wrong. In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial fraction of the "noise" messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value, but this value is sometimes or always unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., no testing for error conditions).

The dual problem of using a function value when the function does not return one is also detected. This is a serious problem that has been observed in "working" programs where, by chance, the desired function value was computed in the function return register.

### 3.1.6 Type Checking

**Lint** enforces the type checking rules of C more strictly than the compilers do. The additional checking goes on in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

Several operators 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 also 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 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, **lint** checks to see that enumeration variables or members are not mixed with other types or other enumerations. Another check ensures that the only operations applied are =, initialization, ==, !=, and function arguments and return.

### 3.1.7 Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider this assignment, where *p* is a character pointer:

```
    p = 1 ;
```

**Lint** has reason to complain. Now, consider the assignment

```
    p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. This assignment clearly signals the desired action. It seems harsh for **lint** to continue to complain about this. On the other hand, if this code is to be truly portable, such constructs should be examined carefully. The **–c** option controls the printing of comments about casts. When **–c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

### 3.1.8 Nonportable Character Use

On most C implementations, characters take on only positive values. **Lint** flags certain comparisons and assignments as illegal or nonportable. For example, the fragment

```
    char c;
        ...
    if( (c = getchar()) < 0 ) ....
```

works where the version of C allows a character to have a negative value, but fails on machines where characters always assume positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, **lint** responds with "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because, on some machines, bitfields are considered signed quantities. While it may seem unintuitive to consider that a two–bit field declared as type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

### 3.1.9 Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy in some implementations. This may happen in programs that 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**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **–a** option.

### 3.1.10 Unorthodox Constructions

**Lint** flags several perfectly legal, but somewhat unorthodox, constructions in the hope of promoting better code quality and clearer style, and even of pointing out bugs. The **–h** option enables these checks. For example, in the statement

```
    *p++ ;
```

the asterisk (*) does nothing. This provokes the message "null effect" from **lint**. In the following program fragment,

```
    unsigned x ;
    if( x < 0 ) ...
```

the test never succeeds. Similarly, the test

```
        if( x > 0 ) ...
```

is equivalent to

```
        if( x != 0 )
```

which may not be the intended action. **Lint** accuses you of making a "degenerate unsigned comparison" in these cases. If the code says

```
        if( 1 != 0 ) ....
```

**lint** reports "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs arising from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
        if( x&077 == 0 ) ...
```

or

```
        x<<2 + 40
```

probably don't do what was intended. The best solution is to place such expressions in parentheses, and **lint** encourages this by an appropriate message.

Finally, when the **–h** option is in force, **lint** complains about variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many to be bad style, often unnecessary, and frequently a bug.

### 3.1.11 Antiquated Syntax

**Lint** attempts to discourage several forms of older syntax. 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 ;
```

This expression could be interpreted as either

```
        a =- 1 ;
```

or

```
        a = -1 ;
```

It is especially perplexing when such ambiguity arises as the result of a macro substitution. The newer and preferred operators (+=, -=, etc. ) don't cause such confusion. To spur the abandonment of the older forms, **lint** complains about these older operators.

A similar issue arises with initialization. Older versions of C allowed

```
        int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example,

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read some distance past *x* to be sure what the declaration really is. 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.

### 3.1.12 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. On machines where double precision values may begin on any integer boundary, it is reasonable to assign integer pointers to double pointers. On other machines, double precision values must begin on even word boundaries; thus, not all such assignments make sense. **Lint** 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 in effect.

### 3.1.13 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 in which the stack runs backwards, function arguments are 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.

So that the efficiency of C on a particular machine isn't unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they 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.

**Lint** checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

draws the complaint:

```
warning: i evaluation order undefined
```

## 3.2 Implementation Details

**Lint** consists of two programs and a driver. The first program is a version of the Portable C Compiler. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator (as the other compilers do), **lint** produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring together all information collected about a given external name. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of **lint**.

## 3.2.1 Portability

This section describes some of the differences between C implementations, and discusses the **lint** features that encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The loader resolves these declarations and cause only a single word of storage to be set aside for *a*. Under some implementations, this isn't feasible, so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts that prevent the proper operation of the program. If **lint** is invoked with the **-p** option, it detects such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. Names known externally to UNIX software have seven significant characters, with the upper/lowercase distinction preserved. On other systems, the number of characters used and the preservation of case distinction may not be handled the same way. This leads to situations where programs that run fine under the UNIX system encounter loader problems on other systems. **Lint -p** 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. The UNIX system uses eight-bit ASCII. Other systems may use other character lengths or even other encoding schemes (e.g., EBCDIC). Moreover, character strings go from high to low bit positions ("left to right") on some systems, and low to high ("right to left") on the others. Thus, code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, are suspect. **Lint** is of little help here, except to flag multi-character character constants.

Other problems are likely to arise in shifting or masking words. C supports a bit-field facility that 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. For example, consider the use of

```
        x &= 0177700 ;
```

to clear the low order six bits of *x*. If the bit field feature cannot be used, the same effect can be obtained by writing the following, which works on many machines:

```
        x &=  9~ 8 077 ;
```

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 that has infiltrated itself into the C language. If there were a good way to discover the programs that would be affected, C could be changed; in any case, **lint** is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact 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.

### 3.2.2 Suppressing Unwanted Output

Sometimes you want **lint** to refrain from citing various constructs that, while technically "wrong", are nevertheless there for a good reason. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of controlling **lint**'s output is often desirable.

The form that this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by **lint** when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, **lint** directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the **lint** directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to **lint**, it can be asserted by the directive

```
        /* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if you want to turn off strict type checking for the next expression, you can use the directive

```
        /* NOSTRICT */
```

This causes the program to revert to the previous default after the next expression. The **-v** option can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by using this directive

```
/* VARARGS */
```

before the function definition. Sometimes, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments to be checked; thus, this causes the first two arguments to be checked, the others unchecked:

```
/* VARARGS2 */
```

Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file (see next section).

### 3.2.3 Library Declaration Files

**Lint** accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

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. You can use the VARARGS and ARGSUSED directives to specify features of the library functions.

**Lint** library files are processed almost exactly like ordinary source files. The only difference is that functions defined on a library file, but not used on a source file, draw no complaints. **Lint** doesn't simulate a full library search algorithm, and complains 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 program is run. When the **-p** option is in effect, another file containing descriptions of the standard I/O library routines that are expected to be portable across various machines is checked. The **-n** option can be used to suppress all library checking.

## 3.3 Summary of Lint Options

The command currently has the form

% **lint** [*options*] *files... library-descriptors...* <RETURN>

The following options are available:

| | |
|---|---|
| **–h** | Perform heuristic checks. |
| **–p** | Perform portability checks. |
| **–v** | Don't report unused arguments. |
| **–u** | Don't report unused or undefined externals. |
| **–b** | Report unreachable **break** statements. |
| **–x** | Report unused external declarations. |
| **–a** | Report assignments of **long** to **int** or shorter. |
| **–c** | Complain about questionable casts. |
| **–n** | Don't do any library checking. |
| **–s** | Perform heuristic checks (same as **h**). |

Chapter 4

# Make: A Program for Maintaining Programs

## 4.1 Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator such as **yacc**(1) or **lex**(1). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules.

Unfortunately, it is very easy to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, you may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can render obsolete a dozen other files. Forgetting to compile a routine that you've changed or one that uses changed declarations result in a program that does not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

**Make**(1) is a program that mechanizes many of the activities of program development and maintenance. If the information on inter-file dependencies and command sequences is stored in a file, the simple command

```
% make <RETURN>
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

```
think — edit — make — test . . .
```

**Make** is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. This chapter is a guide for users of **make**.

## 4.2 Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which it depends exist and are current, then creating the target if it has not been modified since its dependents were. **Make** does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example. A program named *prog* is made by compiling and loading three C language files *x.c*, *y.c*, *and z.c* with the lS library. By convention, the output of the C compilations are found in files named *x.o*, *y.o*, *and z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
        cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information is stored in a file named **makefile**, the command

```
% make <RETURN>
```

performs the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

**Make** operates by using three sources of information: a user-supplied description file (as above), filenames and "last-modified" times from the file system, and built-in rules to bridge some of the gaps.

In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*.

From the file system, **make** discovers that there are three "*.c*" files corresponding to the needed "*.o*" files. It then uses built-in information on how to generate an object from a source file (i.e., issue a **cc**(1) command with the **-c** option).

The following description file is equivalent to the one above, but takes no advantage of **make**'s innate knowledge:

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lS -o prog

x.o : x.c defs
       cc -c x.c

y.o : y.c defs
       cc -c y.c

z.o : z.c
       cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
% make <RETURN>
```

simply announces this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) is recompiled, and then *prog* is created from the new ".o" files. If only the file *y.c* had changed, only it is recompiled, but *prog* must still be reloaded.

If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
% make x.o <RETURN>
```

recompiles *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise, the current time is used. It is often useful to include rules with mnemonic names and commands that don't actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases, one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

**Make** has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

**Note:** To get a dollar sign, escape it with another dollar sign. The sequence $$ is escaped to $.

All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command:

- $*
- $@
- $?
- $<

They are discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
```

The command

```
% make <RETURN>
```

loads the three object files with the *lS* library. The command

```
% make "LIBES= -ll -lS" <RETURN>
```

loads them with both the lex (**-ll**) and the standard (**-lS**) libraries, since macro definitions on the command line override definitions in the description. (The shell requires that you quote arguments that include embedded blanks.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

## 4.3 Description Files and Substitutions

A description file contains three types of information:

- macro definitions
- dependency information
- executable commands

A comment convention is also supplied: all characters after a pound sign (#) are ignored, as is the pound sign itself. Blank lines and lines beginning with this character are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the **make** command line.

Other lines give information about target files. The general form of an entry is:

```
target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[(tab) commands] [# . . .]
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a pound sign (except in quotes) or newline. Commands may appear either

- after a semicolon on a dependency line, or
- on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the usual single–colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out–of–date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked.

In the double–colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built–in rule may also be executed. This detailed form is of particular value in updating archive–type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). **Make** normally stops if any command signals an error by returning a non–zero error code. (Errors are ignored if the **-i** option is specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set:

- $@ is set to the name of the file to be "made"
- $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent filenames.

If a file must be made but there are no explicit commands or relevant built–in rules, the commands associated with the name ".DEFAULT" are used. If no such name exists, **make** prints a message and stops.

## 4.4 Usage

The **make** command takes four kinds of arguments: macro definitions, flags, description filenames, and target filenames. The prototypical **make** command line is:

% **make** [ *flags* ] [ *macro definitions* ] [ *targets* ] <RETURN>

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments are made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are as follows:

| | |
|---|---|
| **-i** | Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file. |
| **-s** | Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file. |
| **-r** | Do not use the built-in rules. |
| **-n** | No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed. |
| **-t** | Touch the target files (causing them to be up-to-date) rather than issue the usual commands. |
| **-q** | Question. The **make** command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date. |
| **-p** | Print out the complete set of macro definitions and target descriptions. |
| **-d** | Debug mode. Print out detailed information on files and times examined. |
| **-f** | Description filename. The next argument is assumed to be the name of a description file. A filename of "-" denotes the standard input. If no -f arguments appear, the file named *makefile* or *Makefile* in the current directory is read. |

**Note:** The contents of the description files override any built-in rules present.

Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

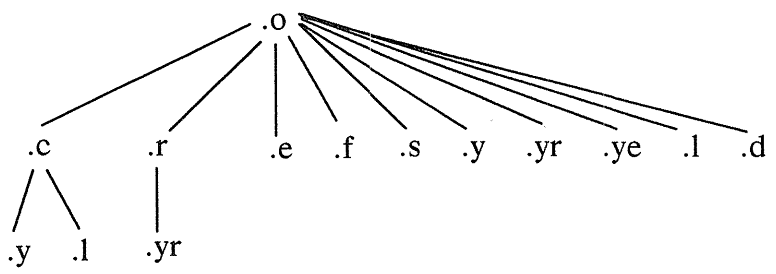
## 4.4.1 Implicit Rules

**Make** uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

| | |
|---|---|
| **.o** | Object file |
| **.c** | C source file |
| **.e** | Efl source file |
| **.r** | Ratfor source file |
| **.f** | Fortran source file |
| **.s** | Assembler source file |
| **.y** | Yacc-C source grammar |
| **.yr** | Yacc-Ratfor source grammar |

| | |
|---|---|
| **.ye** | Yacc–Efl source grammar |
| **.l** | Lex source grammar |

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* is needed and there is an *x.c* in the description or directory, it is compiled. If there is also an *x.l*, that grammar is run through **lex** before compiling the result. However, if there is no *x.c* but there is an *x.l*, then **make** discards the intermediate C–language file and uses the direct link shown in the figure above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

% **make CC=newcc** <RETURN>

causes the *newcc* command to be used instead of the usual C compiler. The macros *CFLAGS*, *RFLAGS*, *EFLAGS*, *YFLAGS*, and *LFLAGS* may be set to cause these commands to be issued with optional flags. Thus,

% **make "CFLAGS=-O"** <RETURN>

causes the optimizing C compiler to be used.

## 4.4.2 An Example

To illustrate the use of **make**, here's the description file used to maintain the **make** command itself. The code for **make** is spread over many C source files and a **yacc** grammar. The description file contains:

```
# Description file for the Make command

P = und -3 | opr -r2 # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c
```

```
cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print: $(FILES) # print recently changed files
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

**Make** usually prints each command before issuing it. Typing **make** with no arguments in a directory containing only the source and description file outputs the following:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars are mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the P macro:

```
make print "P = opr -sp"
```

or

```
make print "P= cat >zap"
```

## 4.5 Suggestions and Warnings

The most common difficulties arise from **make**'s specific understanding of what constitutes a dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what **make** would do, the **-n** option is very useful. The command

```
% make -n <RETURN>
```

orders **make** to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the **-t** (touch) option can save a lot of time, instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
% make -ts <RETURN>
```

("touch silently") causes the relevant files to appear up-to-date. Be careful, though, because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The **-d** (debugging) option causes **make** to print a very detailed description of its activities, including file times. The output is verbose, and recommended only as a last resort.

## 4.6 Summary of Suffixes and Rules

The **make** program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **-r** option is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; **make** looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, **make** acts as described earlier. The transformation rule names are the concatenation of the two suffixes.

The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in your description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, you can just add an entry for ".SUFFIXES" in his own description file; the dependents are added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed.)

The following is an excerpt from the default rules file:

```
.SUFFIXES: .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o:
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o:
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o:
        $(AS) -o $@ $<
.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c:
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

Chapter 5

# System V Extensions to the Make Program

## 5.1 General

This chapter describes an augmented version of **make**(1) specifically supported for System V use. Since this version is upwardly compatible with the old version, we provide only a discussion of the extensions to **make**, along with some reasons for the chosen implementation, and examples to demonstrate the additional features.

The **make** command is an excellent program administration tool. However, previous versions of **make** hindered large-scale use because of the following shortcomings:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) filename format was difficult or impossible.
- Environment variables were completely ignored.
- Ability to maintain files in a remote directory was inadequate.

The augmented version of **make** eliminates these problems. The additional features are within the original syntactic framework of **make** and few, if any, new syntactical entities are introduced. A notable exception is the *include* file capability. Most additions result in a “Don’t know how to make ...” message from the old version of **make**.

This chapter illustrates the additional features of the **make** program. In general, the examples are taken from existing *makefiles*.

## 5.2 Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A new macro, **MAKEFLAGS**, is maintained by **make** and defined as the collection of all input flag arguments into a string (without minus signs). The new macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the *makefile* update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, consider the **MAKEFLAGS** macro (environment variable).

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it isn't present or null, set the internal **make** variable **MAKEFLAGS** to the null string. Otherwise, assume each letter in **MAKEFLAGS** to be an input flag argument and process it as such. (The only exceptions are the **–f**, **–p**, and **–r** flags.)
2. Read and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.
3. Read macro definitions from the command line. Make these non–resettable, so that any further assignments to these names are ignored.
4. Read the internal list of macro definitions found in the file *rules.c* of the source for **make**. The complete makefile that represents the internally defined macros and rules of the current version of **make** is as follows:

```
# LIST OF SUFFIXES

        .SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~

# PRESET VARIABLES

        MAKE=make
        YACC=yacc
        YFLAGS=
        LEX=lex
        LFLAGS=
        LD=ld
        LDFLAGS=
        CC=cc
        CFLAGS=-o
        AS=as
        ASFLAGS=
        GET=get
        GFLAGS=

# SINGLE SUFFIX RULES

        .c:
                $(CC) -n -o $< -o $@

        ~c:
                $(GET) $(GFLAGS) -p $< > $*.c
                $(CC) -n -o $* .c -o $*
                -rm -f $*.c

        .sh:
                cp $< @

        .sh~:
                $(GET) &(GFLAGS) -p $< > .sh
                cp $* .sh $*
                -rm -f $* .sh.
```

```
# DOUBLE SUFFIX RULES
        .c.o:
                $(CC) $(CFLAGS) -c $<

        .c~.o:
                $(GET) $(CFLAGS) -p $< > $*.c
                $(CC) $(CFLAGS) -c $*.c
                -rm -f $*.c

        .c~.c:
                $(GET) $(GFLAGS) -p $< >$*.c

        .s.o:
                $(AS) $(ASFLAGS) -o $@ $<

        .s~.o:
                $(GET) $(GFLAGS) -p $< > $*.s
                $(AS) $(ASFLAGS) -o $* .o $* .s
                -rm -f $*.s

        .y.o:
                $(YACC) $(YFLAGS) $<
                $(CC) $(CFLAGS) -c y.tab.c
                rm y.tab.o$@

        .y~.o:
                $(GET) $(GFLAG) -p $< > $*.y
                $(YACC) $(YFLAGS) $*.y
                $(CC) $(CFLAG) -c y.tab.c
                rm -f y.tab $*.y
                mv y.tab.o $*.o

        .l.o:
                $(LEX) $(LFLAGS) $<
                $(CC) $(CFLAGS) -c lex.yy.c
                rm lex.yy.c
                mv  lex.yy.o $@

        .l~.o:
                $(GET) $(GFLAGS) -p $< > $*.l
                $(LEX) $(GFLAG) $*.l
                $(CC) $(CFLAGS) -c lex.yy.c
                rm -f lex.yy.c $*.l
                mv lex.yy.o $*.o

        .y.c:
                $(YACC) $(YFLAGS) $<
                mv y.tab.c $@

        .y~.c:
                $(GET) $( GFLAGS) -p $< > $*.y
                $(YACC) $(YFLAGS) $*.y
                mv -f $*.c
                -rm -f $*.y

        .l.c:
                $(LEX) $<
                mv lex.yy.c$@

        .c.a:
                $(CC) -c $(FLAGS) $<
                ar rv $@ $*.o
                rm -f $*.o

        .c~.a:
                $(GET) $(GFLAGS) -p $< > $*.c
                $(CC) -c $(CFLAGS) $*.c
                ar rv $@ $*.o
```

```
.s~.a:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        ar rv $@ $*.o
        -rm -f $*.[so]
.h~.h:
        $(GET) $(GFLAGS) -p $< > $*.h
```

Thus, if you type **make –r ...** and a *makefile* includes the *makefile* in the above example, the results are identical to excluding the **–r** option and the *include* line in the *makefile*. To reproduce the internal definitions output shown above, type

