HEWLETT PACKARD

HP 9000 Computers

C Programming Tools

C Programming Tools

HP 9000 Computers

HEWLETT PACKARD

HP Part No. B1864-90009
Printed in USA 01/91

First Edition
E0191

Legal Notices

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

January 1991 ... Edition 1.

Contents

Figures

Tables

1

Introduction

This manual provides you with a tutorial on a few of the C language programming tools that are shipped with your C language product.

The following table provides you with a list and description of the C tools that are covered in this manual. It also provides a reference to chapters in this manual that contain information about these tools.

Table 1-1. Table of C Programming Tools Covered in This Manual

C Tool	Description	For information, read ...
`lex`	A program generator for lexical analysis of text	Chapter 3
`lint`	A C program checker	Chapter 2
`yacc`	A programming tool for describing the input to a computer program	Chapter 4

The following table provides you with a list and description of the C tools that *are not* covered in this manual. It also provides a reference for finding information about these tools.

Table 1-2.
Table of C Programming Tools Not Covered in This Manual

C Tool	Description	For information, read ...
`cb`	A C program beautifier	*HP-UX Reference Vol. 1: Section 1*
`cflow`	A C flow graph generator	*HP-UX Reference Vol. 1: Section 1*
`cpp`	The C language preprocessor	*HP-UX Reference Vol. 1: Section 1*
`ctags`	A C programming tool that creates a tag file for *ex(1)* or *vi(1)* from the specified C, Pascal and FORTRAN sources	*HP-UX Reference Vol. 1: Section 1*
`cxref`	A C program cross-reference generator	*HP-UX Reference Vol. 1: Section 1*

Prerequisites to Reading This Manual

Before reading this manual, you should have a good knowledge of:

- The HP-UX operating system
- The C programming language
- An HP-UX text editor such as vi.

Manual Overview

The following list contains a description of the contents of this manual's chapters and appendix.

Chapter 1	provides a list of C tools, prerequisites to reading this manual, a manual overview, and manual conventions.
Chapter 2	provides a tutorial for `lint` the C program checker. It covers error and problem detection, how to use `lint`, and `lint` directives.
Chapter 3	provides a tutorial for `lex` which is a program generator for lexical processing of character input streams. It covers `lex` source, regular expressions, actions, operator precedence, source definitions, usage, and several `lex` features and examples.
Chapter 4	provides a tutorial for `yacc` which is a general programming tool for describing the input to a computer program. It covers basic specifications, actions, how the parser works, precedence, error handling, advanced `yacc` topics, `yacc` examples, input syntax, and support information.

Manual Conventions

This manual uses the following typographical conventions:

Table 1-3. Typographical Conventions Used in This Manual

If you see ...	It means ...
`computer text`	Indicates text displayed by the computer system. For example, `warning: loop not entered from top` is a warning message given by `lint` when the loop cannot be entered from the top.
italic text	You supply the text. For example, `login:` *login name* is a login prompt displayed by the computer. You would respond by typing in your *login name*. Note that *italic text* is also used for emphasis.

2

lint: A C Program Checker

Introduction

The lint command is a program checker and verifier for C source code. Its main purpose is to supply the programmer with warning messages about problems with the source code's style, efficiency, portability, and consistency. The lint command can be used before compiling a program to check for syntax errors and after compiling a program to test for subtle errors (e.g., type differences, etc.).

Error messages and lint warnings are sent to standard error (stderr). Once the code errors are corrected, the C source file(s) should be run through the C compiler to produce the necessary object code.

How to Use `lint`

The `lint` command has the form:

`lint` [*options*]*files* ... *library-descriptors* ...

where *options* are optional flags to control `lint` checking and messages, *files* are the files to be checked that end with `.c` or `.ln`, and *library descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the `lint` command are:

`-a`	Suppress messages about assignments of long values to variables that are not long.
`-b`	Suppress messages about break statements that cannot be reached.
`-c`	Only check for intrafile bugs; leave external information in files suffixed with `.ln`.
`-h`	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
`-n`	Do not check for compatibility with either the standard or the portable `lint` library.
`-o` *name*	Create a `lint` library from input files named `llib-l`*name*`.ln`.
`-p`	Attempt to check portability to other dialects of C language.
`-s`	Check for cases where the alignment of structures, unions, and pointers may not be portable.
`-u`	Suppress messages about function and external variables used and not defined or defined and not used.
`-v`	Suppress messages about unused arguments and functions.
`-x`	Do not report variables referred to by external declarations but never used.
`-Aa`	Invoke `lint` in ANSI mode.
`-Ac`	Invoke `lint` in compatibility mode. The default is compatibility mode.