```
# make -fp - < /dev/null 2>/dev/null <RETURN>
```

The output appears on the standard output. Default definitions for the C compiler (CC=cc) are also included.

5. Read the environment, treating environment variables as macro definitions and marking them as *exported* (in the shell sense). **MAKEFLAGS*** isn't an internally defined variable (in *rules.c*), so the same assignment is done twice (except when **MAKEFLAGS** is assigned on the command line). It was read previously to turn the debug flag on before anything else was done. **:MAKEFLAGS** is read and set again.
6. Read the *makefile(s)*. The assignments in the *makefile(s)* override the environment. Thus, when a *makefile* is read and executed, you can anticipate the results. That is, you get what is seen unless the **–e** flag is used. The **–e** is an additional command line flag that tells **make** to have the environment override the *makefile* assignments. Thus, typing **make –e ...** causes the variables in the environment to override the definitions in the *makefile* (there is no way to override the command line assignments.) Also, **MAKEFLAGS** overrides the environment if assigned. This is useful for further invocations of **make** from the current *makefile*.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions (from *rules.c*)
2. environment
3. *makefile(s)*
4. command line.

The **–e** flag changes the order to:

1. internal definitions (from *rules.c*)
2. *makefile(s)*
3. environment
4. command line.

This order is general enough to let you define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

## 5.3 Recursive Makefiles

Another feature was added to **make** concerning the environment and recursive invocations. If the sequence "$(MAKE)" appears anywhere in a shell command line, the line is executed even if the **-n** flag is set. Since the **-n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing that actually gets executed is the **make** command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done gets printed out, including output from lower level invocations of **make**.

## 5.4 Format of Embedded Shell Commands

The **make** program remembers embedded newlines and tabs in shell command sequences. Thus, if you put a **for** loop in the makefile with indentation, indentation and backslashes are retained when **make** prints it out. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is added.

## 5.5 Archive Libraries

The **make** program has an improved interface to archive libraries. Previously, you named a member of a library in one of the following ways:

```
lib(object.o)
lib((_localtime))
```

(The second form actually refers to an entry point of an object file within the library. **Make** looks through the library, locates the entry point, and translates it to the correct object filename.) To use this procedure to maintain an archive library, you need this type of *makefile*:

```
lib::   lib(ctime.o)
$(CC) -c -O ctime.c
ar rv lib ctime.o
rm ctime.o
lib::   lib(fopen.o)
$(CC) -c -O fopen.c
ar rv lib fopen.o
rm fopen.o
...and so on for each object ...
```

This is tedious and error-prone, however. Command sequences for adding a C file to a library are the same for each invocation, except for a different filename each time.

The current version lets you access a rule for building libraries. The handle for the rule is the ".a" suffix. Thus, a ".c.a" rule is the rule for compiling a C language source file, adding it to the library, and removing the ".o" cadaver. Similarly, the ".y.a", the ".s.a", and the ".l.a" rules rebuild **yacc**(1), assembler, and **lex**(1) files, respectively. The current archive rules defined internally are ".c.a", ".c{.a", and ".s{.a". You can define in *makefile* other rules needed.

The above two-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual ".c.a" rules are as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

Thus, the $@ macro is the ".a" target (lib); the $< and $* macros are set to the out-of-date C language file; and the filename scans the suffix, respectively (*ctime.c* and *ctime*). The $< macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees

```
lib:    lib(ctime.o)
@echo lib up-to-date
```

Assuming the object in the library is out-of-date with respect to *ctime.c*, and there is no *ctime.o* file, here is the route that **make** takes:

1. Do *lib*.

2. To do *lib*, do each dependent of *lib*.

3. Do *lib*(*ctime.o*).

4. To do *lib*(*ctime.o*), do each dependent of *lib*(*ctime.o*). (There are none.)

5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.o*) has a parenthesis in the name to identify the target suffix as ".a". This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard wired into **make**.

6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, $@ (=*lib*) and $* (=*ctime*).

7. Look for a rule ".X.a" and a file **$*.X**. The first ".X" (in the .SUFFIXES list) that fulfills these conditions is ".c" so the rule is ".c.a", and the file is *ctime.c*. Set $< to be *ctime.c* and execute the rule. In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, and the search ends.

8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

   ```
   echo lib up-to-date
   ```

Note that, to let *ctime.o* have dependencies, the following syntax is required:

```
lib(ctime.o):        $(INCDIR)/stdio.h
```

Thus, explicit references to *.o* files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. The $% macro is evaluated each time $@ is evaluated. If there is no current archive member, $% is null. If an archive member exists, then $% evaluates to the expression between the parenthesis.

Here is a sample *makefile* for a larger library. Note that there are no lingering "*.o" files left around in this example; the result is a library maintained directly from the source files (or, e.g., SCCS files).

```
#  @(#)/usr/src/cmd/make/make.tm 3.2

LIB =lsxlib
PR=lp
INSDIR = /rl/flopO/
INS = eval
lsx:            $(LIB) low.o mch.o
                ld -x low.o mch.o $(LIB)
                mv a.out lsx
                @size lsx

#  Here, $(INS) as either "." or "eval".
lsx:
                $(INS)'cp lsx $(INSDIR)lsx . .
                strip $(INSDIR)lsx . .
                ls -l $(INSDIR)lsx'
print:
                $(PR) header.slow.smch.s*.h*.c Makefile
$(LIB):
                $(LIB)(CLOCK.o)
                $(LIB)(main.o)
                $(LIB)(tty.o)
                $(LIB)(trap.o)
                $(LIB)(sysent.o)
                $(LIB)(sys2.o)
                $(LIB)(syn3.o)
                $(LIB)(syn4.o)
                $(LIB)(sys1.o)
                $(LIB)(sig.o)
                $(LIB)(fio.o)
                $(LIB)(kl.o)
                $(LIB)(alloc.o)
                $(LIB)(nami.o)
                $(LIB)(iget.o)
                $(LIB)(rdwri.o)
                $(LIB)(subr.o)
                $(LIB)(bio.o)
                $(LIB)(decfd.o)
                $(LIB)(sip.o)
                $(LIB)(space.o)
                $(LIB)(puts.o)
                @echo $(LIB) now up to date.
.s.o:
                as -o $*.o header.s $*.s
.o.a:
                ar rv $@ $<
                rm -f $<
.s.a:
                as -o $*.o header.s $*.s
                ar rv $@ $*.o
                rm -f $*.o
```

## 5.6 Source Code Control System Filenames: The Tilde

The syntax of **make** doesn't directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, "s." precedes the filename part of the complete pathname.

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, ".c~.o" refers to the rule that transforms an SCCS C language source file into an object. Specifically, the internal rule is

```
cc~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, you can define other rules and suffixes that may prove useful. The tilde gives you a handle on the SCCS filename format so that this is possible.

## 5.7 The Null Suffix

In the UNIX system source code, many commands consist of a single source file. It was wasteful to maintain an object of such files for **make**. The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program *cat*, a rule in the *makefile* of the following form is needed:

```
.c:
        $(CC) -n -O $< -o $@
```

In fact, this ".c:" rule is internally defined so that no *makefile* is necessary. You need only type

```
# make cat dd echo date  <RETURN>
```

(these are notable single file programs) and all four C language source files are passed through the above shell command line associated with the ".c:" rule. Though you may add others in the *makefile*, the internally defined single suffix rules are

```
.c:
.c~:
.sh:
.sh~:
```

## 5.8 Include Files

The **make** program has an include file capability. If the string *include* appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the string is assumed to be a filename read by the current invocation of **make**. File descriptors are stacked for reading include files so that no more than about 16 levels of nested *includes* are supported.

## 5.9 Invisible SCCS Makefiles

SCCS makefiles are invisible to **make**. Thus, if you type **make**, and only a file named *s.makefile* exists, **make** does a **get**(1) on the file, and then reads and removes the file. Using the **-f**, **make** gets, reads, and removes arguments and include files.

## 5.10 Dynamic Dependency Parameters

A new dependency parameter has been defined. It has meaning only on the dependency line in a makefile. The $$@ refers to the current "thing" to the left of the colon (i.e., $@). Also, the form $$(@F) allows access to the file part of $@. Thus, in

```
cat:    $$@.c
```

the dependency is translated at execution time to the string "cat.c". This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a *makefile* such as:

```
CMDS = cat dd echo date cc cmp comm ar ld chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

This is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file with a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@** is evaluated at execution time. It helps in maintaining */usr/include* from a makefile in */usr/src/head*. Thus, */usr/src/head/makefile* would look like this:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCIDR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F
        cp $? $@
        chmod 0444 $@
```

This should completely maintain the */usr/include* directory whenever one of the above files in */usr/src/head* is updated.

## 5.11 Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The "D" refers to the directory part of the single letter macro. The "F" refers to the filename part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a shell command can be

```
cd $(<D); $(MAKE) $(<F)
```

The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *ucb*. The FRC is a convention for forcing **make** to completely rebuild a target starting from the beginning.

## 5.12 Output Translations

Macros in shell commands can now be translated when evaluated as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in **$(macro)** is that the evaluated **$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus, the occurrence of *string1* in **$(macro)** means that a regular expression of the following form has been found:

```
 *<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. A more general regular expression match could be implemented as needed. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (i.e., those files ending in ".c"). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffices) that define the archive library. These translations are added in an effort to make more general use of the wealth of information generated by **make**.

Chapter 6

# Lex: A Lexical Analyzer Generator

## 6.1 Introduction

**Lex**(1) is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general purpose language that recognizes regular expressions. You must specify the regular expressions in the source specifications given to **lex**. The code written by **lex** recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections that you provide are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

You must also supply the additional code beyond expression matching needed to complete the tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for your program fragments. As a result, a high-level expression language is provided to write the string expressions to be matched while your freedom to write actions is unimpaired. Thus, for example, if you want to use a string manipulation language for input analysis, you are not forced to write processing programs in the same and often inappropriate string handling language.

**Lex** is not a complete language, but rather a generator representing a new language feature that can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, **lex** can write code in different host languages. The host language is used for the output code generated by **lex**, as well as the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes **lex** adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, your background, and the properties of local implementations. Currently, C is the only supported host language. Although **lex** is a UNIX program, code generated by **lex** may be taken anywhere the appropriate compilers exist.

**Lex** turns input expressions and actions (known collectively as *source*), into the host general-purpose language. The generated program, *yylex*, recognizes expressions in a stream (input) and performs the specified actions for each expression as it is detected. The diagram below summarizes these features.

$$\begin{tikzcd}
\text{Source} \arrow[r] & \boxed{\text{lex}} \arrow[r] & \text{yylex} \\
\text{Input} \arrow[r] & \boxed{\text{yylex}} \arrow[r] & \text{Output}
\end{tikzcd}$$

To illustrate, consider a program for deleting from the input all blanks or tabs at the ends of lines. This is all that is needed:

```
[ \t]+$[\t];
```

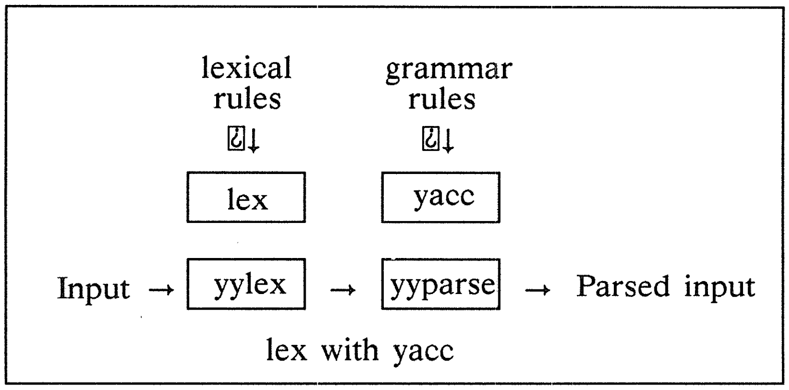
The program contains a *%%* delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by **lex** (*yylex*) ignores these characters. Everything else is copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$[ \t];
[ \t]+  [ \t]printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

You can use **lex** alone for simple transformations, or for analysis and statistics gathering on a lexical level. **Lex** can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface **lex** and **yacc**(1). **Lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate.

When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in the following diagram. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**.



**Yacc** users realize that the name *yylex* is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

**Lex** generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number of **lex** rules or the complexity of the rules is not important in determining speed, unless rules that include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex**.

In the program written by **lex**, the fragments supplied by you (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. You are provided with an opportunity to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

**Lex** is not limited to source that can be interpreted on the basis of one-character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, **lex** recognizes *ab* and leaves the input pointer just before *cd*. This type of backing up is more costly than the processing of simpler languages.

## 6.2 Lex Source

The general format of **lex** source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second *%%* is optional, but the first is required to mark the beginning of the rules.

The absolute minimum **lex** program is

```
%%
```

(no definitions, no rules), which translates into a program that copies the input to the output unchanged.

In the outline of **lex** programs just shown, the *rules* represent a user's control decisions. They are a table in which the left column contains regular expressions and the right column contains actions – program fragments to be executed when the expressions are recognized. Thus, an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. Here, the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose you want to change several words from British to American spelling. **Lex** rules such as

```
colour       printf("color");
mechanise    printf("mechanize");
petrol       printf("gas");
```

would be a start. These rules are not quite enough, however, since the word *petroleum* would become *gaseum*. A way of dealing with this situation is described later.

## 6.3 Lex Regular Expressions

The definitions of regular expressions are very similar to those in QED. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D.*

### 6.3.1 Operators

The following operator characters are available:

- "
- \
- [
- ]
- ^

- -
- ?
- .
- *
- +
- |
- (
- )
- $
- /
- {
- }
- %
- <
- >

If any operator is to be used as a text character, you must escape it with quotes. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus,

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless (but unnecessary) to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. If you quote every non–alphanumeric character being used as a text character, you can avoid having to remember the list of current operator characters, or having to worry that further extensions to **lex** might lengthen the list.

You may also turn an operator character into a text character by preceding it with a backslash (\), as in

```
xyz\+\+
```

This is another, less readable, equivalent of the above expressions. The quoting mechanism may also be used to get a blank into an expression. Normally, blanks or tabs end a rule. Any blank character not contained within square brackets (i.e., [ ] ) must be quoted. Several normal C escapes with a backslash are recognized. For instance, \n is newline, \t is tab, and \b is backspace. To enter a literal backslash, use two backslash characters (\\). Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline, and those included in the list above is always a text character.

## 6.3.2 Character Classes

You can specify classes of characters by using the operator pair represented by square brackets ([ ]). The construction [*abc*] matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special; these are a backslash (\), a dash (–), and a caret (^). A dash indicates ranges. For example, this indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline:

```
[a–z0–9<>_]
```

Ranges may be given in either order. Using a dash between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and gets a warning message. (For example, [0-z] in S–2ASCIIS0 is many more characters than it is in S–2EBCDICS0.) If you must include the dash character in a character class, it should be first or last; thus, this matches all the digits and the two signs:

```
[–+0–9]
```

In character classes, a caret (^) must appear as the first character after the left bracket, to indicate that the resulting string is to be complemented with respect to the computer character set. Thus,

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters, and

```
[^a-zA-Z]
```

matches any character that is not a letter. The backslash character (\) provides the usual escapes within character class brackets.

## 6.3.3 Arbitrary Character Match

A period or dot ( . ) matches all characters except newline. Escaping into octal is possible, although non–portable. The following matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde):

```
[\40-\176]
```

## 6.3.4 Optional Expressions

A question mark (?) indicates an optional element of an expression. Thus,

```
ab?c
```

matches either *ac* or *abc*.

## 6.3.5 Repeated Expressions

Repetitions of classes are indicated by an asterisk (*) and a plus sign (+). Thus,

```
a*
```

is any number of consecutive *a* characters, including zero, while

```
a+
```

matches one or more instances of *a*. Furthermore, this

```
[a-z]+
```

represents all strings of lowercase letters, and

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

## 6.3.6 Alternation and Grouping

The pipe operator ( | ) indicates alternation. For example, this

```
(ab|cd)
```

matches either *ab* or *cd*. Note that we use parentheses for grouping, although they are not necessary on the outside level. In fact, this

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions, e.g.,

```
(ab|cd+)?(ef)*
```

which matches such strings as *abefef, efefef, cdef,* or *cddd* (but not *abc*, *abcd*, or *abcdef*).

## 6.3.7 Context Sensitivity

**Lex** recognizes a small amount of surrounding context. The two simplest operators for this are the caret (^) and the dollar sign ($). If the first character of an expression is a caret, the expression is only matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of the caret, complementation of character classes, since that only applies within the square bracket operators. If the last character is a dollar sign, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the slash (/), which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus,

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in **lex** by *start* conditions, which are explained later. If a rule is only to be executed when the **lex** automaton interpreter is in start condition x, use angle brackets and prefix the rule by

```
<x>
```

If we consider "being at the beginning of a line" to be start condition *ONE*, then a caret is equivalent to

```
<ONE>
```

### 6.3.8 Repetitions and Definitions

Braces ({ }) specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example,

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the **lex** input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

An initial percent (%) character is special; it is the separator for **lex** source segments.

## 6.4 Lex Actions

When an expression written according to the rules presented in the last section is matched, **lex** executes the corresponding action. This section describes some **lex** features that aid in the writing of actions.

**Note:** By default, **lex** performs a copy of the input to the output on all strings not otherwise matched.

**Lex** users who want to absorb the entire input without producing any output must provide rules to match everything. This is the normal situation when **lex** is being used with **yacc**. If you assume that an action is something **lex** does instead of the default action (copying the input to the output), it follows that a rule that merely copies can be safely omitted. As a corollary to this, a character combination that is omitted from the rules and appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

A simple "solution" is to ignore the input. Specifying a C null statement (;), as an action causes this result. A frequent rule is

```
[ \t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is to use the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written as

```
" "             |
"\t"            |
"\n"            ;
```

with the same result, although in different style. The quotes around \n and \t aren't required.

In more complex actions, you probably want to know the actual text that matched some expression like *[a–z]+*. **Lex** leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule such as

```
[a-z]+      printf("%s", yytext);
```

prints the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and s indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO. Thus,

```
[a-z]+      ECHO;
```

is the same as the above. Since the default action is just to print the characters found, why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches *read* it normally matches the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is further explained below.

It is sometimes more convenient to know the end of what has been found; hence **lex** also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, you might write

```
[a-zA-Z]+       {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string overwrites the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Additional characters previously matched are returned to the input. This provides the same sort of lookahead offered by the slash (/) operator, but in a different form.

Consider a language that defines a string as a set of characters between a set of double quotation marks. To include quotation marks in a string, you must escape them with a preceding backslash (\). The regular expression that matches such a case is somewhat confusing, so that it might be preferable to write

```
\"[^"]*\" {
        if (yytext[yyleng-1] == '\\')
        yymore();
        else
        ... normal user processing
        }
```

This, when faced with a string such as *"abc\"*def", first matches the five characters *"abc*; then the call to *yymore()* causes the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of *"=–a"*. Suppose you want this treated as *"=– a"* but a message also printed. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=–". Alternatively, you may want this treated as "= –a". To do this, just return the minus sign as well as the letter to the input.

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

This performs the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup is required in the rule action. It isn't necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=–3", however, makes

```
=-/[^ \t\n]
```

an even better rule.

In addition to these routines, **lex** also permits access to the I/O routines it uses. They are as follows:

| | |
|---|---|
| **input**() | Returns the next input character |
| **output**(*c*) | Writes the character *c* on the output |
| **unput**(*c*) | Pushes the character *c* back onto input stream to be read later by **input**() |

By default, these routines are provided as macro definitions, but you can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from unusual places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the **lex** lookahead does not work. **Lex** does not look ahead at all if it does not have to, but every rule ending in a plus sign, asterisk, question mark, or dollar sign, or containing a slash implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. The character set used by **lex** is discussed below.

**Note:** The standard **lex** library imposes a 100–character limit on backup.

Another **lex** library routine that you may want to redefine is *yywrap()* which is called whenever **lex** reaches an end–of–file. If *yywrap* returns a 1, **lex** continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, provide a *yywrap* that arranges for new input and returns 0. This instructs **lex** to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, and the like at the end of a program. Note that it is not possible to write a normal rule that recognizes end–of–file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled, since a value of 0 returned by input is taken to be end–of–file.

## 6.5 Ambiguous Source Rules

**Lex** can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses as follows:

- The longest match is preferred.
- Among rules that matched the same number of characters, the rule given first is preferred.

Assume that the rules

```
integer keyword action ...;
[a-z]+  identifier action ...;
```

have been given in that order. If the input is *integers*, it is taken as an identifier, because *[a-z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) does not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* *dangerous*. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. In fact, it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression matches

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, stops after *'first'*. The consequences of errors like this are mitigated by the fact that the . (dot) operator does not match a newline. Thus, expressions like

```
.*
```

stop on the current line. Don't try to defeat this with expressions like

```
[.\n]+
```

or equivalents; the generated program tries to read the entire input file, causing internal buffer overflows.

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some **lex** rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that dot does not include newline. Since *she* includes *he*, **lex** normally does not recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

You may override this choice. The action *REJECT* means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose you really want to count the included instances of *he*:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression is then be counted. In this example, of course, you could note that *she* includes *he* but not vice versa, and omit the *REJECT* action on *he*; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, *REJECT* is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is, the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* is to be incremented, the appropriate source is

```
%%
[a-z][a-z]      {digram[yytext[0]][yytext[1]]++; REJECT;}
7    .. 9 9\n      ;
```

where the *REJECT* is necessary to pick up a letter pair beginning at every character, rather than at every other character.

**Note:** *REJECT* doesn't rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and *REJECT* is executed, you must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate input that is not yet processed.

## 6.6 Lex Source Definitions

As we have stated, the format of **lex** source is:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. There is a need for additional options to define variables for use in user programs and for use by **lex**. These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source not intercepted by **lex** is copied into the generated program. Such source falls into three classes:

- Lines that aren't part of a **lex** rule or action beginning with a blank or tab are copied into the **lex** generated program. Such source input prior to the first %% delimiter is external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **lex** which contains the actions. This material must look like program fragments, and should precede the first **lex** rule. As a side effect of the above, lines beginning with a blank or tab and containing a comment are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow the host language convention.
- Anything included between lines containing only %{ and %} is copied out as above, and the delimiters are discarded. This permits entering text like preprocessor statements that must begin in column 1, or copying lines that don't look like programs.
- Anything after the third %% delimiter, regardless of format, is copied out after the **lex** output.

Definitions intended for **lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define **lex** substitution strings. The format of such lines is

```
name translation
```

where *translation* becomes associated with *name*. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Thus, using {D} for the digits and {E} for an exponent field might abbreviate rules to recognize numbers:

```
D       [0-9]
E       [DEde][-+]?{D}+
%%
{D}+    printf("integer");
{D}+"."{D}*({E})?           |
{D}*"."{D}+({E})?           |
{D}+{E}                           printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point.