The names of files that contain C language programs should end with the suffix .c, which is mandatory for lint and the C compiler.

The lint command accepts certain arguments, such as:

```
-lm
```

The lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined on a library file but are not used on a source file do not result in messages. The lint command does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, lint checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the -p option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The -n option can be used to suppress all library checking.

Directives

The alternative to using options to suppress `lint`'s comments about problem areas is to use directives. Directives appear in the source code in the form of code comments. The `lint` command recognizes five directives.

`/*NOTREACHED*/`	stops an unreachable code comment about the next line of code.
`/*NOSTRICT*/`	stops `lint` from strictly type checking the next expression.
`/*ARGSUSED*/`	stops a comment about any unused parameters for the following function.
/*VARARGS*n**/	stops `lint` from reporting variable numbers of parameters in calls to a function. The function's definition follows this comment. The first *n* parameters must be present in each call to the function; `lint` comments if they aren't. If "/*VARARGS*/" appears without the *n*, none of the parameters need be present. This comment must precede the actual code for a function. It *should not* precede `extern` declarations.
`/*LINTLIBRARY*/`	must be placed at the beginning of a source file. This directive tells `lint` that the source file is used to create a `lint` library file and to suppress comments about the unused functions. `lint` objects if other files redefine routines that are found there.

Problem Detection

Remember that a compiler reports errors only when it encounters program source code that cannot be converted into object code. The main purpose of `lint` is to find problem areas in C source code that it considers to be inefficient, nonportable, bad style, or a possible bug, but which the C compiler accepts as error-free because it can be converted into object code.

Comments about problems that are local to a function are produced as each problem is detected. They have the form:

(*line #*) `warning`: *message text*

Information about external functions and variables is collected and analyzed after `lint` has processed the source files. At that time, if a problem has been detected, it sends a warning message with the form:

message text

followed by a list of external names causing the message and the file where the problem occurred.

Code causing `lint` to issue a warning message should be analyzed to determine the source of the problem. Sometimes the programmer has a valid reason for writing the problem code. Usually, though, this is not the case. The `lint` command can be very helpful in uncovering subtle programming errors.

The `lint` command checks the source code for certain conditions, about which it issues warning messages. These can be grouped into the following categories:

- variable or function is declared but not used
- variable is used before it is set
- portion of code is unreachable
- function values are used incorrectly
- type matching does not adhere strictly to C rules
- code has portability problems
- code construction is strange.

The code that you write may have constructions in it that `lint` objects to but that are necessary to its application. Warning messages about problem areas that you know about and do not plan to correct can be suppressed. There are two methods for suppressing warning messages from `lint`. The use of `lint` options is one. The `lint` command can be called with any combination of its defined option set. Each option causes `lint` to ignore a different problem area. The other method is to insert `lint` directives into the source code. The `lint` directives are discussed in the section "Directives."

Unused Variables and Functions

The `lint` command objects if source code declares a variable that is never used or defines a function that is never called. Unused variables and functions are considered bad style because their declarations clutter the code.

Unused static identifiers cause the message:

```
(1) static identifier 'name' defined but never used
```

Unused automatic variables cause the message:

```
(1) warning: 'name' unused in function 'name'
```

A function or external variable that is unused causes the message:

```
name defined but never used
```

followed by the function or variable name, the line number and file in which it was defined. The `lint` command also looks at the special case where one of the parameters of a function is not used. The warning message is:

```
warning: (line number) 'arg_name' in 'func_name'
```

If functions or external variables are declared but never used or defined, `lint` responds with

```
name declared but never used or defined
```

followed by a list of variable and function names and the names of files where they were declared.

Suppressing Unused Functions and Variables Reports

Sometimes it is necessary to have unused function parameters to support consistent interfaces between functions. The `-v` option can be used with `lint` to have warnings about unused parameters suppressed.