To correctly handle the problem posed by a FORTRAN expression such as *35.EQ.I* (which doesn't contain a real number), a context–sensitive rule such as

```
[0-9]+/"."EQ          printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. We discuss these possibilities further below in the "Summary of Source Format."

## 6.7 Usage

There are two steps to compiling a **lex** source program. First, the **lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of **lex** subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag **–ll**. So an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use **lex** with **yacc**, see the next section. Although the default **lex** I/O routines use the C standard library, the **lex** automata themselves do not do so; if private versions of *input*, *output*, and *unput* are given, the library can be avoided.

## 6.8 Lex and Yacc

Before you use **lex** with **yacc**, note that **lex** writes a program named *yylex()*, the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** calls *yylex()*. In this case, each **lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing this line in the last section of **yacc** input:

```
# include "lex.yy.c"
```

If the grammar is named *good* and the lexical rules are named *better*, you can use this command sequence:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (**–ly**) should be loaded before the **lex** library to obtain a main program that invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

## 6.9 More Examples

Consider copying an input file while adding 3 to every positive number divisible by 7. Here is a **lex** source program to do just that:

```
%%
        int k;
[0-9]+      {
        k = atoi(yytext);
        if (k%7 == 0)
        printf("%d", k+3);
        else
        printf("%d",k);
        }
```

The rule [0-9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. A percent (%) operator (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. However, you may not want the program to alter such input items as *49.63* or *X7*, or to increment the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one:

```
%%
        int k;
-?[0-9]+        {
        k = atoi(yytext);
        printf("%d", k%7 == 0 ? k+3 : k);
        }
-?[0-9.]+       ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a dot ( . ) or preceded by a letter are picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form

```
a?b:c
```

means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program that histograms the lengths of words, where a word is defined as a string of letters:

```
        int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
if (lengs[i] > 0)
printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input, it prints the table. The final statement *return(1)*; indicates that **lex** is to perform wrapup. If *yywrap* returns zero (false), it implies that further input is available and the program is to continue reading and processing. If you provide a *yywrap* that never returns true, it generates an infinite loop.

To further illustrate, here are some parts of a program for converting double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish uppercase and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W        [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
 {d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
printf(yytext[0]=='d'? "real" : "REAL");
}
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]        ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of the caret (^). Next, we find a few rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+ |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ {
/* convert constants */
for(p=yytext; *p != 0; p++)
{
if (*p == 'd' || *p == 'D')
*p=+ 'e'- 'd';
ECHO;
}
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'-'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. Following that, a series of names must be respelled to remove their initial *d*. By using the array *yytext*, the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}    |
{d}{c}{o}{s}    |
{d}{s}{q}{r}{t} |
{d}{a}{t}{a}{n} |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}    |
{d}{l}{o}{g}10  |
{d}{m}{i}{n}1   |
{d}{m}{a}{x}1   {
        yytext[0] =+ 'a' - 'd';
        ECHO;
        }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}        {yytext[0] =+ 'r' - 'd';
                ECHO;
                }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*        |
[0-9]+      |
\n      |
.       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

## 6.10 Left Context Sensitivity

It is sometimes desirable to have several sets of lexical rules applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^), for example, is a prior context operator, recognizing immediately preceding left context just as a dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended to produce a facility similar to that for adjacent right context. However, it is unlikely to be as useful, since the relevant left context frequently appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another; the use of *start* conditions on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all.

It may be more convenient, however, to have **lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only recognized when **lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Suppose you want to copy the input to the output, changing the word *magic* to *first* on every line that began with the letter *a*, changing *magic* to *second* on every line that began with the letter *b*, and changing *magic* to *third* on every line that began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag. The following program should be adequate:

```
        int flag;
%%
^a       {flag = 'a'; ECHO;}
^b       {flag = 'b'; ECHO;}
^c       {flag = 'c'; ECHO;}
\n       {flag = 0 ; ECHO;}
magic    {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule that is only recognized when **lex** is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state, this resets the initial condition of the **lex** automaton interpreter:

```
BEGIN 0;
```

A rule may be active in several start conditions. Thus, this is a legal prefix:

```
<name1,name2,name3>
```

Rules not beginning with an angle bracket (< >) prefix are always active. The previous example can be written another way:

```
%START AA BB CC
%%
^a        {ECHO; BEGIN AA;}
^b        {ECHO; BEGIN BB;}
^c        {ECHO; BEGIN CC;}
\n        {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

Here, the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than your own code.

## 6.11 Character Set

The programs generated by **lex** handle character I/O only through the routines *input*, *output*, and *unput*. Thus, **lex** accepts the character representation provided in these routines and uses it to return values in *yytext*. For internal use, a character is represented as a small integer that, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter a is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines that translate the characters, **lex** must be given a *translation table*. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

indicating the value associated with each character. Thus, this example character table

```
%T
1       Aa
2       Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

maps the lowercase and uppercase letters together into the integers 1 through 26, newline into 27, plus (+) and minus (–) into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

## 6.12 Summary of Source Format

The general form of a **lex** source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of the following:

- Definitions, in the form "name space translation".
- Included code, in the form "space code".
- Included code, in the form

  ```
  %{
  code
  %}
  ```

- Start conditions, given in the form

  ```
  %S name1 name2 ...
  ```

- Character set tables, in the form

  ```
  %T
  number space character-string
  ...
  %T
  ```

- Changes to internal array sizes, in the form

  ```
  %x  nnn
  ```

  where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

  | | |
  |---|---|
  | p | positions |
  | n | states |
  | e | tree nodes |
  | a | transitions |
  | k | packed character classes |
  | o | output array size |

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

| | |
|---|---|
| x | the character "x" |
| "x" | an "x", even if x is an operator |
| \x | an "x", even if x is an operator |
| [xy] | the character x or y |

| | |
|---|---|
| [x-z] | the characters x, y or z |
| [^x] | any character but x |
| . | any character but newline |
| ^x | an x at the beginning of a line |
| <y>x | an x when **lex** is in start condition y |
| x$ | an x at the end of a line |
| x? | an optional x |
| x* | 0,1,2, ... instances of x |
| x+ | 1,2,3, ... instances of x |
| x\|y | an x or a y |
| (x) | an x |
| x/y | an x but only if followed by y |
| {xx} | the translation of xx from the definitions section |
| x{m,n} | *m* through *n* occurrences of x |

Chapter 9

# A C Language Reference

## 9.1 Introduction

This chapter describes the C language on the DEC PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it 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.

## 9.2 Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") are ignored except as they serve to separate tokens. Some white space is needed 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 that could constitute a token.

### 9.2.1 Comments

A slash followed by an asterisk (i.e., /*) introduces a comment. The reverse of this (i.e., */) terminates a comment. Comments do not nest.

### 9.2.2 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, used by various assemblers and loaders, are more restricted:

| | |
|---|---|
| DEC PDP–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 |

### 9.2.3 Keywords

These identifiers are reserved for use as keywords, and may not be used otherwise:

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

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**.

### 9.2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics that affect sizes are summarized in Section 9.2.6.

#### 9.2.4.1 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. The digits **8** and **9** have octal value 10 and 11 respectively. 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 that exceeds the largest unsigned machine integer is likewise taken to be **long**.

#### 9.2.4.2 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.

#### 9.2.4.3 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 non–graphic characters, the single quote ( ' ) and the backslash ( \ ), may be represented according to the following table of escape sequences:

| | | |
|---|---|---|
| newline | NL(LF) | **\n** |
| horizontal tab | HT | **\t** |
| 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 that are taken to specify the value of the desired character. A special case of this construction is \0 (not followed by a digit), to indicate the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

#### 9.2.4.4 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.

## 9.2.5 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 Section 9.4) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte at the end of each string so that programs that scan the string can find its end. In a string, the double quote character ( " ) must be preceded by a backslash ( \ ). Also, the same escapes as described for character constants may be used. A backslash and the immediately following newline are ignored.

## 9.2.6 Hardware Characteristics

The following table summarizes certain hardware properties that vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought a priori.

| | DEC PDP–11 | Honeywell 6000 | IBM 370 | Interdata 8/32 |
|---|---|---|---|---|
| | ASCII | ASCII | EBCDIC | ASCII |
| char | 8 bits | 9 bits | 8 bits | 8 bits |
| int | 16 | 36 | 32 | 32 |
| short | 16 | 36 | 16 | 16 |
| long | 32 | 36 | 32 | 32 |
| float | 32 | 36 | 32 | 32 |
| double | 64 | 72 | 64 | 64 |
| range | $\pm 10^{\pm 38}$ | $\pm 10^{\pm 38}$ | $\pm 10^{\pm 76}$ | $\pm 10^{\pm 76}$ |

## 9.3 Syntax Notation

In the syntax notation used in this chapter, 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 non-terminal symbol is indicated by the subscript "opt," so that

{ *expression* $_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in Section 9.18 of this chapter.

## 9.4 What's In a Name?

C 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, and register. Automatic variables are local to each invocation of a block (see Section 9.9.2), 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.

C supports several fundamental types of objects:

- Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and 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.
- Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo *$2 sup n$* 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 are referred to as *arithmetic* types. Types **char** and **int** of all sizes are collectively called *integral* types. Types **float** and **double** are collectively called *floating* types.

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* that 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.

## 9.5 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 that 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.

## 9.6 Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. Section 9.6.6 summarizes the conversions demanded by most ordinary operators; it is supplemented by the discussion of each operator.

### 9.6.1 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 in this chapter, only the PDP–11 sign–extends. On the PDP–11, character variables range in value from –128 to 127; the characters of the ASCII alphabet are all positive. 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.

### 9.6.2 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.

### 9.6.3 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 doesn't 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.

### 9.6.4 Pointers and Integers

An integer or long integer 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.

### 9.6.5 Unsigned Integers

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 $2 sup roman wordsize$). 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.

### 9.6.6 Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern is 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.

## 9.7 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 Section 9.7.4) are those expressions defined in Sections 9.7.1–9.7.3. Within each section, the operators have the same precedence. Left– or right–associativity is specified in each section for the operators discussed therein.

The precedence and associativity of all the expression operators is summarized in the grammar of Section 9.18. 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. All 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.

## 9.7.1 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-lvalue . 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 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**; 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 9.8.6.)

An expression in parentheses expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses doesn't 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, a primary expression has type "pointer to ...", a subscript expression is **int**, and 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 section together with the discussions in Sections 9.7.1, 9.7.2, and 9.7.4 on identifiers, *, and + respectively; 9.14.3 summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma–separated list of expressions that 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**; and as usual, 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 Sections 9.7.2 and 9.8.7.

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.

On the other hand, 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 an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a – and a >) 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 9.8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See Section 9.14.1.

## 9.7.2 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 $2 sup n$, 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 0, 0 if the value of its operand is non-zero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one'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 of addition (9.7.4) and assignment operators (9.7.14) 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. The result is noted, and then 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. The result is noted, and then 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 in 9.8.7.

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 integer 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 type name in parentheses. 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**.

## 9.7.3 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 chapter, 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 **float**.

## 9.7.4 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, one that 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 as 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.

### 9.7.5 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 (and is, on the PDP–11) arithmetic (fill by a copy of the sign bit).

### 9.7.6 Relational Operators

The relational operators group left–to–right, but 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.

### 9.7.7 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, but the result is machine dependent unless 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.

### 9.7.8 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.

### 9.7.9 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.

### 9.7.10 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.

### 9.7.11 Logical AND Operator

> *logical-and-expression:*
> > *expression && expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 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**.

### 9.7.12 Logical OR Operator

> *logical-or-expression:*
> > *expression || expression*

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

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**.

## 9.7.13 Conditional Operator

*conditional-expression:*
  *expression ? expression : expression*

Conditional expressions group right–to–left. The first expression is evaluated and if it is non–zero, 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; otherwise, if both are pointers 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.

## 9.7.14 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 equal sign (=), 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.

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 7.4; all right operands and all non–pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers that cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

### 9.7.15 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, for example in a list of actual arguments to functions (see Section 9.7.1) and lists of initializers (see Section 9.8.6), the comma operator as described in this section 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.

## 9.8 Declarations

Declarations specify the interpretation that C gives to each identifier; they don't necessarily reserve storage associated with the identifier. Declarations have the form

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

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-specifierst* $_{\text{opt}}$
  *sc-specifier decl-specifiers* $_{\text{opt}}$

The list must be self-consistent in a way described in Section 9.8.1.

### 9.8.1 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; it is discussed in 9.8.8. The meanings of the various storage classes were discussed in 9.4.

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 (10) 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**, **char**, 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.

## 9.8.2 Type Specifiers

The type-specifiers are

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

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

short int
long int
unsigned int
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 and unions are discussed in 9.8.5; declarations with **typedef** names are discussed in 9.8.8.

## 9.8.3 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 9.8.6. 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.

## 9.8.4 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 like this, where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator:

```
T D1
```

Suppose this declaration makes the identifier have type "... **T**," where the "..." is empty if **D1** is a plain identifier (type of **x** in "**int x**" is just **int**). 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 either of these two forms

```
D[constant-expression]

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 9.15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that 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 multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. Functions may not return arrays, structures, unions or functions, although they may return pointers to such things. Further, 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 that returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then 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. For 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 three-dimensional array of integers, with rank 3x5x7. In complete detail, **x3d** 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**.

## 9.8.5 Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object that 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 series of declarations for 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*

Usually, 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 that increase as their declarations are read left-to-right. Each non-field 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 that doesn't 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, 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 doesn't restrict the types of things that are declared as fields, but implementations aren't required to support any but integer fields. Moreover, even **int** fields may be considered unsigned. On the PDP-11, fields aren't signed and have only integer values. 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; they 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 that 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 may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

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**.

## 9.8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by an equal sign (=), 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 (described in 9.15), or expressions that 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.

Uninitialized static and external variables are guaranteed to start off as 0; uninitialized automatic and register variables 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) then 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 that 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 3 elements from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**. Also, this initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0:

```
float y[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

Finally, this

```
char msg[] = "Syntax error on line %s0;
```

shows a character array whose members are initialized with a string.

## 9.8.7 Type Names

In two contexts (specifying type conversions explicitly by means of a cast, and as an argument of **sizeof**), you should supply the name of a data type. Use a "type name" (a declaration for an object of that type that omits the name of the object) to do this.

*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 must be non-empty. This restriction makes it 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 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" respectively.

## 9.8.8 Typedef

Declarations whose "storage class" is **typedef** don't define storage. Instead, they define identifiers that can be used later as if they were type keywords naming fundamental or derived types:

*typedef-name:*
*identifier*

If a declaration involves **typedef**, each identifier appearing therein becomes syntactically equal to the type keyword (naming the type associated with the identifier) in the way described in 9.8.4.
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** declaration does not introduce brand new types, only synonyms for types that 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.

## 9.9 Statements

Except as indicated, statements are executed in sequence.

### 9.9.1 Expression Statement

Most statements are expression statements, having the form

> *expression ;*

Usually expression statements are assignments or function calls.

### 9.9.2 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, so that the initializations aren't performed. Initializations of **static** variables are done only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

### 9.9.3 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 non-zero, 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**.

### 9.9.4 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 non-zero. The test takes place before each execution of the statement.

### 9.9.5 Do Statement

The **do** statement has the form

> **do** *statement* **while** ( *expression* ) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

### 9.9.6 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.

### 9.9.7 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 9.15.

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 **case** and **default** prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see **break**, 9.9.8.

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.

## 9.9.8 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.

## 9.9.9 Continue Statement

This statement causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement (i.e, to the end of the loop):

```
continue ;
```

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 Section 9.9.13.)

## 9.9.10 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.

## 9.9.11 Goto Statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (9.9.12) located in the current function.

### 9.9.12 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 sub-blocks in which the same identifier has been redeclared. See Section 9.11.

### 9.9.13 Null Statement

The null statement has the form

> ;

A null statement is useful to carry a label just before the right brace ( } ) of a compound statement or to supply a null body to a looping statement such as **while**.

## 9.10 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 (9.8.2) 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.

### 9.10.1 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 9.11.2 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 that 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.

C 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 ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

### 9.10.2 External Data Definitions

An external data definition has the form

> *data-definition:*
>     *declaration*

The storage class of such data may be **extern** (the default) or **static**, but not **auto** or **register**.

## 9.11 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, 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, characterized by the rule that references to the same external identifier are references to the same object.

### 9.11.1 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 that are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see 9.11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file 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 (9.8.5) 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 that don't conflict. Members and tags follow the same scope rules as other identifiers. All **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***.

### 9.11.2 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 that refer to the same external identifier refer to the same object, so be sure that that the type and size specified in the definition are compatible with those specified by each function that 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 multi–file program, an external data definition without the **extern** specifier must appear in exactly one of the files. Any other files that 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**.

## 9.12 Compiler Control Lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with a pound sign (#) communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and are effective, independent of scope, until the end of the source program file.

### 9.12.1 Token Replacement

A compiler-control line of the form

> **#define** *identifier token-string*

(note the lack of a trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

> **#define** *identifier( identifier* , ..., *identifier* ) *token-string*

where there is no space between the first identifier and the left parenthesis, is a macro definition with arguments. Subsequent instances of the first identifier followed by a left parenthesis, a sequence of tokens delimited by commas, and a right parenthesis 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 don't separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject 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 a backslash (\) 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.

### 9.12.2 File Inclusion

A compiler control line of the form

> **#include** *"filename"*

causes the replacement of that line by the entire contents of *filename*. The named file is sought first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

> **#include** <*filename*>

searches only the standard places, and not the directory of the source file. You may nest **#include**'s.

### 9.12.3 Conditional Compilation

A compiler control line of the form

> **#if** *constant-expression*

checks to see if the constant expression (see 9.15) evaluates to non-zero.

A control line of the form

```
#ifdef identifier
```

checks to see if the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a **#define** control line. A control line of the form

```
#ifndef identifier
```

checks to see if 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 an **#else** or, lacking an **#else**, the **#endif**, are ignored. These constructions may be nested.

### 9.12.4 Line Control

For the benefit of other preprocessors that generate C programs, a line of the form

```
#line constant identifier
```

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 filename does not change.

## 9.13 Implicit Declarations

You need not always 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, since **auto** functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning ...", it is implicitly declared to be **extern**.

In an expression, an identifier followed by a left parenthesis and not already declared is contextually declared to be "function returning **int**".

## 9.14 Types Revisited

This section summarizes operations that can be performed on objects of certain types.

### 9.14.1 Structures and Unions

There are only two things that can be done with a structure or union, that is, to name one of its members by means of the period or dot ( . ) operator, or to take its address by a unary ampersand (&). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, these operations (but not necessarily others) may well be allowed.

Section 9.7.1 says that in a direct or indirect structure reference (with . or ->) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before ., and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a -> is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure. Such constructions are non-portable.

### 9.14.2 Functions

Only two things 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. To pass one function to another, say, for example:

```
int f();
...
g(f);
```

Then the definition of g might read

```
g(funcp)
int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

**Note:** An f must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by the left parenthesis.

### 9.14.3 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 so that **E1[E2]** is identical to *((E1)+(E2)). Because of the conversion rules that 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 multi-dimensional arrays. If **E** is an *n*- dimensional array of rank $i times j times ... times k$, then **E** appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j times ... times k$. 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.

For example, consider

```
int x[3][5];
```

Here **x** is a 3x5 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 5 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.

Thus, arrays in C are stored row-wise (last subscript varies fastest), and the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

## 9.14.4 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, described in Sections 9.7.2 and 9.8.7.
A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is needed depends on the machine. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. We give details for some specific machines 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 if the subject pointer doesn't refer to an object suitably aligned in storage. A pointer to an object of a given size may always 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. The pointer 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 is measured in bytes. The **chars** have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the 18 bits, and the two bits that select the character in a word just to their right. Thus **char** pointers are measured in units of $2 sup 16$ bytes; everything else is measured in units of $2 sup 18$ machine words. All **double** quantities and aggregates containing them must lie on an even word address (0 mod $2 sup 19$).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be 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.

## 9.15 Constant Expressions

At times, C requires expressions that 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, 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 ampersand (&) operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary ampersand 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.

## 9.16 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. However, the compilers all behave 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, 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**, multi–character character constants may be permitted. However, 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 left–to–right on other machines. These differences are invisible to isolated programs that don't indulge in type punning (for example, 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 doesn't initialize structures containing bit–fields, and doesn't accept a few assignment operators in certain contexts where the value of the assignment is used.

## 9.17 Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, 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 actually decrements **x** since the equal sign and the dash are adjacent, but which might easily be intended to assign **-1** to **x.**

The syntax of initializers has changed. Previously, the equal 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+2)
```

resembles a function declaration closely enough to confuse the compilers.

## 9.18 Syntax Summary

This summary is intended as an aid to understanding C syntax, rather than an exact statement of the language.

### 9.18.1 Expressions

The basic expressions are:

```
expression:
        primary
        * expression
        & expression
        - 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 ]
        lvalue . 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.

## 9.18.2 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**
  *struct-or-union-specifier*
  *typedef-name*

*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*

## 9.18.3 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*
- ;

### 9.18.4 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:*
    *type-decl-list*
    *function-statement*

*function-statement:*
    { *declaration-list* $_{opt}$ *statement-list* }

*data-definition:*
    **extern** $_{opt}$ *type-specifier* $_{opt}$ *init-declarator-list* $_{opt}$ ;

    **static** $_{opt}$ *type-specifier* $_{opt}$ *init-declarator-list* $_{opt}$ ;

### 9.18.5 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 identifier*

## 9.19 Recent Changes to C

A few extensions have been made to the C language beyond what is described in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1978. This section describes some of the major extensions.

### 9.19.1 Structure Assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

### 9.19.2 Enumeration Type

A new data type analogous to the scalar types of Pascal has been provided. It is:

*enum-specifier*

with syntax

*enum-specifier:*
- **enum** { *enum-list* }
- **enum** *identifier* { *enum-list* }
- **enum** *identifier*

*enum-list:*
- *enumerator*
- *enum-list* , *enumerator*

*enumerator:*
- *identifier*
- *identifier* = *constant-expression*

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, puce };
        ...
enum color *cp, col;
```

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 identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with an equal sign (=) appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with an equal sign gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct; unlike structure tags and members, they are drawn from the same set as ordinary identifiers. Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation, all enumeration variables are treated as if they were **int**.

Chapter 10

# Ratfor: A Preprocessor for a Rational Fortran

## 10.1 Introduction

Many programmers feel that Fortran is the closest thing to a universal programming language currently available. It is often the most "efficient" language available, particularly for programs requiring much computation. In spite of this, Fortran has some important weaknesses.

Perhaps the worst deficiency is in the control flow statements (conditional branches and loops) that express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces you into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some further restrictions). And, of course, there can be no ELSE to a Fortran IF; you cannot specify an alternative action if the IF is not satisfied.

The Fortran DO limits you to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran) to go from 1 to N-1. Moreover, the DO is useless if your problem doesn't map into an arithmetic progression.

Thus, Fortran programs must be written with many labels and branches. The resulting code is very difficult to read and understand; hence, it is hard to debug and modify.

To overcome the deficiencies, and to translate Fortran into a language that uses a preprocessor, **ratfor**(1) is available. This chapter explains the basic concepts and implementations of this language.

## 10.2 Language Design

**Ratfor** attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language is Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, **ratfor**'s primary task is to conceal this part of Fortran from the user, by providing good control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, many of the "cosmetic" deficiencies of Fortran can be eliminated, providing a language that is easier and more pleasant to read and write.

Beyond these two aspects – control flow and cosmetics – **ratfor** does nothing more about Fortran's weaknesses. Though it would be straightforward to extend it to provide character strings, for example, they aren't needed by everyone, and the preprocessor would be harder to implement. The design of **ratfor** was based on the principle that it need not know any Fortran; any language feature that required understanding of Fortran has been omitted. Even within the confines of control flow and cosmetics, we'd tried to be selective in the features provided, focusing only the most useful constructs.

The remainder of of this section contains an informal description of **ratfor**. The control flow aspects will be quite familiar to readers familiar with languages like Algol, PL/I, and Pascal. The cosmetic changes are equally straightforward.

### 10.2.1 Statement Grouping

Fortran doesn't provide a way to group statements together, short of making them into a subroutine. A standard construction "if a condition is true, do this group of things"

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead, you must state the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or must be modified, you must translate this group of statements back into a clearer form before anyone can tell what it does.

**Ratfor** eliminates this error-prone and confusing back-and-forth translation. The first form shown above is the way the computation is written in **ratfor**. A group of statements can be treated as a unit by enclosing them in braces ( { } ). Throughout the language, wherever a single **ratfor** statement can be used, several may be enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and, of course, **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The greater-than character (>) is clearer than ".GT.", so **ratfor** translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (e.g., *"x>100"*), quotes are not allowed in ANSI Fortran, so **ratfor** converts them into the right number of *H*'s.

**Ratfor** is a free–form language. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. Our previous example could also be written in **ratfor** like this:

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

Here, 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 (**ratfor** or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated, because the statement is clearly not finished on the first line. In general, **ratfor** continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free–form language permits wide latitude in formatting styles, readability is important. In particular, proper indentation is necessary to make the logical structure of the program obvious to the reader.

### 10.2.2 The "else" Clause

**Ratfor** provides an **else** statement to handle the construction "if a condition is true, do this thing, otherwise do that thing." For instance, this writes out the smaller of *a* and *b*, then the larger, and sets *sw* appropriately:

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that isn't a translation. To understand the Fortran version, you must scan the entire program and see that no other statement branches to statements 10 or 20 before you know for sure that you're dealing with an **if–else** construction.

With the **ratfor** version, there's no question about how to get to the parts of the statement. The **if–else** is a single unit that can be read and understood, or ignored if not relevant. The program says what it means. If the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is as follows, where the **else** part is optional:

```
if (legal Fortran condition)
        ratfor statement
else
        ratfor statement
```

The legal Fortran condition is anything that can legally go into a Fortran Logical IF. **Ratfor** doesn't check this clause, since it doesn't know enough Fortran to tell what is permitted. The **ratfor** statement is any **ratfor** or Fortran statement, or any collection of them in braces.

## 10.2.3 Nested "if" Statements

The statement following an **if** or an **else** can be any **ratfor** statement, so it can also be another **if** or **else**. Consider this problem: the variable *f* is to be set to −1 if *x* is less than zero, to +1 if *x* is greater than 100, and to 0 otherwise. Then in **ratfor**, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here, the statement after the first **else** is another **if–else**. Logically, it's just a single statement (though rather complicated). This code says what it means. Any version written in straight Fortran is necessarily indirect, because Fortran doesn't let you be so specific. Following an **else** with an **if** is one way to write a multi-way branch in **ratfor**. This structure provides a way to specify one of several alternatives:

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
...
else
        - - -
```

**Ratfor** also provides a **switch** statement that does the same job in certain special cases; in more general situations, we have to make do with spare parts. The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until you find one that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply.

If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

### 10.2.4 Ambiguity in "if–else" Structures

Let's consider complicated structures involving nested **if**'s and **else**'s. For example,

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
                else
                write(6, 2) y
```

shows two **if**'s and only one **else**. With which **if** does the **else** belong?

This is a genuine ambiguity in **ratfor**, as it is in many other programming languages. The ambiguity is resolved in **ratfor** (as elsewhere) by saying that in such cases the **else** goes with the closest previous un–**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

We recommend that you resolve such cases by explicit braces, just to make your intent clear. Using the case above, then, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

This doesn't change the meaning, but it eliminates ambiguity. If we want the other association, we must write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
                write(6, 2) y
```

### 10.2.5 The "switch" Statement

The **switch** statement provides a clean way to express multi–way branches that branch on the value of some integer–valued expression. The syntax is

```
switch (expression) {
        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The expression inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches; then, the statements following that **case** are executed. If no cases match the expression, and a **default** section exists, the statements with it are done; if there is no **default**, nothing is done. In any case, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C should be aware that this behavior is not the same as the C **switch**.)

## 10.2.6 The "do" Statement

The **do** statement in **ratfor** is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
        do 10 i = 1, n
                x(i) = 0.0
                y(i) = 0.0
                z(i) = 0.0
10      continue
```

The syntax is:

```
do legal-Fortran-DO-text
        ratfor statement
```

The part that follows the keyword **do** must be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (not currently permitted in ANSI Fortran), they can be used in a **ratfor do**.

The **ratfor** statement part is often enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array *m* to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of *m* to -1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement following the **do** is logically a single statement, however complicated, and thus needs no braces.

## 10.2.7 The "break" and "next" Statements

**Ratfor** provides a statement for leaving a loop early, and one for beginning the next iteration. A **break** causes an immediate exit from the **do**; in effect it is a branch to the statement after the **do**. A **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

The **break** and **next** also work in the other **ratfor** looping constructions explained in the next few sections. A **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. Specifying **next 2** iterates the second enclosing loop. Multi-level **break**'s and **next**'s aren't used much, since they lead to code that is hard to understand and slightly risky to change.

## 10.2.8 The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this is typically done once with *I* set to 2, even though common sense suggests that perhaps it shouldn't be. Of course, a **ratfor do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k  {
                ---
        }
```

but this has to be a conscious act, and is often overlooked.

A more serious problem with the DO statement is that it encourages programs written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write them. Code that must be contorted to fit the requirements imposed by the Fortran DO is that much harder to write and understand.

To overcome these difficulties, **ratfor** provides a **while** statement that is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria:

```
real function sin(x, e)
        # returns sin(x) to accuracy e, by
        # sin(x) = x - x**3/3! + x**5/5! - ...
```

```
sin = x
term = x

i = 3
while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
}

return
end
```

**Note:** If the routine is entered with **term** already smaller than *e*, the loop is done zero times, that is, no attempt is made to compute *x**3* and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears – the code works at one of its boundaries. (The test *i<100* is the other boundary – making sure the routine stops after some maximum number of iterations.)

A pound sign (#) in a line marks the beginning of a comment; the rest of the line is the comment itself. Comments and code can co-exist on the same line. You can make marginal remarks, which is impossible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran) to emphasize the natural divisions of a program.

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 in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose *nextch* is a function that returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is simply

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, necessary here to mark the end of the **while**; if not present, the **while** controls the next statement. When the loop is broken, *ich* contains the first non-blank. Of course, the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) consider this line illegal.

### 10.2.9 The "for" Statement

The **for** statement is another **ratfor** loop that attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. Thus, a DO loop is

```
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, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they are done zero times if *n* is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be rewritten with a **for** as

```
        for (i=3; abs(term) > e & i < 100; i=i+2) {
                term = -term * x**2 / float(i*(i-1))
                sin = sin + term
        }
```

The syntax of the **for** statement is

```
        for ( init ; condition ; increment )
                ratfor statement
```

The *init* is any single Fortran statement that gets done once before the loop begins. The *increment* is any single Fortran statement that gets done at the end of each pass through the loop, before the test. The *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons must always be present. A non-existent condition is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things hard to express with a DO statement and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
        for (i = 80; i > 0; i = i - 1)
                if (card(i) != blank)
                        break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (The **break** and **next** work in **for**'s and **while**'s just as in **do**'s). If *i* reaches zero, the card is all blank.

This code is rather difficult to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
                DO 10 J = 1, 80
                        I = 81 - J
                        IF (CARD(I) .NE. BLANK) GO TO 11
        10      CONTINUE
                I = 0
        11      ...
```

The version that uses the **for** handles the termination condition properly for free; *i* is zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression. The following program walks along a list (stored in an integer array *ptr*) until it finds a zero pointer, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

### 10.2.10 The "repeat-until" Statement

In spite of warnings, there are times when you need a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        ratfor statement
until (legal Fortran condition)
```

The *ratfor statement* part 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 bare **repeat** is the cleanest way to specify an infinite loop. Of course, such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

The **repeat-until** statement is less popular than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

### 10.2.11 More on "break" and "next" Statements

A **break** exits immediately from **do**, **while**, **for**, and **repeat-until**. A **next** goes to the test part of **do**, **while**, and **repeat-until**, and to the increment step of a **for**.

### 10.2.12 The "return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable that can be assigned to; the last value stored in it is the function value upon return. Here is a routine called *equal* that returns 1 if two arrays are identical, and zero if different. Array ends are marked by the special value of -1.

```
# equal _ compare str1 to str2;
#   return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1) {
                equal = 1
                return
        }
equal = 0
return
end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, **ratfor** provides such a **return** statement. In a function *F*,

```
return(expression)
```

is equivalent to

```
{ F = expression; return }
```

For example, here is *equal* again:

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1)
                return(1)
        return(0)
end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. We present another version of *equal* shortly.

## 10.2.13 Cosmetics

The visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, **ratfor** provides a number of cosmetic facilities that make programs more readable.

## 10.2.14 Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. 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 others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and it is placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

### 10.2.15 Translation Services

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; the next character is taken literally. This provides a way to get quotes (and the backslash itself) into quoted strings:

```
"\\\'"
```

Here, a string containing a backslash and an apostrophe is represented. (It isn't the standard convention of doubled quotes, but it is easier to use and more general.)

Any line beginning with a percent (%) is unaltered except that the percent is stripped off and the the line is moved one position to the left. This helps in inserting control cards, and other items that should not be transmogrified (e.g., an existing Fortran program). Use a percent only for ordinary statements; do not use it for the condition parts of **if**, **while**, etc., or the output may appear in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a percent character:

| | | | |
|---|---|---|---|
| == | .eq. | != | .ne. |
| > | .gt. | >= | .ge. |
| < | .lt. | <= | .le. |
| & | .and. | \| | .or. |
| ! | .not. | ^ | .not. |

These translations are also provided for input devices with restricted character sets:

| | | | |
|---|---|---|---|
| [ | { | ] | } |
| $( | { | $) | } |

### 10.2.16 The "define" Statement

Any string of alphanumeric characters can be defined as a name. Thereafter, whenever that name occurs in the input (delimited by non-alphanumerics), 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.

A **define** is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50
dimension a(ROWS), b(ROWS, COLS)
        if (i > ROWS  |  j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

Here, the defining text is everything after the comma up to the balancing right parenthesis (allowing multi-line definitions). You should use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers.

Here's the routine *equal* again, this time with symbolic constants:

```
define      YES      1
define      NO       0
define      EOS      -1
define      ARB      100

# equal _ compare str1 to str2;
#       return YES if equal, NO if not
        integer function equal(str1, str2)
        integer str1(ARB), str2(ARB)
        integer i
        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == EOS)
                        return(YES)
        return(NO)
        end
```

### 10.2.17 The "include" Statement

The statement

```
include file
```

inserts the file found on input stream file into the **ratfor** input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical.

### 10.2.18 Limitations

**Ratfor** catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since **ratfor** knows no Fortran, any errors you make are reported by the Fortran compiler, so from time to time you must relate a Fortran diagnostic back to the **ratfor** source.

Keywords are reserved; using **if**, **else**, etc., as variable names typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran *nH* convention is not recognized anywhere by **ratfor**; use quotes instead.

## 10.3 Implementation

**Ratfor** was originally written in C on the UNIX operating system. The language is specified by a context free grammar and the compiler constructed using **yacc**(1).

The **ratfor** grammar is simple and straightforward, being essentially

```
prog   : stat
       | prog   stat
stat   : if (...) stat
       | if (...) stat else stat
       | while (...) stat
       | for (...; ...; ...) stat
       | do ... stat
       | repeat stat
       | repeat stat until (...)
       | switch (...) { case ...: prog ...
               default: prog }
       | return
       | break
       | next
       | digits   stat
       | { prog }
       | anything unrecognizable
```

The observation that **ratfor** knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first item on a source line is not a keyword (**if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels *L* and *L+1* are generated and the value of *L* is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (it may be some distance away and include nested **if**'s), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the statement part of the **else**. Code generation for the various loops is equally simple.

You might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted to

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" makes even a measurable difference. In the few cases where it is important, the offending lines can be protected by a percent (%).

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an ad hoc recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of **ratfor** is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a **ratfor** written in itself and originally bootstrapped with the C version. The **ratfor** version was written so as to translate into the portable subset of Fortran, so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form *c*vfc;* avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. **Ratfor** doesn't gratuitously generate non-standard Fortran.)

The **ratfor** version is about 1500 lines of **ratfor** (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the **ratfor** version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

## 10.4 Benefits and Drawbacks of Ratfor

"It's so much better than Fortran" is the most common response of users when asked how well **ratfor** meets their needs. Although cynics might consider this vacuous, it does appear that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in **ratfor** is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be read. The looping statements that test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in **ratfor**; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in **ratfor** tend to be as readable as programs written in more modern languages like Pascal. Once freed from the limits of Fortran's clerical detail and rigid input format, you can easily write code that is readable, even esthetically pleasing. For example, here is a **ratfor** implementation of the linear table search discussed by Knuth:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large part (5400 lines) of **ratfor**, including a subset of the **ratfor** preprocessor itself, can be found in *Software Tools*, by B. W. Kernighan and P. J. Plauger (Addison-Wesley, 1976).

The biggest single problem is that many Fortran syntax errors are not detected by **ratfor** but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending **ratfor** line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since **ratfor** generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in **ratfor**'s ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

A number of implementation weaknesses are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by **ratfor**, and this is perceived as a problem by users with a large body of existing Fortran programs. Protecting every line with a percent character (%) is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by *Struct*, a program (developed at AT&T Bell Laboratories) that converts arbitrary Fortran programs into **ratfor**.

Users who export programs often complain that the generated Fortran is "unreadable" because it isn't tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be improved (**ratfor** now has an option to copy **ratfor** comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success – since **ratfor** is relatively easy to modify, there are now several dialects of **ratfor**. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

## 10.5 Conclusions

**Ratfor** demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. You must avoid throwing in "features" – things that the user may trivially construct within the existing framework.

You must also avoid getting sidetracked on irrelevancies. For instance, it seems pointless for **ratfor** to prepare a neatly formatted listing of either its input or its output. Users can prepare their own organized input. It is much more important that the language provide free–form input so that you can format it neatly. (In most instances, no one should have to read the output anyway.)