If `lint` is run on a file that is linked with other files at compile time, many external variables and functions can be defined but not used, as well as used but not defined. If there is no guarantee that the definition of an external object is always seen before the object code is used, it is declared `extern`. The `-u` option can be used to stop complaints about all external objects, whether or not they are declared `extern`. If you want to inhibit complaints about only the `extern` declared functions and variables, use the `-x` option.

Table 2-1. Options for Suppressing Unused Function and Variable Reports

Option	Description
`-v`	suppress warnings about unused parameters
`-u`	stops complaints about all external objects, whether or not they are declared `extern`
`-x`	inhibits complaints about only the `extern` declared functions and variables

Set/Used Information

A problem exists in a program if a variable's value is used before it is assigned. Although `lint` attempts to detect occurrences of this, it takes into account only the physical location of the code. If code using a local variable is located before the variable is given a value, the message is:

```
warning: 'name' may be used before set
```

The `lint` command also objects if automatic variables are set in a function but not used. The message given is:

```
warning: 'name' set but not used in function 'func_name'
```

Note that `lint` *does not* have an option for suppressing the display of warnings for variables that are used but not set or set but not used.

Unreachable Code

The `lint` command checks for three types of unreachable code. Any statement following a `goto`, `break`, `continue`, or `return` statement must either be labeled or reside in an outer block for `lint` to consider it reachable. If neither is the case, `lint` responds with:

```
warning: (line number) statement not reached
```

The same message is given if `lint` finds an infinite loop. It only checks for the infinite loop cases of `while(1)` and `for(;;)`. The third item that `lint` looks for is a loop that cannot be entered from the top. If one is found, then the message sent is:

```
warning: loop not entered from top
```

The `lint` command's detection of unreachable code is by no means exhaustive. Warning messages can be issued about valid code, and conversely `lint` may overlook code that cannot be reached.

Suppressing Unreadable Code Reports

Programs that are generated by yacc or lex can have many unreachable break statements. Normally, each one causes a complaint from lint. The -b option can be used to force lint to ignore unreachable break statements.

Function Value

The C compiler allows a function containing both the statement

```
return();
```

and the statement

```
return(expression);
```

to pass through without complaint. The lint command, however, detects this inconsistency and responds with the message:

```
warning: function 'name' has 'return(expression)' and 'return'
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

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

Notice that if a tests false, f will call g and then return with no defined value. This will trigger a message for lint. If g (like exit) never returns, the message will still be produced when in fact nothing is wrong. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, lint detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes used, it may represent bad style (e.g., not testing for error conditions).

The lint command will not issue a diagnostic message if that function call is cast as void. For example,

```
    (void) printf("%d\n",i);
```

tells lint to not warn about the ignored return value.

The dual problem – using a function value when the function does not return one – is also detected. This is a serious problem.

The lint command *does not* have an option for suppressing the display of warnings for inconsistent return functions and functions that return no value.

Portability

The -p option of lint aids the programmer in writing portable code in four areas:

- character comparisons;
- pointer alignments (this is default on PA-RISC computers);
- length of external variables;
- type casting.

Character representation varies on different machines. Characters may be implemented as signed values. As a result, certain comparisons with characters give different results on different machines. The expression

```
c<0
```

where c is defined as type char, is always false if characters are unsigned values. If, however, characters are signed values, the expression could be either true or false. Where character comparisons could result in different values depending on the machine used, lint outputs the message:

```
warning: nonportable character comparison
```

Legal pointer assignments are determined by the alignment restrictions of the particular machine used. For example, one machine may allow double-precision values to begin on any modulo-4 boundary, but another may restrict them to modulo-8 boundaries. If alignment requirements are different, code containing an assignment of a double pointer to an integer pointer could cause problems. The lint command attempts to detect where the effect of pointer assignments is machine dependent. The warning that it sends is:

```
warning: possible pointer alignment problem
```

The amount of information about external symbols that is loaded depends on: the machine being used, the number of significant characters, and whether or not uppercase/lowercase distinction is kept. The `lint -p` command truncates all external symbols to six characters and allows only one case distinction. (It changes uppercase characters to lowercase.) This provides a worst-case analysis so that the uniqueness of an external symbol is not machine-dependent.

The effectiveness of type casting in C programs can depend on the machine that is used. For this reason, `lint` ignores type casting code. All assignments that use it are subject to `lint`'s type checking.

Alignment Portability

The -s option of the lint command checks for the following portability considerations:

- pointer alignments (same as -p option)
- a structure's member alignments
- trailing padding of structures and unions

The checks made for pointer alignments are exactly the same as for the -p option. The warning for these cases is:

```
warning: possible pointer alignment problem
```

The alignment of structure members is different between architectures. For example, MC680x0 computers pad structures internally so that all fields of type int begin on an even boundary. In contrast, PA-RISC computers pad structures so that all fields of type int begin on a four-byte boundary. The following structure will be aligned differently on the two architectures:

```
struct s
   { char c;
     long l; /* The offset equals 2 on MC680x0 computers */
   };        /* and 4 on PA-RISC computers.             */
```

In many cases the different alignment of structures does not affect the behavior of a program. However, problems can happen when raw structures are written to a file on one architecture and read back in on another. The lint command checks for cases where a structure member is aligned on a boundary that is not a multiple of its size (for example, int on int boundary, short on short boundary, and double on double boundary). The warning that it sends is:

```
warning: alignment of struct 'name' may not be portable
```

The lint command also checks for cases where the internal padding added at the end of a structure may differ between architectures. The amount of trailing padding can change the size of a structure. The warning that lint sends is:

```
warning: trailing padding of struct/union 's' may not be portable
```

Strange Constructions

A *strange construction* is code that `lint` considers to be bad style or a possible bug.

The `lint` command looks for code that has no effect. For example:

```
*p++;
```

where the * has no effect. The statement is equivalent to "`p++;`". In cases like this, the message:

```
warning: null effect
```

is sent.

The treatment of unsigned numbers as signed numbers in comparison causes `lint` to report:

```
warning: degenerate unsigned comparison
```

The following code would produce such a message:

```
unsigned x;
    .
    .
    .
if (x >= 0) ...
```

The `lint` command also objects if constants are treated as variables. If the boolean expression in a conditional has a set value due to constants, such as

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

`lint`'s response is:

```
warning: constant in conditional context
```

To avoid operator precedence confusion, `lint` encourages using parentheses in expressions by sending the message:

```
warning: precedence confusion possible: parenthesize!
```

The `lint` command judges it bad style to redefine an outer block variable in an inner block. Variables with different functions should normally have different names. If variables are redefined, the message sent is:

warning: *name* redefinition hides earlier one

Suppressing Strange Construction Reports

The `-h` option suppresses `lint` diagnostics of strange constructions.

Standards Compliance

The lint libraries are arranged for standards checking. For example,

```
lint -D_POSIX_SOURCE file.c
```

checks for routines referenced in file.c but not specified in the POSIX standard.

The lint command also accepts ANSI standard C (-Aa) as well as compatible C (-Ac). In ANSI mode, lint invokes the ANSI preprocessor (/lib/cpp.ansi) instead of the compatibility preprocessor (/lib/cpp). ANSI mode lint should be used on source that is compiled with the ANSI standard C compiler.

lex: A Lexical Analyzer and Generator

Introduction

The lex command 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 which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to lex. The lex generated code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program generated by lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

The `lex` command is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." At present, the only host language is C. Just as general purpose languages can produce code to run on different computer hardware, `lex` can generate code in different host languages. The host language is used for the output code generated by `lex` and also for 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, the user's background, and the properties of local implementations. The `lex` command itself exists on HP-UX, but the code generated by `lex` may be taken anywhere the appropriate compilers exist.

The `lex` command turns the user's expressions and actions (called *source*) into the host general-purpose language. The generated program is named `yylex`, and recognizes expressions in a stream (called *input*). The `yylex` command performs the specified actions for each expression as it is detected.

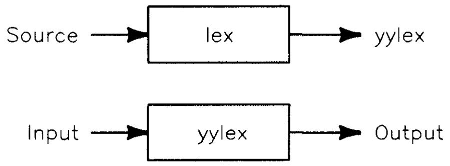

Figure 3-1. An Overview of `lex`

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%                          A space is required before \t.
[ \t]+$         ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules and one rule. This rule contains a regular expression which 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," similar to `vi`. No action is specified, so the program generated by `lex` (`yylex`) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

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

The finite automaton generated for this source will scan 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.

The lex command can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. The lex command can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface lex and yacc. The 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 Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by lex. The yacc command users will 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.

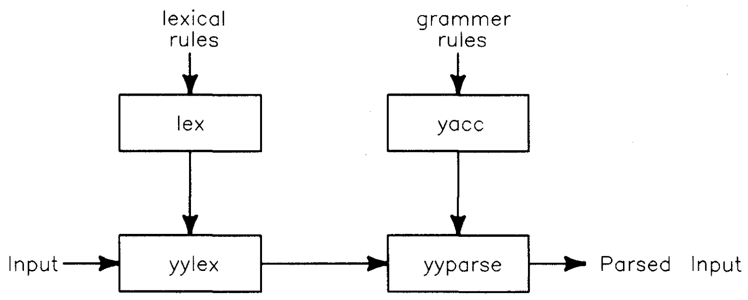

Figure 3-2. Using lex **with** yacc

The `lex` command 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 which include forward context require a significant amount of re-scanning. 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 user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered, as cases of a switch statement in C. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

The `lex` command is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for `ab` and another for `abcdefg`, and the input stream is `abcdefh`, `lex` will recognize `ab` and leave the input pointer just before `cd` ... Such backup is more costly than the processing of simpler languages.

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 thus