Chapter 11

# The M4 Macro Processor

## 11.1 Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor - replacement of text by other text.

The **m4**(1) macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer. M3 was, in turn, based on a macro processor documented by B. W. Kernighan and P. J. Plauger (*Software Tools*, Addison-Wesley, Inc., 1976). Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion contained in that source.

A suitable front end for Ratfor and C, **m4** has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of **m4** is to copy its input to its output. As it reads the input, however, it checks each alphanumeric "token" (i.e., string of letters and digits). If a token is the name of a macro, then **m4** replaces that name by its defining text, and pushes the resulting string back onto the input for rescanning. You may call macros with arguments; **m4** collects the arguments and substitutes into the right places in the defining text before rescanning the text.

M4 provides a collection of about twenty built-in macros that perform various useful operations. You can also define new macros. Built-ins and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

## 11.2 Usage

Basic usage of **m4** is as follows:

```
% m4 [files] <RETURN>
```

Each argument file is processed in order; if there are no arguments, or if a dash (–) is used as an argument, the standard input is read. The processed text is written on the standard output, which may be captured for subsequent processing with

```
% m4 [files] > outputfile <RETURN>
```

## 11.3 Defining Macros

The primary built–in function of **m4** is **define**, used for defining new macros. Thus,

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* are replaced by *stuff*. The *name* must be alphanumeric and must begin with a letter (an underscore also counts as a letter). The *stuff* argument is any text that contains balanced parentheses; it may stretch over multiple lines. Therefore, as a typical example, this defines *N* to be 100, and uses this "symbolic constant" in a later **if** statement:

```
define(N, 100)
 ...
if (i > N)
```

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built–in name isn't followed immediately by a left parenthesis, **m4** assumes that it has no arguments. This is the situation for *N* above. It is actually a macro with no arguments. When used, it need not be followed by ellipses (...).

**M4** recognizes a macro name only if it's surrounded by non–alphanumerics. Thus, in this example, the variable *NNN* is absolutely unrelated to the defined macro *N*, even though it contains many *N*'s:

```
define(N, 100)
 ...
if (NNN > 100)
```

You may also define things in other ways, e.g., this defines both *M* and *N* to be 100:

```
define(N, 100)
define(M, N)
```

But, what if *N* is redefined? In other words, is *M* defined as *N* or as 100? In **m4**, the latter is true; *M* is 100, so even if *N* subsequently changes, *M* doesn't. This happens because **m4** expands macro names into their defining text as soon as possible. Thus, when the string *N* is seen as the arguments of **define** are being collected, it is immediately replaced by 100, as if you had typed this in the first place:

```
define(M, 100)
```

If this isn't what you really want, there are two ways out of it. The first, specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when you ask for *M* later, you'll always get the value of *N* at that time (because the *M* is replaced by *N* which is replaced by 100).

## 11.4 Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotes (‘ ’) is not expanded immediately, but has the quotes stripped off. If you type

```
define(N, 100)
define(M, ‘N’)
```

the quotes around the *N* are stripped off as the argument is being collected, but they have served their purpose, and *M* is defined as the string *N*, not 100. **M4** always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you must quote it in the input, as in

```
‘define’ = 1;
```

To further illustrate this, consider redefining *N* this way:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

M4 ignores this statement, since you can only define things that look like names. But, to really redefine *N*, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(‘N’, 200)
```

In **m4**, it is often wise to quote the first argument of a macro. If single quotes are not convenient, you can change the quote characters with the built-in **changequote**:

```
changequote([, ])
```

This makes the new quote characters the left and right brackets. You can restore the original characters by typing:

```
changequote
```

Two additional built-ins relate to **define**. The **undefine** built-in

```
undefine(‘N’)
```

removes the definition of some macro or built-in.

This removes the definition of *N*. You can remove a built-in with **undefine**, as in

```
undefine(‘define’)
```

but once you remove one, you can never get it back.

The built-in **ifdef** lets you determine if a macro is currently defined. In particular, **m4** has predefined the name *unix* on the corresponding system:

```
        ifdef('unix', 'define(wordsize,16)' )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

The **ifdef** actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
        ifdef('unix', on UNIX, not on UNIX)
```

## 11.5 Arguments

So far we have discussed the simplest form of macro processing: replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of *$n* is replaced by the *n*th argument when the macro is actually used. Thus, the macro *bump*, defined as

```
        define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1, so

```
        bump(x)
```

is

```
        x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is *$0*, although that is less commonly used.) Arguments not supplied are replaced by null strings. Therefore, we can define a macro *cat*, which simply concatenates its arguments, like this:

```
        define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
        cat(x, y, z)
```

is equivalent to

```
        xyz
```

Since no corresponding arguments were provided, *$4* through *$9* are null.

M4 discards leading unquoted blanks, tabs, or newlines that occur during argument collection. It retains all other white space. Thus

```
        define(a,   b   c)
```

defines *a* to be *b   c*.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
        define(a, (b,c))
```

there are only two arguments; the second is literally *(b,c)*. And, of course, a bare comma or parenthesis can be inserted by quoting it.

## 11.6 Arithmetic Built-ins

M4 provides two built-in functions for performing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus, to define a variable as "one more than N", write

```
define(N, 100)
define(N1, 'incr(N)')
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides these operators (in decreasing order of precedence):

```
unary + and -
** or ^ (exponentiation)
*  /  % (modulus) +
- ==  !=  <  <=  >  >=
!         (not)
& or && (logical and)
| or ||   (logical or)
```

You may use parentheses to group operations where needed. All operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX software.

As a simple example, suppose we want *M* to be *2**N+1*. Then,

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

**Note:** As a matter of principle, we recommend quoting the defining text for a macro unless it is very simple indeed (e.g., just a number).

## 11.7 File Manipulation

You can include a new file in the input at any time by using the built-in **include**:

```
include(filename)
```

This inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (i.e., its replacement text) is the contents of the file; this can be captured in definitions and the like.

A fatal error occurs if the file named in **include** cannot be accessed. To control this, you may use the alternate form **sinclude** ("silent include"), which says nothing and continues if it can't access the file.

You can also divert the output of **m4** to temporary files during processing, and output the collected material upon command. **M4** maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. You can, however, bring back diversions at any time (i.e, append them to the current diversion). Thus,

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted text. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

## 11.8 System Command

You can run any program in the local operating system with the **syscmd** built-in. For example, to run the UNIX system **date**(1) command, use this:

```
syscmd(date)
```

Normally, **syscmd** is used to create a file for a subsequent **include**.

To help make unique filenames, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp**(3). Thus, a string of *XXXXX* in the argument is replaced by the process id of the current process.

## 11.9 Conditionals

A built-in called **ifelse** lets you perform arbitrary conditional testing. Its simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If these are identical, **ifelse** returns the string *c*; otherwise it returns *d*. Thus, we might define a macro called *compare*, which compares two strings and returns "yes" or "no" if they are the same or different.

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent too-early evaluation of **ifelse**. If the fourth argument is missing, it is treated as empty.

An **ifelse** can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. Here, if the string *a* matches the string *b*,

```
ifelse(a, b, c, d, e, f, g)
```

the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

## 11.10 String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus,

```
len(abcdef)
```

is 6, and

```
len((a,b))
```

is 5.

The built-in **substr** can be used to produce substrings of strings. The **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero), and is *n* characters long. If *n* is omitted, the rest of the string is returned, so

```
substr(‘now is the time’, 1)
```

is

```
ow is the time
```

If *i* or *n* are out of range, various sensible things happen.

The **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs, or -1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

```
translit(s, f, t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. This replaces the vowels by the corresponding digits:

```
translit(s, aeiou, 12345)
```

If *t* is shorter than *f*, characters that don't have an entry in *t* are deleted; as a limiting case, if *t* isn't present at all, characters from *f* are deleted from *s*. So the following deletes vowels from *s*:

```
translit(s, aeiou)
```

A built-in called **dnl** deletes all characters that follow it up to and including the next newline. It is most useful for discarding empty lines that otherwise tend to clutter up **m4** output. For example, if you write

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line isn't part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines disappear. Another way to achieve this, is

```
divert(-1)
        define(...)
        ...
divert
```

## 11.11 Printing

The built-in **errprint** writes its arguments on the standard error file. Thus, you can say

```
errprint('fatal error')
```

The built-in **dumpdef** is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise, you get the ones you name as arguments. Don't forget to quote the names!

## 11.12 Summary of Built-ins

The following summarizes the usage and function of built-in commands available to users of **m4**.

| | |
|---|---|
| **changequote**(*L*, *R*) | Restores original characters or makes new quote characters the left and right brackets |
| **define**(*name*, *replacement*) | Defines new macros. |
| **divert**(*number*) | Diverts output to 1-out-of-10 diversions. |
| **divnum** | Returns the number of the currently active diversion. |
| **dnl** | Reads and discards characters up to and including the next newline. |
| **dumpdef**(*'name'*, *'name'*, ...) | Dumps the current names and definitions of items named as arguments. |
| **errprint**(*s*, *s*, ...) | Prints its arguments on the standard error file. |
| **eval**(*numeric expression*) | Prints arbitrary arithmetic on integers. |
| **ifdef**(*'name'*, *this if true*, *this if false*) | Determines whether a macro is currently defined. |
| **ifelse**(*a*, *b*, *c*, *d*) | Performs arbitrary conditional testing. |
| **include**(*file*) | Returns the contents of the file named in the argument. A fatal error occurs if the filename cannot be accessed. |
| **incr**(*number*) | Returns the value of its argument incremented by 1. |
| **index**(*s1*, *s2*) | Returns the position where the second argument begins in the first argument pf index. |
| **len**(*string*) | Returns the number of characters that makes its argument. |
| **maketemp**(...*XXXXX*...) | Facilitates making unique filenames. |
| **sinclude**(*file*) | Returns the contents of the file named in the argument. The macro remains silent and continues if the file is inaccessible. |
| **substr**(*string*, *position*, *number*) | Produces substrings of strings. |

| | |
|---|---|
| **syscmd**(*s*) | Executes the UNIX system command given in the first argument. |
| **translit**(*str*, *from*, *to*) | Performs character transliteration. |
| **undefine**(‘*name*’) | Removes user-defined or built-in macro definitions. |
| **undivert**(*number*,*number*,...) | Discards the diverted text. |

Chapter 12

# Bc: An Arbitrary Precision Desk–Calculator Language

## 12.1 Introduction

**Bc**(1) is a language and a compiler for doing arbitrary precision arithmetic. The output of the compiler is interpreted and executed by a collection of routines called **dc**(1), which can input, output, and do arithmetic on indefinitely large integers and scaled fixed–point numbers. These routines are themselves based on a dynamic storage allocator. Overflow doesn't occur until all available core storage is exhausted.

The compiler is by no means intended to provide a complete programming language; it is a minimal language facility that was written in **yacc**(1).

The language has a complete control structure as well as immediate–mode operation. Functions can be defined and saved for later execution. Two five hundred–digit numbers can be multiplied to give a thousand digit result in about ten seconds.

**Bc** is most helpful in doing computation with large integers, and especially when computation must be accurate to many decimal places. It is also helpful in converting numbers from one base to another.

A small collection of library functions is available. The library currently consists of *sine* (s), *cosine* (c), *arctangent* (a), *natural logarithm* (l), *exponential* (e) and *Bessel* functions of integer order (j(n,x)). To load a set of library functions, type

```
% bc -l <RETURN>
```

A scaling provision permits the use of decimal point notation. (The library mentioned above sets the scale to 20; you may reset it.) Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of the UNIX operating system.

The syntax of **bc** was deliberately selected to agree substantially with the C language. In fact, the following constructs work in **bc** just as they do in the C language. Consult the next section of this chapter or *The C Programming Language* by B. W. Kernigham and D. M. Ritchie (Prentice–Hall, 1978) for their specific workings.

```
x=y=z    is the same as    x=(y=z)
x =+ y                     x=x+y
x=- y                      x = x-y
x =* y                     x = x*y
x =/ y                     x = x/y
x =% y                     x = x%y
x =^y                      x = x^y
x++                        (x=x+1)-1
x--                        (x=x-1)+1
++x                        x = x+1
--x                        x = x-1
```

**Note:** Even if you don't intend to use the constructs, typing one inadvertently may produce unexpected results. Furthermore, in some of these constructions, spaces are significant. For example, x=-y and x= -y are very different; the first replaces x by x-y and the second by -y.

If you type

% **bc** *file* ... <RETURN>

**bc** reads and executes the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions. To exit a **bc** program, type "quit".

## 12.2 Simple Computations With Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators –, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward 0. Division by 0 produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). This expression is interpreted to mean that –3 is to be added to 7:

```
7+-3
```

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with a caret (^) having the greatest binding power, then asterisk (*)

and percent (%) and slash (/), and finally plus (+) and minus (-). Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. These expressions

```
a^b^c
a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c
(a*b)*c
```

**Bc** shares with Fortran and C the undesirable convention that

```
a/b*c
```

is equivalent to

```
(a/b)*c
```

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

increases by three the value of the contents of the register *x*. When, as in this case, the outermost operator is an equal sign (=), the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

If an assignment statement is placed in parentheses, it then has a value and can be used anywhere that an expression can. For example, this line makes the indicated assignment, and it also prints the resulting value:

```
(x=y+17)
```

To illustrate a use of the value of an assignment statement even when parentheses are not used, consider the following:

```
x = a[i=i+1]
```

This causes a value to be assigned to *x*; it also increments *i* before using it as a subscript.

The built-in square root function truncates its result to an integer (but also see scaling below). The lines

```
x = sqrt(191) x
```

produce the printed result

```
13
```

## 12.3 Bases

**Bc** uses special internal quantities, called *ibase* and *obase*. The contents of *ibase*, initially set to 10, determine the base used for interpreting numbers read in. Thus,

```
ibase = 8 11
```

produces the output line

```
9
```

and you are set up to do octal to decimal conversions. Don't, however, try to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement is ineffectual. For those who deal in hexadecimal notation, the characters A-F are permitted in numbers (regardless of base in effect) and are interpreted as digits having values 10-15 respectively. Typing this changes you back to decimal input base regardless of current input base:

```
ibase = A
```

Negative and large positive input bases are permitted but are useless. No mechanism exists for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of *obase*, initially set to 10, serve as the base for output numbers. Thus,

```
obase = 16 1000
```

produces the output line

```
3E8
```

which is to be interpreted as a 3–digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting *obase* to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Continued lines end with a backslash (\). Decimal output conversion is almost instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non–decimal output conversion of a 100–digit number takes about three seconds.

Remember that *ibase* and *obase* have no effect on the course of internal computation or the evaluation of expressions, but only affect input and output conversion, respectively.

## 12.4 Scaling

A third special internal quantity called *scale* is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its *scale*.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the certain rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity *scale*. The scale of a quotient is the contents of the internal quantity *scale*. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of *scale*.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation (and not rounding) is performed.

The contents of *scale* must be no greater than 99 and no less than 0. It is initially set to 0. If you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities *scale*, *ibase*, and *obase* can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of *scale* by one, and the line

```
scale
```

causes the current value of *scale* to be printed.

The value of *scale* retains its meaning as a number of decimal digits to be retained in internal computation even when *ibase* or *obase* aren't equal to 10. The internal computations (conducted in decimal, regardless of the bases) are performed to the number of decimal digits specified, never hexadecimal or octal or any other kind of digits.

## 12.5 Functions

The name of a function is a single lower-case letter. Bc permits function names to collide with simple variable names. Twenty-six different defined functions are permitted, in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace ( } ). Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of these two forms:

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared automatic by a statement of the form

```
auto x,y,z
```

There can be only one "auto" statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to 0 on entry to the function, and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. Parameters named in a function definition are treated the same as the automatic variables of that function, with the single exception that parameters are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, is the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them, e.g., b().

If the function *a* above is defined, then the line

```
a(7,3.14)
```

causes the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

causes the value of x to become 60.

## 12.6 Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. **Bc** permits the names of arrays to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to 0 and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[]) define f(a[]) auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

## 12.7 Control Statements

You can use "if", "while", and "for" statements to alter the flow within programs or to cause iteration. The range of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written this way:

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for "equal to" and != stands for "not equal to". The meaning of the remaining relational operators is clear.

**Note:** Beware of using = instead of == in a relational. Although both of them are legal (so you do not get a diagnostic message), = does not do a comparison.

The "if" statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The "while" statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The "for" statement begins by executing "expression1". The relation is then tested; if true, the statements in the range of the "for" are executed. Then "expression2" is executed. The relation is tested, and so on. The typical use of the "for" statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which prints the integers from 1 to 10.

Here are some examples of the use of the control statements:

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

This line prints *a* factorial if *a* is a positive integer:

```
f(a)
```

Here is the definition of a function that computes values of the binomial coefficient (assuming *m* and *n* are positive integers):

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

## 12.8 Summary of Important Features

This section contains a synopsis of the important features and constructs associated with the use of **bc**.

### 12.8.1 Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

#### 12.8.1.1. Comments

**Bc** uses a comment convention identical to that of C and of PL/I. Thus, comments begin with /* and end with */ characters.

#### 12.8.1.2 Identifiers

There are three kinds of identifiers - ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower–case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at 0 so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named *x*, an array named *x* and a function named *x*, all of which are separate and distinct.

#### 12.8.1.3 Keywords

The following are reserved keywords:

```
ibase       if
obase       break
scale       define
sqrt        auto
length      return
while       quit
for
```

### 12.8.1.4 Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits *A-F* are also recognized as digits with values 10-15, respectively.

## 12.8.2 Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

### 12.8.2.1 Primitive Expressions

*Named expressions* These primitive expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named. The following are named expressions:

| | |
|---|---|
| *identifiers* | Simple identifiers are named expressions. They have an initial value of 0. |
| *array-name*[*expression*] | Array elements are named expressions. They have an initial value of 0. |
| **scale** | This internal register is a named expression. It is the number of digits after the decimal point to be retained in arithmetic operations. It has an initial value of 0. |
| **ibase** and **obase** | These internal registers are the input and output number radix respectively. Both have initial values of 10. |

*Function calls* A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments as follows:

*function-name*([*expression*[,*expression*...]])

A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. Thus, changes made to the formal parameters don't affect the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is 0 if no expression is provided or if there is no return statement.

Types of function calls are defined as follows:

| | |
|---|---|
| **sqrt**(*expression*) | The result is the square root of the expression, and is truncated in the least significant decimal place. The scale of the result is the larger of the scale of the expression or the value of **scale**. |
| **length**(*expression*) | The result is the total number of significant decimal digits in the expression. The scale of the result is 0. |

| | |
|---|---|
| **scale**(*expression*) | The result is the scale of the expression. The scale of the result is 0. |

*Constants* These are another form or primitive expressions.

*Parentheses* An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

### 12.8.2.2 Unary Operators

The unary operators bind right to left.

| | |
|---|---|
| *-expression* | The result is the negative of the expression. |
| ++*named–expression* | The named expression is incremented by one. The result is the value of the named expression after incrementing. |
| *––named–expression* | The named expression is decremented by one. The result is the value of the named expression after decrementing. |
| *named–expression*++ | The named expression is incremented by one. The result is the value of the named expression before incrementing. |
| *named–expression––* | The named expression is decremented by one. The result is the value of the named expression before decrementing. |

### 12.8.2.3 Exponentiation Operator

The exponentiation operator binds right to left.

*expression ^ expression* The result is the first expression raised to the power of the second expression. The second expression must be an integer. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, then the scale of the result is:

min(*axb*,max(**scale**,*a*))

### 12.8.2.4 Multiplicative Operators

The operators *, /, % bind left to right.

*expression * expression* The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result is:

min(*a*+*b*,max(**scale**,*a*,*b*))

*expression / expression* The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

*expression % expression* The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a-a/b*b*. The scale of the result is the sum of the scale of the divisor and the value of **scale**.

### 12.8.2.5 Additive Operators

The additive operators bind left to right.

| | |
|---|---|
| *expression* + *expression* | The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions. |
| *expression* – *expression* | The result is the difference of the two expressions, and the scale of the result is the maximum of the scales of the expressions. |

### 12.8.2.6 Assignment Operators

The assignment operators bind right to left.

| | |
|---|---|
| *named–expression* = *expression* | This results in assigning the value of the expression on the right to the named expression on the left. |
| *named–expression* =+ *expression*<br>*named–expression* =– *expression*<br>*named–expression* =* *expression*<br>*named–expression* =/ *expression*<br>*named–expression* =% *expression*<br>*named–expression* =^ *expression* | The result of these expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the equal sign. |

## 12.8.3 Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

*expression* < *expression*
*expression* > *expression*
*expression* <= *expression*
*expression* >= *expression*
*expression* == *expression*
*expression* != *expression*

## 12.8.4 Storage Classes

**Bc** has only two storage classes: global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to it. All other identifiers are assumed to be global and are available to all functions. All identifiers, global and local, have an initial value of 0. Identifiers declared as **auto** are allocated on entry to the function and released upon return from the function. Thus, they don't retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in **bc** don't work in exactly the same way as they do in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

## 12.8.5 Statements

Normally, statements are typed one to a line. You may also type several statements on a line separated by a semicolon or newline. Except where altered by control statements, execution is sequential.

### 12.8.5.1 Expression Statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

### 12.8.5.2 Compound Statements

Statements may be grouped together and used when one statement is expected by surrounding them with braces ( { } ).

### 12.8.5.3 Quoted String Statements

"any string"

This statement prints the string inside the quotes.

### 12.8.5.4 If Statements

**if**(*relation*)*statement*

The substatement is executed if the relation is true.

### 12.8.5.5 While Statements

**while**(*relation*)*statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

### 12.8.5.6 For Statements

**for**(*expression; relation; expression*)*statement*

The for statement is the same as

```
first-expression
while(relation) {
        statement
        last-expression
}
```

All three expressions must be present.

### 12.8.5.7 Break Statements

**break**

A **break** causes termination of a **for** or **while** statement.

### 12.8.5.8 Auto Statements

**auto** *identifier*[,*identifier*]

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The **auto** statement must be the first statement in a function definition.

### 12.8.5.9 Define Statements

**define**([*parameter*[,*parameter...*]]){
    *statements*}

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

### 12.8.5.10 Return Statements

**return**

**return**(*expression*)

The **return** statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

### 12.8.5.11 Quit Statement

The **quit** statement stops execution of a **bc** program and returns control to UNIX system processing when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

Chapter 13

# Dc: An Interactive Desk Calculator

An arbitrary precision arithmetic package, **dc**(1) is implemented in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily, **dc** operates on decimal integers, but you may specify an input base, output base, and a number of fractional digits to be maintained.

A language called **bc**(1) accepts programs written in the familiar style of higher-level programming languages and compiles output interpreted by **dc**. (**Bc** was first introduced in a paper entitled *BC – An Arbitrary Precision Desk-Calculator Language*, written by L. L. Cherry and R. Morris.) Some of the commands described in this chapter were designed for the compiler interface and are difficult for human users to manipulate.

Numbers typed into **dc** are put on a push-down stack. **Dc** commands take the top number or two off the stack, perform the desired operation, and push the result on the stack. If you supply a filename argument, **dc** takes input from that file until its end, and then uses the standard input.

## 13.1 User–Oriented Commands

This section describes **dc** commands intended for human use. Any number of commands are permitted on a line. Blanks and newline characters are ignored except within numbers and in places where a register name is expected.

**number** Push the value of the number onto the main stack. A number is an unbroken string of the digits 0–9, and the capital letters A-F (treated as digits with values 10–15 respectively). The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

| | |
|---|---|
| + - * % ^ | Add (+), subtract (–), multiply (*), divide (/), remainder (%), or exponentiate (^) the top two values on the stack. Pop the two entries off the stack; push the result on the stack in their place. The result of a division is an integer truncated toward zero. An exponent must not have any digits after the decimal point. |
| s*x* | Pop the top of the main stack and store it in a register named *x*, where *x* may be any character. If the **s** is capitalized, treat *x* as a stack and push the value onto it. Any character (including blank or newline) is a valid register name. |
| l*x* | Push the value in register *x* onto the stack, without altering the register *x* itself. If the **l** is capitalized, treat register *x* as a stack and pop its top value onto the main stack. All registers start with empty value, which is treated as a zero by the command **l** and as an error by the command **L**. |
| **d** | Duplicate the top value on the stack. |
| **p** | Print the top value on the stack, without changing the value itself. |
| **f** | Print all values on the stack and in registers. |
| **x** | Treat the top element of the stack as a character string; remove it from the stack, and execute it as a string of **dc** commands. |
| [ ... ] | Put the bracketed character string onto the top of the stack. |
| **q** | Exit the program. If executing a string, pop the recursion level by two. If **q** is capitalized, pop the top value on the stack and then pop the string execution level by that value. |
| <*x*>*x*=*x*!<*x*!>*x*!=x | Pop and compare the top two elements of the stack. If they obey the stated relation, execute register *x*. The exclamation point represents negation. |
| v | Replace the top element on the stack by its square root. Truncate the square root of an integer to an integer. (See detailed desciption below on treatment of numbers with decimal points.) |
| ! | Interpret the rest of the line as a UNIX command. Return control to **dc** when the UNIX command terminates. |
| **c** | Pop all values on the stack (the stack becomes empty). |
| **i** | Pop the top value on the stack and use it as the number radix for further input. If **i** is capitalized, push the value of the input base onto the stack. Currently, no mechanism exists for the input of arbitrary numbers in bases less than 1 or greater than 16. |
| **o** | Pop the top value on the stack and use it as the number radix for further output. If **o** is capitalized, push the value of the output base onto the stack. |
| **k** | Pop the top of the stack, and use that value as a scale factor to influence the number of decimal places maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, push the value of the scale factor onto the stack. |

| | |
|---|---|
| z | Push the value of the stack level onto the stack. |
| ? | Take a line of input from the input source (e.g., the keyboard) and execute it. |

## 13.2 Internal Representation of Numbers

**Dc** uses a dynamic storage allocator for storing numbers internally. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, all digits should be in the range 0–99, and the number should have no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation (analogous to two's complement notation for binary numbers). The high order digit of a negative number is always –1 and all other digits are in the range 0–99. The digit preceding the high order –1 digit is never a 99. The representation of –157 is 43,98,–1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,3 where the scale has been italicized to emphasize the fact that it isn't the high order digit. The value of this extra byte is called the scale factor of the number.

## 13.3 The Allocator

**Dc** uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and **dc** is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2.

When a string request is made, the allocator first checks the free list for a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has one next to it in core and, if free, can be combined with it to make a string twice as long. (This is an implementation of the "buddy system" of allocation detailed by K. C. Knowlton in *A Fast Storage Allocator*, Comm. ACM 8, Oct. 1965).

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in **dc**. If the allocator runs out of headers while trying to allocate a string, it also asks the system for more space.

The allocator contains routines for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines. The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

## 13.4 Internal Arithmetic

All arithmetic operations are done on integers. The operand(s) needed for the operation is popped from the main stack and its scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required (as it is for addition), zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called *scale* plays a part in the results of most arithmetic operations. It is the bound on the number of decimal places retained in arithmetic computations. This register may be set to the number on the top of the stack truncated to an integer with the **k** command; **k** may be used to push the value of *scale* on the stack. *Scale* must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations include the exact effect of *scale* on the computations.

### 13.4.1 Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,–1 by the digit –1. Digits not in the range 0–99 must be brought into that range, propagating any carries or borrows that result.

### 13.4.2 Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum, which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register *scale* and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

### 13.4.3 Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity *scale*. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out to be one unit too low, but if it is, the next trial quotient is larger than 99 and this is adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

### 13.4.4 Remaindering

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

### 13.4.5 Square Roots

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity *scale* and the scale of the operand. The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

```
x sub {n+1} ~=~ half ( x sub n + y over x sub n )
```

The initial guess is found by taking the integer square root of the top two digits.

### 13.4.6 Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

## 13.5 Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a minus sign (-). The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example, be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** pushes the value of the input base on the stack.

## 13.6 Output Commands

The command **p** causes the top of the stack to be printed. It doesn't remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It works correctly for any base. The command **O** pushes the value of the output base on the stack.

## 13.7 Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they don't affect arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

## 13.8 Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register **x**. The *x* can be any character. An **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

## 13.9 Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

## 13.10 Subroutine Definitions and Calls

Enclosing a string in square brackets ( [ ] ) pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

## 13.11 Programming dc

Use the load and store commands together with square brackets ( [ ] ) to store strings; **x** to execute; and the testing commands <, >, =, !<, !>, and != to program **dc**. The **x** command assumes the top of the stack is a string of **dc** commands and executes it. The testing commands compare the top two elements on the stack, and if the relation holds, execute the register that follows the relation. Thus, this prints the numbers 0–9:

```
[lip1+  si  li10>a]sa 0si  lax
```

## 13.12 Push–Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push–down registers and arrays. In addition to the stack that commands work on, **dc** uses individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers, but not as push–down stacks. An **l** doesn't affect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are colon (:) and semi–colon (;). A :*x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. A ;*x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

## 13.13 Miscellaneous Commands

An exclamation point (!) interprets the rest of the line as a UNIX command and passes it to UNIX software to execute. One other compiler command is Q. This command uses the top of the stack as the number of levels of recursion to skip.

## 13.14 Design Choices

The dynamic storage allocator lets a general purpose program be used for a variety of other tasks. It has some value for input and for compiling (i.e., the bracket [...] commands) where a string's length cannot be known in advance. Thus, at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

Using 100 as a base for internal arithmetic may not seem very beneficial. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

A stack–type arithmetic design allowed all **dc** commands from addition to subroutine execution to be implemented in essentially the same way. As a result, there was much logical separation of the final program into modules with very little communication between modules.

The lack of interaction between the scale and the bases provided an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation that had global notions of scale and base didn't work well. If the value of *scale* was to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale isn't used for any essential purpose by any part of the program; it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The scales of the results of arithmetic were designed so that significant digits are never thrown away if, on appearance, you actually want them. Thus, if you wanted to add the numbers 1.5 and 3.517, it seemed reasonable to provide the result 5.017 without requiring that you necessarily specify obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless you asked for them by specifying a value for *scale*. Square root can be handled like multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way for the program to guess how many places you want. In this case only, you must specify a *scale* to get any decimal places at all.

The scale of remainder makes it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

Chapter 14

# Curses: Screen Functions With An "Optimal" Cursor

## 14.1 Overview

The **curses**(3X) package helps C programmers do the most common type of terminal dependent functions - movement optimization and optimal screen updating - with nearly as much ease as is necessary to simply print or read text.

The package is composed of three parts: one for screen updating; another for screen updating with user input; and a third for cursor motion optimization. You can use the motion optimization without using either of the other two. You can also perform screen updating and input without any programming knowledge of the motion optimization, or indeed of the database itself.

In the **curses** package, a window is defined as an internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

Furthermore, a terminal (or terminal screen) is defined as the package's idea of what the terminal's screen currently looks like, i.e., what you see now. A screen is a subset of windows that are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for you.

### 14.1.1 Compiling Data

To use the library, you must have certain types and variables defined. Thus, you need

```
#include <curses.h>
```

at the top of the program source. The header file *<curses.h>* must include *<sgtty.h>*. Compilations should have the following form:

% **cc** [ *flags* ] *file* ... **-lcurses -ltermlib** <RETURN>

**Note:** The screen package also uses the standard I/O library, so *<curses.h>* includes *<stdio.h>*. You need not also include it, although it is harmless if you do.

### 14.1.2 Screen Updating

To update the screen optimally, the routines must know what the screen currently looks like and what you want it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined to describe a window image to the routines, including its starting position on the screen (the (y, x) coordinates of the upper left hand corner) and its size. One of these, *curscr* (current screen), is a screen image of how the terminal currently looks. By default, *stdscr* (standard screen) is provided as a screen on which to make changes.

A window is a purely internal representation, used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen, but is more like an array of characters on which to make changes.

When you have a window that describes what some part the terminal should look like, *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. The *refresh()* routine makes the terminal, in the area covered by the window, look like that window.

**Note:** Changing something on a window does not change the terminal. Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This lets you maintain several different ideas of what a portion of the terminal screen should look like. You can make changes to windows in any order, regardless of motion efficiency. Then, at will, you can effectively say "make it look like this," and let the package worry about the best way to do this.

### 14.1.3 Naming Conventions

Although the routines can use several windows, only two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what you want the terminal to look like next. You should never really access *curscr* directly. Make changes to the appropriate screen, and then call the routine *refresh()* (or *wrefresh()*).

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, call *addch()* with the desired character. If a different window is to be used, use the routine *waddch()* (window-specific *addch()*).

**Note:** Actually, *addch()* is a "#define" macro with arguments, as are most of the functions that deal with *stdscr* as a default.

This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines that do not do this are those to which a window must always be specified.

To move the current (y, x) coordinates from one point to another, use the *move()* and *wmove()* routines. It is often desirable to first move and then to perform some I/O operation. To avoid clumsiness, most I/O routines can be preceded by the prefix "mv" and the desired (y, x) coordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

**Note:** The window description pointer (*win*) comes before the added (y, x) coordinates. If such pointers are need, they are always the first parameters passed.

## 14.2 Variables

Many variables that describe the terminal environment are available:

| Type | Name | Description |
|---|---|---|
| WINDOW * | curscr | Current version of the screen (terminal screen). |
| WINDOW * | stdscr | Standard screen. Most updates are usually done here. |
| char * | Def_term | Default terminal type if type cannot be determined. |
| bool | My_term | Use the terminal specification in Def_term as terminal, regardless of real terminal type. |
| char * | ttytype | Full name of the current terminal. |
| int | LINES | Number of lines on the terminal. |
| int | COLS | Number of columns on the terminal. |
| int | ERR | Error flag returned by routines on a fail. |
| int | OK | Error flag returned by routines when things go right. |

Several "#define" constants and types are also generally useful:

| | |
|---|---|
| reg | storage class "register" (e.g., *reg int i;*) |
| bool | boolean type, actually a "char" (e.g., *bool doneit;*) |
| TRUE | boolean "true" flag (1) |
| FALSE | boolean "false" flag (0) |

**addch(ch)**
*char* *ch;*

**waddch(win, ch)**
*WINDOW** *win;*
*char* *ch;*

Add the character *ch* on the window at the current (y, x) coordinates. If the character is a newline (\n), clear the line to the end. Then, change the current (y, x) coordinates to the beginning of the next line if newline mapping is on, or to the next line at the same x coordinate if it is off. A return (\r) moves to the beginning of the line on the window. Tabs (\t) are expanded into spaces in the normal tabstop positions of every 8 characters. Return ERR if screen will be made to scroll illegally.

**addstr(str)**
*char** *str;*

**waddstr(win, str)**
*WINDOW** *win;*
*char** *str;*

Add the string pointed to by *str* on the window at the current (y, x) coordinates. In this case, put on as much as possible. Return ERR if screen will be made to scroll illegally.

**box(win, vert, hor)**
*WINDOW** *win;*
*char* *vert, hor;*

Draw a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, leave the corners blank (to avoid a scroll).

**clear()**

**wclear(win)**
*WINDOW** *win;*

Reset the entire window to blanks. If *win* is a screen, this sets the clear flag, which causes a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) coordinates to (0, 0).

**clearok(scr, boolf)**
*WINDOW** *scr;*
*bool* *boolf;*

Set the clear flag for the screen *scr*. If *boolf* is TRUE, force a clear-screen to be printed on the next *refresh()*; stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, doesn't alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call causes a clear-screen, even if the window passed to *refresh()* is not a screen.

**clrtobot()**

**wclrtobot(win)**
*WINDOW*　win;*

Wipe the window clear from the current (y, x) coordinates to the bottom. Do not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "mv" command.

**clrtoeol()**

**wclrtoeol(win)**
*WINDOW*　win;*

Wipe the window clear from the current (y, x) coordinates to the end of the line. This has no associated "mv" command.

**delch()**

**wdelch(win)**
*WINDOW*　win;*

Delete the character at the current (y, x) coordinates. Shift each character after it on the line to the left, and make the last character blank.

**deleteln()**

**wdeleteln(win)**
*WINDOW*　win;*

Delete the current line. Move every line below the current one up, and make the bottom line blank. Don't change the current (y, x) coordinates.

**erase()**

**werase(win)**
*WINDOW*　win;*

Erase the window to blanks without setting the clear flag. This is similar to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "mv" command.

**insch(c)**
**charc;**

**winsch(win, c)**
*WINDOW*　win;*
*char　c;*

Insert *c* at the current (y, x) coordinates. Each character after it shifts to the right; the last character disappears. Return ERR if screen will be made to scroll illegally.

**insertln()**

**winsertln(win)**
*WINDOW*　win;*

Insert a line above the current one. Shift every line below the current line down, and make the bottom line disappear. The current line becomes blank, and the current (y, x) coordinates remain unchanged. Return ERR if screen will be made to scroll illegally.

**move(y, x)**
*int* *y, x;*

**wmove(win, y, x)**
*WINDOW** *win;*
*int* *y, x;*

Change the current (y, x) coordinates of the window to (*y*, *x*). Return ERR if screen will be made to scroll illegally.

**overlay(win1, win2)**
*WINDOW** *win1, *win2;*

Overlay *win1* on *win2*. Place the contents of *win1*, insofar as they fit, on *win2* at their starting (y, x) coordinates. Do this non–destructively (i.e., blanks on *win1* leave the contents of the space on *win2* untouched).

**overwrite(win1, win2)**
*WINDOW** *win1, *win2;*

Overwrite *win1* on *win2*. Place the contents of *win1*, insofar as they fit, on *win2* at their starting (y, x) coordinates. Do this destructively (i.e., blanks on *win1* become blank on *win2*).

**printw(fmt, arg1, arg2, ...)**
*char** *fmt;*

**wprintw(win, fmt, arg1, arg2, ...)**
*WINDOW** *win;*
*char** *fmt;*

Perform a *printf()* on the window starting at the current (y, x) coordinates. Use *addstr()* to add the string on the window. Use the field width options of *printf()* to avoid leaving things on the window from earlier calls. Return ERR if screen will be made to scroll illegally.

**refresh()**

**wrefresh(win)**
*WINDOW** *win;*

Synchronize the terminal screen with the desired window. If the window isn't a screen, update only that part covered by it. Return ERR if screen will be made to scroll illegally. Update whatever possible without causing a scroll.

**standout()**

**wstandout(win)**
*WINDOW** *win;*

**standend()**

**wstandend(win)**
*WINDOW** *win;*

Start and stop putting characters onto *win* in standout mode. The *standout()* routine causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). The *standend()* routine stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see next major section).

**crmode()**

**nocrmode()**

Set or unset the terminal to/from *cbreak* mode.

**echo()**

**noecho()**

Set the terminal to echo or not echo characters.

**getch()**

**wgetch(win)**
*WINDOW**  *win;*

Get a character from the terminal and, if necessary, echo it on the window. Return ERR if screen will be made to scroll illegally. Otherwise, return the character gotten. If *noecho* is set, leave the window unaltered. (To retain control of the terminal, one of *noecho*, *cbreak*, or *rawmode* must be set. Otherwise, whatever routine you call to read characters sets *cbreak* for you, and then resets to the original mode when finished.)

**getstr(str)**
*char**  *str;*

**wgetstr(win, str)**
*WINDOW**  *win;*
*char**  *str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. Set tty modes if necessary, and then call *getch()*(or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. Strip the newline off the string. Return ERR if screen will be made to scroll illegally.

**raw()**

**noraw()**

Set or unset the terminal to/from raw mode. (On version 7 UNIX software, this also turns off newline mapping (see *nl()*)).

**scanw(fmt, arg1, arg2, ...)**
*char**  *fmt;*

**wscanw(win, fmt, arg1, arg2, ...)**
*WINDOW**  *win;*
*char**  *fmt;*

Perform a *scanf()* through the window using *fmt*. Use consecutive *getch()*'s (or *wgetch(win)*'s) to do this. Return ERR if screen will be made to scroll illegally.