```
%%
```

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

In the outline of lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see the section "lex Regular Expression") 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. In this example 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 just 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 it is desired to change a number of words from British to American spelling. The lex command rules such as

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

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

lex Regular Expressions

The definitions of regular expressions are similar to those in *ed(1)*. 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`.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. 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. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the previous expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but space, tab, newline and the list above is always a text character.

Note that the initial percent operator (%) is special because it is the separator for `lex` source segments.

Character Classes

Classes of characters can be specified using the operator pair [ ]. 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 \, -, and ^. The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underscore. Ranges may be given in either order. Using - between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and will get a warning message. (For example, [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates 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; or

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

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

```
(dot or period)
```

is the class of all characters except newline. Escaping into octal is possible although non-portable:

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

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either `ac` or `abc`.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive `a` characters, including zero; while

```
a+
```

is one or more instances of `a`. For example,

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

is 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.

Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either `ab` or `cd`. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

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

matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.

Context sensitivity

The `lex` command will recognize a small amount of surrounding context. The two simplest operators for this are ^ and **$**. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is **$**, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, 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` as explained in the section on left context sensitivity. If a rule is only to be executed when the `lex` automaton interpreter is in start condition `x`, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition `ONE`, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions

The operators {} 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`.

```
a{2, }
```

matches two or more occurrences of `a`, while

```
a{3}
```

matches exactly three occurrences of a and is equivalent to aaa.

3

Operator Precedence

The `lex` command operators are handled according to the following rules of precedence:

- The `lex` command operators are ranked in the following order of precedence, beginning with highest precedence and proceeding to the lowest precedence:
- All operations on a single line have the same precedence.
  - `* ? +`
  - concatenation
  - repetition
  - `$ ^`
  - `|`
  - `/ <>`

lex Actions

When an expression is matched, lex executes the corresponding action. This section describes some features of lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When lex is being used with yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done 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 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

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

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

In more complex actions, the user will often want to know the actual text that matched some expression like [a-z]+. The lex command leaves this text in an external character array named yytext. Thus, to print the name found, a rule like

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

will print 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 is 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`:

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

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

Sometimes it is 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, the user 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 would overwrite 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. Further characters previously matched are returned to the input. This provides the same sort of look ahead offered by the / operator, but in a different form.

Example

Consider a language which defines a string as a set of characters between double quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression that matches such a string is somewhat confusing, so you might prefer to use:

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

which will, when faced with a string such as `"abc\"def "`, first match the five characters `"abc\` ; then the call to `yymore()` will cause 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 user processing.”

The function `yyless()` might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of `=-a`.[1] Suppose it is desired to treat this as `=- a` but print a message. A rule might be

3