**delwin(win)**
*WINDOW**  *win;*

Delete the window from existence. Free all resources for future use via **calloc**(3C). If a window has a *subwin()* allocated window inside it, deleting the outer window the subwindow isn't affected, even though this does invalidate it. Therefore, delete subwindows before their outer windows.

**endwin()**

Finish up window routines before exit. Restore the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. You should always call this before exiting. It does not exit, and is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

**getyx(win, y, x)**
*WINDOW** *win;*
*in* *ty, x;*

Put the current (y, x) coordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

**inch()**

**winch(win)**
*WINDOW** *win;*

Return the character at the current (y, x) coordinates on the given window. Don't make any changes to the window. Has no associated "mv".

**initscr()**

Initialize the screen routines. (This must be called before using any screen routines; it initializes necessary terminal–type data.) If standard input is not a tty, set the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is TRUE, always use *Def_term*.

**leaveok(win, boolf)**
*WINDOW** *win;*
*boo* *lboolf;*

Set the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, leave the cursor after the last update on the terminal, and change the current (y, x) coordinates for *win* accordingly. If it is FALSE, move the cursor to the current (y, x) coordinates. (This flag, initially FALSE, retains its value until you change it.)

**longname(termbuf, name)**
*char** *termbuf, *name;*

Fill in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. The routine tells you, in a readable format, what terminal we think you have. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

**mvwin(win, y, x)**
*WINDOW** *win;*
*int* *y, x;*

Move the home position of the window *win* from its current starting coordinates to ( *y, x*). If part or all of the window is off the edge of the terminal screen, return ERR and do not change anything.

*WINDOW **
**newwin(lines, cols, begin_y, begin_x)**
*int* *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), set that dimension to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* x *COLS*, use *newwin(0, 0, 0, 0)*.

**nl()**

**nonl()**

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <RETURN> to <LINE-FEED>. If no mapping is done, *refresh()* can do more optimization; thus, we recommend that you turn it off.

**scrollok(win, boolf)**
*WINDOW** *win;*
*bool* *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

**touchwin(win)**
*WINDOW** *win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

*WINDOW **
**subwin(win, lines, cols, begin_y, begin_x)**
*WINDOW** *win;*
*int* *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) in the middle of the window *win*. (Thus, any change made to either window in the area covered by the subwindow is made on both windows; *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*.) If either *lines* or *cols* is 0 (zero), set that dimension to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively.

**unctrl(ch)**
*char* *ch;*

Return a string that represents *ch*. (This is actually a debug function for the library, but it is of general usefulness.) Make control characters their upper-case equivalents preceded by a caret (^). Leave other letters just as they are. To use *unctrl()*, you must have this in your file:

```
#include <unctrl.h>
```

**gettmode()**

Get the tty stats. This is normally called by *initscr()*.

**mvcur(lasty, lastx, newy, newx)**
*int* *lasty, lastx, newy, newx;*

Move the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from ex(1) version 2.6. You can use this optimization without the screen routines. With the screen routines, you should not call this. The *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

**scroll(win)**
*WINDOW** *win;*

Scroll the window upward one line. You normally need not use this.

**savetty()**

Save the current tty characteristic flags. This is performed automatically by *initscr()* and *endwin()*.

**resetty()**

Restore the tty characteristic flags to what *savetty()* stored. This is performed automatically by *initscr()* and *endwin()*.

**setterm(name)**
*char** *name;*

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

**tstp()**

If the new **tty**(4) driver is in use, save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen; *initscr()* sets the signal SIGTSTP to trap to this routine.

## 14.3 Capabilities Provided by Termcap (BSD4.2 only)

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here. For a full description, see **termcap**(5) in the *DOMAIN/IX Programmer's Reference for BSD4.2*.

### 14.3.1 Overview

Capabilities from **termcap** are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities that have a *P* at the front of their comment. This normally is a number of milliseconds to pad the operation.

In the current system, which has no true programmable delays, we do this by sending a sequence of pad characters. These characters are normally nulls, but they can be changed (specified by *PC*). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes that shift several lines.) This is specified as, e.g., *12** before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say *P**.

## 14.3.2 Variables Set by "setterm()"

| Type | Name | Pad | Description |
|---|---|---|---|
| char * | AL | P* | Add new blank Line |
| bool | AM | | Automatic Margins |
| char * | BC | | Back Cursor movement |
| bool | BS | | BackSpace works |
| char * | BT | P | Back Tab |
| bool | CA | | Cursor Addressable |
| char * | CD | P* | Clear to end of Display |
| char * | CE | P | Clear to End of line |
| char * | CL | P* | CLear screen |
| char * | CM | P | Cursor Motion |
| char * | DC | P* | Delete Character |
| char * | DL | P* | Delete Line sequence |
| char * | DM | | Delete Mode (enter)char * |
| DO | | | DOwn line sequencechar * |
| ED | | | End Delete mode |
| bool | EO | | Erase Overstrikes with ' 'char * |
| EI | | | End Insert mode |
| char * | HO | | HOme cursor |
| bool | HZ | | HaZeltine - braindamage |
| char * | IC | P | Insert Character |
| bool | IN | | Insert-Null blessing |
| char * | IM | | Enter Insert Mode (IC usually set, too) |
| char * | IP | P* | Pad after char Inserted using IM+IE |
| char * | LL | | Quick to Last Line, column 0 |
| char * | MA | | Ctrl character MAp for cmd mode |
| bool | MI | | Move in Insert mode |
| bool | NC | | No Cr: \r sends \r\n then eats \n |
| char * | ND | | Non-Destructive space |
| bool | OS | | OverStrike works |
| char | PC | | Pad Character |
| char * | SE | | Standout End (may leave space) |
| char * | SF | P | Scroll Forwards |
| char * | SO | | Stand Out begin (may leave space) |
| char * | SR | P | Scroll in Reverse |
| char * | TA | P | TAb (not ^I or with padding) |
| char * | TE | | Terminal address enable Ending sequence |
| char * | TI | | Terminal address enable Initialization |
| char * | UC | | Underline a single Character |
| char * | UE | | Underline Ending sequence |
| bool | UL | | UnderLining works even though !OS |
| char * | UP | | UPline |
| char * | US | | Underline Starting sequence |
| char * | VB | | Visible Bell |
| char * | VE | | Visual End sequence |
| char * | VS | | Visual Start sequence |
| bool | XN | | Newline gets eaten after wrap |

**Note:** US and UE, if they don't exist in the **termcap** entry, are copied from SO and SE in *setterm()*. Names starting with *X* are reserved for special cases.

### 14.3.3 Variables Set by "gettmode()"

| Type | Name | Description |
|---|---|---|
| bool | NONL | Term can't hack linefeeds doing a CR |
| bool | GT | Gtty indicates Tabs |
| bool | UPPERCASE | Terminal generates only uppercase letters |

## 14.4 The WINDOW Structure

```
# define          WINDOW    struct _win_st
struct _win_st {
                  short     _cury, _curx;
                  short     _maxy, _maxx;
                  short     _begy, _begx;
                  short     _flags;
                  bool      _clear;
                  bool      _leave;
                  bool      _scroll;
                  char      **_y;
                  short     *_firstch;
                  short     *_lastch;
};
# define          _SUBWIN       01
# define          _ENDLINE      02
# define          _FULLWIN      04
# define          _SCROLLWIN 010
# define          _STANDOUT     0200
```

The current (y, x) coordinates for the window are *_cury* and *_curx*. New characters added to the screen are included at this point. The maximum values allowed for (*_cury, _curx*) are *_maxy* and *_maxx*. The starting (y, x) coordinates on the terminal for the window (i.e., the window's home) are *_begy* and *_begx*. The *_cury*, *_curx*, *_maxy*, and *_maxx* coordinates are measured relative to (*_begy, _begx*), not a terminal's home.

**Note:** All variables that you don't normally access directly are named with an initial underscore ( _ ) to avoid conflicts with user variables.

The *_clear* variable tells if a clear-screen sequence is to be generated on the next *refresh()* call. It is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, regardless of the dimensions of the window involved. The *_leave* variable is TRUE if the current (y, x) coordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. If scrolling is allowed, *_scroll* is TRUE.

Since *_y* is a pointer to an array of lines that describe the terminal,

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line.

The *flags* variable can have one or more values or'd into it. *_SUBWIN* means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. *_ENDLINE* says that the end of the line for this window is also the end of a screen. *_FULLWIN* says that this window is a screen. *_SCROLLWIN* indicates that the last character of this screen is at the lower right–hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. *_STANDOUT* says that all characters added to the screen are in standout mode.

## 14.5 Examples

This section contains representative (but not comprehensive) examples of usage.

### 14.5.1 Screen Updating

The following examples demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections that are irrelevant to the examples, and are therefore usually not included. The data structure definitions should help you understand what the relevant portions do.

#### 14.5.1.1 Twinkle

This is a moderately simple program that prints pretty patterns on the screen. It switches between patterns of asterisks, putting them on one–by–one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated in the following.

```
# include           <curses.h>
# include           <signal.h>
/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */
# define            NCOLS     80
# define            NLINES    24
# define            MAXPATTERNS         4
struct locs {
            char    y, x;
};
typedef struct locs           LOCS;
LOCS          Layout[NCOLS * NLINES];      /* current board layout */
int           Pattern,                     /* current pattern number */
              Numstars;                    /* number of stars in pattern */
main() {
              char                  *getenv();
              int                   die();
              srand(getpid());             /* initialize random sequence */
              initscr();
              signal(SIGINT, die);
              noecho();
              nonl();
              leaveok(stdscr, TRUE);
              scrollok(stdscr, FALSE);
              for (;;) {
                            makeboard();           /* make the board setup */
                            puton('*');            /* put on '*'s */
              puton(' ');           /* cover up with ' 's */
        }
}
```

```
/*
 * On program exit, move the cursor to the lower left corner by direct addressing, since
 * current location is not guaranteed. We lie and say we used  to be at the upper right
 * corner to guarantee absolute addressing.
 */
die() {
        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}
/*
 * Make the current board setup.  It picks a random pattern and calls ison() to determine
 *  if the character is on that pattern or not.
 */ makeboard() {
        reg int         y, x;
        reg LOCS        *lp;
        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}
/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {
        switch (Pattern) {
          case 0:          /* alternating lines */
                return !(y & 01);
          case 1:          /* box */
                if (x >= LINES && y >= NCOLS)
                        return FALSE;
                if (y < 3 || y >= NLINES - 3)
                                return TRUE;
                        return (x < 3 || x >= NCOLS - 3);
          case 2:              /* holy pattern! */
                        return ((x + y) & 01);
          case 3:              /* bar across center */
                        return (y >= 9 && y <= 15);
          }
          /* NOTREACHED */
}
puton(ch)
reg char            ch; {
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;
        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp; *
                lp = Layout[r];
                Layout[r] = temp;
        }
        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

### 14.5.1.2 Life

This program plays the famous computer pattern game of life (*Scientific American*, May, 1974). The calculational routines create a linked list of structures defining the location of each piece. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it lets them worry about how the last position looked, instead of you. It also demonstrates some of the input routines.

```
# include           <curses.h>
# include           <signal.h>
/*
 * Run a life game. This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */
struct lst_st {                               /* linked list element */
        int              y, x;                /* (y, x) position of piece */
struct lst_st      *next, *last;              /* doubly linked */
};
typedef struct lst_st       LIST;
LIST      *Head;                 /* head of linked list */
main(ac, av)
int             ac;
char            *av[]; {
                int       die();
                evalargs(ac, av);                      /* evaluate arguments */
                initscr();                             /* initialize screen package */
                signal(SIGINT, die);                   /* set to restore tty stats */
                crmode();                              /* set for char-by-char */
                noecho();                              /* input */
                nonl();                                /* for optimization */
                getstart();                            /* get starting position */
                for (;;) {
                        prboard();             /* print out current board */
                        update();              /* update board position */
                }
}
/*
 * This is the routine which is called when rubout is hit. It resets the tty stats to their
 * original values. This is the normal way of leaving the program.
 */
die() {
                signal(SIGINT, SIG_IGN);               /* ignore rubouts */
                mvcur(0, COLS-1, LINES-1, 0);          /* go to bottom of screen */
                endwin();                              /* set terminal to initial state */
                exit(0);
}
/*
 * Get the starting position from the user. They keys u, i, o, j, l, m, ,, and . are used for
 * moving their relative directions from the k key. Thus, u move diagonally up to the left, ,
 * moves directly down, etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file. The list is built after the board setup is ready.
 */ getstart() {
                reg char             c;
                reg int              x, y;
                box(stdscr, '|', '_');                 /* box in the screen */
                move(1, 1);                            /* move to upper left corner */
                do {
                        refresh();                          /* print current position */
        if ((c=getch()) == 'q')
                                break;
                        switch (c) {
                          case 'u':
                          case 'i':
                          case 'o':
                          case 'j':
                          case 'l':
                          case 'm':
                          case ',':
                          case '.':
                                adjustyx(c);
                                break;
                          case 'f':
                                mvaddstr(0, 0, "File name: ");
                getstr(buf);
                                readfile(buf);
                                break;
                          case 'x':
                                addch('X');
                                break;
                          case ' ':
                                addch(' ');
                                break;
                        }
        }
        if (Head != NULL)                                   /* start new list */
        dellist(Head);
        Head = malloc(sizeof (LIST));
        /*
         * loop through the screen looking for 'x's, and add a list element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}
/*
 * Print out the current board position from the linked list
 */ prboard() {
        reg LIST        *hp;
        erase();                                    /* clear out last position */
        box(stdscr, '|', '_');                      /* box in the screen */
        /*
         * go through the list adding each piece to the newly blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');
        refresh();
}
```

### 14.5.2 Motion optimization

The following shows how motion optimization is written on its own. Programs that flit around without regard for existing conditions usually don't need the overhead of space and time associated with screen updating; they should use motion optimization instead.

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {
        reg char                *sp;
        char                    *getenv();
        int                      _putchar(), die();
        srand(getpid());                        /* initialize random sequence */
        if (isatty(0)) {
                gettmode();
                if (sp=getenv("TERM"))
                                setterm(sp);
                        signal(SIGINT, die);
        }
        else {
                        printf("Need a terminal on %d\n", _tty_ch);
                        exit(1);
        }
        _puts(TI);
        _puts(VS);
        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
                        makeboard();            /* make the board setup */
                        puton('*');             /* put on '*'s */
                                puton(' ');     /* cover up with ' 's */
                }
}
/*
 * _putchar defined for tputs() (and _puts())
 */
 _putchar(c)
 reg char               c; {
                putchar(c);
}
puton(ch)
char            ch; {
                static int              lasty, lastx;
                reg LOCS                *lp;
                reg int                 r;
                reg LOCS                *end;
                LOCS                    temp;
                end = &Layout[Numstars];
                for (lp = Layout; lp < end; lp++) {
                        r = rand() % Numstars;
                        temp = *lp;
                        *lp = Layout[r];
                        Layout[r] = temp;
                }
                for (lp = Layout; lp < end; lp++)
                                        /* prevent scrolling */
                        if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                                mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                                        lasty = lp->y;
                                        if ((lastx = lp->x + 1) >= NCOLS)
                                                if (AM) {
                                                        lastx = 0;
                                                        lasty++;
                                        }
                                        else
                                                        lastx = NCOLS - 1;
                }
```

Chapter 15

# BSD4.2 Interprocess Communications (IPC)

## 15.1 Introduction

One of the most important parts of DOMAIN/IX *bsd4.2* is its interprocess communication (IPC) facilities. These facilities are the result of several years of research at the University of California at Berkeley. They incorporate many of the ideas from current research, while trying to maintain a philosophical compatibility with the UNIX system.

Prior to the advent of these IPC facilities, the only standard mechanism that allowed two processes to communicate was the pipe. Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes make them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the UNIX IPC facilities met with mixed reaction. Most of the problems were related to the fact that these facilities were tied to the UNIX file system; either through naming or implementation. Consequently, the IPC facilities provided in 4.2BSD are designed as a totally independent subsystem. The 4.2BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system–like name space (a space where all names are pathnames) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe–like entity. These extensions have resulted in a completely new part of the system with which users need time to familiarize themselves. As these facilities are used more, they will probably be refined.

This chapter is organized into several topics: new system calls and the basic model of communication, supporting library routines useful in constructing distributed applications, the client/server model used in developing applications (including examples of the two major types of servers), and advanced topics that sophisticated users may encounter when using the IPC facilities.

## 15.2 Basics

The basic building block for communication is the socket, an endpoint of communication to which a name may be bound. Each socket in use has a type and one or more associated processes. Sockets exist within communication domains, abstractions introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the Internet communication domain, socket names contain an Internet address and port number. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2BSD IPC supports two separate communication domains: the UNIX domain, and the Internet domain (used by processes that communicate via DARPA standard communication protocols). The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. Currently, the DOMAIN/IX system only supports the Internet domain.

### 15.2.1 Socket Types

Sockets are typed according to communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although nothing prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets are currently available to users. A stream socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bi-directionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes.

A datagram socket supports bi-directional flow of data not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated and, possibly, in an order different from the one sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A raw socket provides users access to the underlying communication protocols that support socket abstractions. These sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol. Raw sockets aren't intended for the general user; they are mainly for those developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. We consider the use of raw sockets later in this chapter.

Two potential socket types having interesting properties are the sequenced packet socket and the reliably delivered message socket. A sequenced packet socket is identical to a stream socket, except that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types are loosely defined, they are currently unimplemented in BSD4.2. Thus, we describe only the three socket types now supported.

## 15.2.2 Socket Creation

To create a socket, use the **socket**(2) system call:

**s = socket(domain, type, protocol);**

This call requests that the system create a socket in the specified *domain* and of the specified *type*. You may also request a particular protocol. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols comprising the communication domain and may be used to support the requested socket type. You are returned a descriptor (a small integer number) that may be used in later system calls that operate on sockets. The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>.

**Note:** The manifest constants are named AF_*name* as they indicate the "address format" to use in interpreting names. Currently, the DOMAIN/IX system only supports the Internet domain constant, AF_INET.

The socket types are also defined in this file, and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain, the following call might be used:

**s = socket(AF_INET, SOCK_STREAM, 0);**

This call results in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on–machine use, a sample call might be:

**s = socket(AF_INET, SOCK_DGRAM, 0);**

To obtain a particular protocol, select the protocol number, as defined within the communication domain. For the Internet domain, the available protocols are defined in <*netinet/in.h*>. Better yet, use one of the library routines discussed in Section 15.3, e.g., **getprotobyname**(3N):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

A socket call may fail for several reasons. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail if the protocol is unknown (EPROTONOSUPPORT), or if the socket has no supporting protocol (EPROTOTYPE).

## 15.2.3 Binding Names

A socket is created without a name. Until a name is bound to a socket, processes can't reference it. Consequently, no messages may be received on it. The **bind**(2) call assigns a name to a socket:

**bind(s, name, namelen);**

The bound name is a variable length byte string that is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties that comprise the domain). In the Internet domain, names contain an Internet address and port number.

In binding an Internet address, the actual call is rather simple:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, &sin, sizeof (sin));
```

However, the selection of what to place in the address *sin* requires some discussion. We return to the problem of formulating Internet addresses when we discuss the library routines used in name resolution.

### 15.2.4 Connection Establishment

With a bound socket, it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process being a client and the other a server. The client requests services from the server by initiating a connection to the server's socket. The server, when willing to offer its advertised services, passively listens on its socket. On the client side, the **connect**(2) call initiates a connection. Using the Internet domain, this might appear as follows:

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket (see Section 15.5.4). An error is returned when the connection is unsuccessful (although any name automatically bound by the system remains). Otherwise, the socket is associated with the server, and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

| | |
|---|---|
| ETIMEDOUT | After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost. |
| ECONNREFUSED | The host refused service for some reason. When connecting to a host running 4.2BSD, this error is usually due to a server process not being present at the requested name. |
| ENETDOWN (EHOSTDOWN) | These operational errors are returned based on status information delivered to the client host by the underlying communication services. |
| ENETUNREACH (EHOSTUNREACH) | These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Often, the status returned isn't sufficient to distinguish a network being down from a host being |

down. In these cases, the system is conservative and indicates that the entire network is unreachable.

For the server to receive a client's connection, it must perform two steps after binding its socket. First, it must indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the **listen**(2) call specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection isn't refused; the individual messages comprising the request are ignored. This gives a busy server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. It is still possible (though unlikely) to get the ETIMEDOUT error back. The system limits the backlog figure supplied with **listen** to a maximum of five pending connections on any one queue. This avoids the problem of processes monopolizing system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may **accept**(2) a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wants to find out who its client is, it may supply a buffer for the client socket's name. The value–result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name isn't of interest, the second parameter may be zero.

The **accept** fuction normally blocks. That is, the call to accept doesn't return until a connection is available or the system call is interrupted by a signal to the process. Further, a process cannot indicate that it accepts connections only from specific individual(s). The user process must consider who the connection is from and close down the connection if it doesn't want to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the **accept** call, there are alternatives (see Section 15.5 for further information).

## 15.2.5 Data Transfer

With a connection established, data may begin to flow. A number of calls can be used to send and receive data. With the peer entity at each end of a connection anchored, you can send or receive a message without specifying the peer. As you might expect in this case, the normal **read**(2) and **write**(2) system calls are useable:

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to the above calls, the new calls **send**(2) and **recv**(2) may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While **send** and **recv** are virtually identical to **read** and **write**, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

| | |
|---|---|
| MSG_OOB | Send/receive out-of-band data. |
| MSG_PEEK | Look at data without reading. |
| MSG_DONTROUTE | Send data without routing packets. |

Out-of-band data is a notion specific to stream sockets, and one that we don't immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PREVIEW is specified with a **recv** call, any data present is returned to the user, but treated as still unread. That is, the next **read** or **recv** call applied to the socket returns the data previously previewed.

## 15.2.6 Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a **close**(2) to the descriptor:

```
close(s);
```

If data is associated with a socket that promises reliable delivery (e.g., a stream socket) when a **close** takes place, the system continues to try transferring the data. However, if the data is still undelivered after a fairly long period of time, it is discarded. Should you have no use for any pending data, the system may perform a **shutdown**(2) on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if you are no longer interested in reading data, 1 if no more data is to be sent, or 2 if no data is to be sent or received. Applying **shutdown** to a socket causes any data queued to be immediately discarded.

## 15.2.7 Connectionless Sockets

Thus far, we have been concerned mostly with sockets that follow a connection-oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to enable the recipient of a message to identify the sender. To send data, use the **sendto** primitive as follows:

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values indicate the intended recipient of the message. When using an unreliable datagram interface, it's unlikely that errors will be reported to the sender. Where information is present locally to recognize a message that may never be delivered (e.g., when a network is unreachable), the call returns -1 and the global value *errno* contains an error number.

To receive messages on an unconnected datagram socket, use the **recvfrom** primitive:

**recvfrom(s, buf, buflen, flags, &from, &fromlen);**

Once again, the *fromlen* parameter is handled in a value–result fashion, initially containing the size of the *from* buffer.

Besides the two calls mentioned above, datagram sockets may also use the **connect** call to associate a socket with a specific address. In this case, any data sent on the socket is automatically addressed to the connected peer, and only data received from that peer is delivered to the user. Only one connected address is permitted for each socket (i.e., no multi–casting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other details of datagram sockets are described in Section 15.5.

### 15.2.8 Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex I/O requests among multiple sockets and/or files. This is done using the **select**(2) call:

**select(nfds, &readfds, &writefds, &execptfds, &timeout);**

**Select** takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or–ing bits of the form

```
1 << fd
```

That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The *nfds* parameter specifies the range of file descriptors (i.e., one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection blocks indefinitely.

**Note:** To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

**Select** normally returns the number of file descriptors selected. If the **select** call returns due to the timeout expiring, then a value of –1 is returned along with the error number EINTR.

**Select** provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in Section 15.5.

## 15.3 Network Library Routines

We have indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task, a number of routines have been added to the standard C run–time library. In this section, we consider the new routines provided to manipulate network addresses.

While the 4.2BSD networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we recommend that the same user interface be maintained in accessing network–related address databases. The only difference should be the values returned to the user. Since these values are normally supplied to the system, users needn't be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name intended for human consumption (e.g., the *login server* on host *monet*). This name, and the name of the peer host, must then be translated into network addresses not necessarily suitable for human consumption. Finally, the address must then used in locating a physical location and route to the service.

The specifics of these three mappings are likely to vary between network architectures. For instance, it is undesirable for a network to require that hosts be named so that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wants to communicate. This ability to have hosts named in a location–independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

## 15.3.1 Host Names

A host name to address mapping is represented by the *hostent* structure:

```
struct hostent {
        char    h_name;       /* official name of host */
        char    **h_aliases;  /* alias list */
        int     h_addrtype;   /* host address type */
        int     h_length;     /* length of address */
        char    *h_addr;      /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine **gethostbyname**(3N) takes a host name and returns a *hostent* structure, while the routine **gethostbyaddr**(3N) maps host addresses into a *hostent* structure. A host may have many addresses, all having the same name. **Gethostybyname** returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine **gethostent**(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network, this routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                                continue;
                if (strcmp(name, hp->h_name)) {
                                for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                        if (strcmp(name, *cp) == 0)
                                        goto found;
                                continue;
                }
        found:
                 if (in_netof(*(struct in_addr *)hp->h_addr)) == net)

                                break;
        }
        endhostent(0);         return (hp); }
```

**Note:** The standard routine **in_netof**(3N) returns the network portion of an Internet address.

## 15.3.2 Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct netent {
        char    *n_name;      /* official name of net */
        char    **n_aliases;  /* alias list */
        int     n_addrtype;   /* net address type */
        int     n_net;        /* network # */
};
```

The routines **getnetbyname**(3N), **getnetbynumber**(3N), and **getnetent**(3N) are the network counterparts to the host routines described above.

## 15.3.3 Protocol Names

For protocols, the *protoent* structure defines the protocol–name mapping used with the routines **getprotobyname**(3N), **getprotobynumber**(3N), and **getprotoent**(3N):

```
struct  protoent {
        char    *p_name;     /* official protocol name */
        char    **p_aliases; /* alias list */
        int     p_proto;     /* protocol # */
};
```

## 15.3.4. Service Names

Information regarding services is more complicated. A service is expected to reside at a specific port and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Also, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines must be bypassed in favor of homegrown routines like the *gethostbynameandnet* routine described above. A service mapping is described by the *servent* structure:

```
struct  servent {
        char    *s_name;     /* official service name */
        char    **s_aliases; /* alias list */
        int     s_port;      /* port # */
        char    *s_proto;    /* protocol to use */
};
```

The routine **getservbyname**(3N) maps service names to a servent structure by specifying a service name and (optionally) a qualifying protocol. Thus, the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol; the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server that uses the TCP protocol. Also provided are the routines **getservbyport**(3N) and **getservent**(3N). The **getservbyport** routine has an interface similar to that provided by **getservbyname**; an optional protocol name may be specified to qualify lookups.

## 15.3.5 Miscellaneous

With the support routines we've described, an application program should rarely have to deal directly with addresses. Thus, services can be developed in a largely network–independent fashion. However, purging all network dependencies is difficult. So long as you must give network addresses when naming services and sockets, there will always be some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown here:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp- >h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

**Note:** This example is considered in more detail in Section 15.4.

To make the remote login program independent of the Internet protocols and addressing scheme, we would be forced to add a layer of routines that masked the network dependent aspects from the mainstream login code. For the current facilities available in the system, the effort doesn't seem worthwhile. Perhaps when the system is adapted to different network architectures, the utilities will be reorganized more cleanly.

Aside from the address-related database routines, several other interesting routines are available. Most of these simplify manipulation of names and addresses. The following summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values:

| | |
|---|---|
| **bcmp**(*s1, s2, n*) | Compare byte-strings; 0 if same, not 0 otherwise. |
| **bcopy**(*s1, s2, n*) | Copy *n* bytes from *s1* to *s2*. |
| **bzero**(*base, n*) | Zero-fill *n* bytes starting at base. |
| **htonl**(*val*) | Convert 32-bit quantity from host to network byte order. |
| **htons**(*val*) | Convert 16-bit quantity from host to network byte order. |
| **ntohl**(*val*) | Convert 32-bit quantity from network to host byte order. |
| **ntohs**(*val*) | Convert 16-bit quantity from network to host byte order. |

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines that return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies that you should encounter the byte swapping problem only when interpreting network addresses. Thus, you need this code to print out an Internet port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX, these routines are defined as null macros.

## 15.4 Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server examined in Section 15.2. In this section, we look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well-known set of conventions before service may be rendered and accepted. This set of conventions comprises a protocol that must be implemented at both ends of a connection. The protocol may be symmetric or asymmetric.

In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, the other the slave. The TELNET protocol used in the Internet for remote terminal emulation is an example of a symmetric protocol. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. Whether the specific protocol used to get service is symmetric or asymmetric, a client process and a server process must exist for service access.

A server process normally listens at a well-known address for service requests. Alternative schemes that use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it needs. Then, the Courier server process creates the appropriate server process (based on a database) and splices the client and server together, voiding its part in the transaction.

The Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, in spite of how attractive it may be for standardizing access to services, this scheme introduces some overhead due to the intermediate process involved. Implementations that provide this type of service within the system can minimize the cost of client server rendezvous. The portal notion described on the **telnetd**(8) and **ftpd**(8) manual pages embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

## 15.4.1 Servers

In the DOMAIN/IX system, most servers are accessed at known Internet addresses. When a server is started at boot time, it advertises it services by listening at a well-known location. For example, the remote login server's main loop is of this form:

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                                if (errno != EINTR)
                                        perror("rlogind: accept");
                                continue;
                }
                if (fork() == 0) {
                                close(f);
                                doit(g, &from);
                }
                close(g);
        }
}
```

The first step taken by the server is to look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1); }
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important, as the server probably doesn't want to receive signals delivered to the process group of the controlling terminal.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The **bind**(2) call is required to ensure that the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);
g = accept(f, &from, &len);
if (g < 0) {
        if (errno != EINTR)
                perror("rlogind: accept");
        continue;
}
if (fork() == 0) {
        close(f);
        doit(g, &from);
}
close(g); }
```

An **accept**(2) call blocks the server until a client requests service, and could return a failure status if interrupted by a signal such as SIGCHLD (see Section 15.5). Therefore, the return value from **accept** is checked to ensure that a connection has actually been established.

With a connection in hand, the server forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the **accept** is closed in the parent. The address of the client is also handed the *doit* routine for authenticating clients.

## 15.4.2 Clients

We showed the client side of the remote login service earlier in this chapter. Separate, asymmetric roles of the client and server are apparent in the code. The server is a passive entity, listening for client connections; the client process is an active entity, initiating a connection when invoked.

Let's consider more closely the steps taken by the client remote login process. As in the server process, the first step is locating the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next, the destination host is looked up with a **gethostbyname** call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

All that remains now is establishing a connection to the server at the requested host and starting up the remote login protocol. The address buffer is cleared, then filled with the Internet address of the foreign host and number of the port where the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp- >h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated:

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol are not considered here.

## 15.4.3 Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One in particular is the **rwho**(1C) service that provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to broadcast information to all hosts connected to a particular network, is of interest as a sample usage of datagram sockets.

A user on any machine running the **rwho** server may find out the current status of a machine with the **ruptime**(1C) program. The output generated is illustrated here:

```
arpa      up     9:45,      5 users, load     1.15,    1.39,    1.31
cad       up     2+12:04,   8 users, load     4.67,    5.13,    4.59
calder    up     10:10,     0 users, load     0.27,    0.15,    0.14
dali      up     2+06:28,   9 users, load     1.04,    1.20,    1.65
degas     up     25+09:48,  0 users, load     1.49,    1.43,    1.41
ear       up     5+00:05,   0 users, load     1.51,    1.54,    1.56
ernie     down   0:24
esvax     down   17:04
ingres    down   0:26
kim       up     3+09:16,   8 users, load     2.03,    2.46,    3.11
matisse   up     3+06:18,   0 users, load     0.03,    0.03,    0.05
medea     up     3+09:39,   2 users, load     0.35,    0.37,    0.50
merlin    down   19+15:37
miro      up     1+07:20,   7 users, load     4.59,    3.28,    2.12
monet     up     1+00:43,   2 users, load     0.22,    0.09,    0.07
oz        down   16:09
statvax   up     2+15:57,   3 users, load     1.52,    1.81,    1.86
ucbvax    up     9:34,      2 users, load     6.08,    5.16,    3.28
```

Status information for each host is periodically broadcast by **rwho** server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The server performs two separate tasks. First, it receives status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the **rwho** port are interrogated to ensure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host.

When a host has not been heard from for an extended period of time, the database interpretation routines assume that the host is down and indicate such on the status reports. Although somewhat prone to error (e.g., a server may be down while a host is actually up), this algorithm serves our current needs.

The second task performed by the server is supplying information about the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other **rwho** servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but not very interesting. Having to decide where to transmit the resultant packet does, however, indicate some problems with the current protocol.

The **rwho** server, in a simplified form, is pictured as follows:

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                                if (cc < 0 && errno != EINTR)
                                        perror("rwhod: recv");
                                continue;
                }
                if (from.sin_port != sp->s_port) {
                                fprintf(stderr, "rwhod: %d: bad from port\n",
                                        ntohs(from.sin_port));
                                continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                                fprintf(stderr, "rwhod: malformed host name from %x\n",
                                        ntohl(from.sin_addr.s_addr));
                                continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY|O_CREATE|O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Status information is broadcast on the local network. Networks that don't support the idea of broadcast must use another scheme to simulate or replace broadcasting. While it's possible to list known neighbors (based on the status received), some bootstrapping information is needed, as a server started up on a quiet network has no known neighbors and thus never receives or sends any status information. The routing table management process also faces this problem in propagating routing status information.

The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that aren't (possibly) direct neighbors. If the server can support networks that provide a broadcast capability, as well as those that don't, then networks with an arbitrary topology may share status information.

Note: If a host is connected to multiple networks, it receives status information from itself. This can lead to an endless, wasteful exchange of information.

The second problem with the current scheme is that the **rwho** process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance difficult. 4.2BSD attempts to isolate host-specific information from applications by providing system calls that return the necessary information. (An example of such a system call is the **gethostname**(2) call, which returns the host's "official" name.)

Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the **rwho** server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is interrogatation of system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

## 15.5 Advanced Topics

A number of facilities have yet to be discussed. For most IPC users, the mechanisms already described suffice in constructing distributed applications. However, others must use some of the features that we consider in this section.

### 15.5.1 Out-of-Band Data

The stream socket abstraction includes the notion of out-of-band data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independent of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to you at any one time. For communications protocols that support only in-band signaling (i.e., urgent data is delivered in sequence with normal data) the system extracts the data from the normal data stream and stores it separately. This lets you choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out-of-band message, supply the MSG_OOB flag to **send** or **sendto** calls. To receive out-of-band data, indicate MSG_OOB when performing a **recvfrom** or **recv** call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK **ioctl** is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read returns data after the mark. Otherwise (assuming out-of-band data has arrived), the next read provides data sent by the client prior to transmission of the out-of-band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown here:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                                perror("ioctl");
                                break;
                }
                if (mark)
                                break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
         ...
}
```

## 15.5.2 Signals and Process Groups

Because the SIGURG and SIGIO signals exist, each socket has an associated process group (as is true for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP **ioctl**:

**ioctl(s, SIOCSPGRP, &pgrp);**

A similar **ioctl**, SIOCGPGRP, determines the current process group of a socket.

## 15.5.3 Pseudo Terminals

Many programs don't function properly without a terminal for standard input and output. Since a socket is not a terminal, a pseudo terminal is often needed for processes to be able to communicate over the network. A pseudo terminal is actually a pair of devices, master and slave, that let a process serve as an active agent in communication between processes and users.

Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given to the slave as input. Thus, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side.

The remote login server uses pseudo terminals for remote login sessions. If you log in to a machine across the network, you are provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process.

When you send an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out-of-band message to the server process who then uses the signal number, sent as the data value in the out-of-band message, to perform a **killpg**(2) on the appropriate process group.

### 15.5.4 Internet Address Binding

Binding addresses to sockets in the Internet domain can be rather complex. Communicating processes are bound by an association composed of local and foreign addresses and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique; thus, no duplicate *<protocol, local address, local port, foreign address, foreign port>* tuples may exist.

The **bind** system call lets a process specify half of an association, <local address, local port>, while the **connect** and **accept** primitives complete a socket's association. (The association is created in two steps, so be careful not to violate its need to be unique.) It is unrealistic to expect user programs to always know proper values for the local address and local port, since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding, the notion of a wildcard address is available. If you specify an address as INADDR_ANY (a manifest constant defined in *<netinet/in.h>*), the system interprets the address as any valid address. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, suppose a host is on networks 46 and 10 and a socket is bound as we just showed in our code. If an **accept** call is performed, the process can accept connection requests that arrive either from network 46 or network 10.

In a similar fashion, if a local port is left unspecified (specified as zero), the system selects an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. First, ports numbered 0 through 1023 are reserved for privileged users (i.e., super user). Secondly, the port number should not be currently bound to some other socket.

To find a free port number in the privileged range, the remote shell server uses the following code:

```
struct sockaddr_in sin;
...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
...
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("bind");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports lets processes executing in a secure environment perform authentication based on the originating address and port number.

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application. This is due to associations being created in a two–step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports.

In this situation, the system disallows binding the same local address and port number to a socket if a previous data connection's socket is present. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses already in use may be bound. This doesn't violate the uniqueness requirement, as the system still ensures at connect time that any other sockets with the same local address and port do not have the same foreign address and port. If an association already exists, the error EADDRINUSE is returned.

Local address binding may be somewhat haphazard when a host is on multiple networks. Logically, you may expect the system to bind the local address associated with the network through which a peer is communicating.

For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 is arriving via network 10, the local address to be bound is the host's address on network 10, not network 46. This isn't always the case. For reasons too complicated to discuss here, the local address bound may appear to be chosen at random.

This property of local address binding is normally invisible to users unless the foreign host doesn't understand how to reach the address selected. (For example, if network 46 isn't known to network 32, and the local address is bound to that located on network 46, then even though a route between the two hosts exists through network 10, a connection ultimately fails.)

### 15.5.5 Broadcasting and Datagram Sockets

By using a datagram socket, broadcast packets can be sent on many networks supported by the system. However, the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super-user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a **sendto** call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the sender's address and port (datagram sockets are anchored before a message is allowed to go out).

### 15.5.6 Signals

Two additional new signals, SIGURG and SIGIO, may be used in conjunction with the interprocess communication facilities.

The SIGURG signal is associated with the existence of an urgent condition. It is currently supplied a process when out-of-band data is present at a socket. If multiple sockets have out-of-band data awaiting delivery, a **select** call may be used to determine those sockets with such data. The SIGIO signal is used with interrupt driven I/O (not presently implemented).

SIGCHLD is an old signal that helps in the construction of server processes. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to reap child processes after exiting.

The remote login server loop shown in a previous example may be augmented. The following code shows how this may be done. Be aware, however, that if the parent server process fails to reap its children, a large number of zombie processes may be created.

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                                perror("rlogind: accept");
                continue;
                }
                ...
        }
        ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

# Index

Primary page references are listed first. The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index.

## Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *DOMAIN/IX Support Tools Guide*
Order No.: 009413 Revision: 00 Date of Publication: November, 1986

What type of user are you?

_____ System programmer; language ____________________
_____ Applications programmer; language ____________________
_____ System maintenance person _____ Manager/Professional
_____ System Administrator _____ Technical Professional
_____ Student Programmer _____ Novice
_____ Other

How often do you use the DOMAIN system?____________________

What parts of the manual are especially useful for the job you are doing?

____________________

What additional information would you like the manual to include?

____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)

____________________

____________________
Your Name Date

____________________
Organization

____________________
Street Address

____________________
City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

**APOLLO COMPUTER INC.**
**Technical Publications**
**P.O. Box 451**
**Chelmsford, MA 01824**

FOLD