```
=-[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 it might be desired to treat this 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 = ...
                }
```

will perform 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 would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity.

[1] In early versions of C, the assignment operators had the form `=op`. C now writes assignment operators in the form `op=` to avoid the ambiguity illustrated here.

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

1. input() which returns the next input character;
2. output(c) which writes the character c on the output; and
3. unput(c) pushes the character c back onto the input stream to be read later by input().

By default these routines are provided as macro definitions, but the user 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 strange 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 look-ahead will not work.

The lex command does not look ahead at all if it does not have to, but every rule ending in +, *, ?, or $ or containing / implies look-ahead. Look-ahead is also necessary to match an expression that is a prefix of another expression. For a discussion of the character set used by lex, read the section "Character Set" found in this manual. The standard lex library imposes a 100 character limit on backup.

Another lex library routine that the user will sometimes 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, the user should provide a yywrap which 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, etc., at the end of a program. Note that it is not possible to write a normal rule which 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.

Ambiguous Source Rules

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

3

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

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

to be 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`) will 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. But 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 will match

```
'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, will stop after `'first'`. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like `[.\n]+` or equivalents; the `lex` generated program will try 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 . does not include newline. Since she includes he, lex will normally *not* recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the user would like to 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 the user really wants 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 will then be counted. In this example, of course, the user 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 digraph table of the input is desired; normally the digraphs overlap, that is the word `the` is considered to contain both `th` and `he`. Assuming a two-dimensional array named `digraph` to be incremented, the appropriate source is

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

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

lex Source Definitions

Remember the format of the lex source:

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

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program 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. There are three classes of such things.

1. Any line which is not part of a lex rule or action which begins with a blank or tab is copied into the lex generated program. Such source input prior to the first %% delimiter will be 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 which begin with a blank or tab, and which contain 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.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., 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 beginning in column 1, is assumed to define `lex` substitution strings. The format of such lines is

```
name translation
```

3

and it causes the string given as a translation to be associated with the 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. Using `{D}` for the digits and `{E}` for an exponent field, for example, 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 does not 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. These possibilities are discussed below under "Summary of Source Format."

Usage

There are two steps in 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 in a file named lex.yy.c. The I/O library is defined in terms of the C standard library.

HP-UX

The library is accessed by the loader flag -ll for C, so an appropriate set of commands is

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

The resulting program is placed in the usual file a.out for later execution. To use lex with yacc see below. 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.

lex and yacc

3

If you want to use lex with yacc, note that what lex generates is a program named yylex(), the name required by yacc for its lexical analyzer. Normally, the default main program in the lex library calls this routine, but if yacc is loaded, and its main program is used, yacc will call 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 the line

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

in the last section of yacc input. If the grammar is gram.y and the lexical rules are scan.l the HP-UX command sequence can just be:

```
yacc gram.y
lex scan.l
cc y.tab.c -ly -ll
```

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

Alternatively, the -d option of yacc can be used to generate a file y.tab.h of token definitions. This can be included in the lex program by placing

```
%{
#include "y.tab.h"
%}
```

in the definitions section of the lex input file. If the grammar is gram.y and the lexical rules are in file scan.l, the HP-UX command sequence is:

```
yacc -d gram.y
lex scan.l
cc y.tab.c lex.yy.c -ly -ll
```

Examples

As a simple example, consider copying an input file while adding 3 to every positive number that is divisible by 7. Here is a suitable lex source program

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

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The 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. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

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

Numerical strings containing a . or preceded by a letter will be 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 which histograms the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100]; /*  Because this line has leading blanks,
                         *  it is copied to the lex.yy.c file (read
                         *  the section "lex Source Definitions."
                         */
%%
[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. To provide a `yywrap` that never returns true causes an infinite loop.

As a larger example, *here are some program fragments which convert* 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:

```
^" "{5} [^ 0]    ECHO;
```

In the regular expression, the quotes surround the blank and are follow by the repeat operator `{5}`. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of `^`. Here are some 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 == 'e';
        else if (*p == 'D')
            *p = 'E';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D` converting it to `e` or `E`, respectively. The modified constant, now single-precision, is written out again. There follow a series of names which 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.

Left-Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be 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 ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context 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 which 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 will only be 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.

Consider the following problem: copy the input to the output, changing the word magic to first on every line which began with the letter a, changing magic to second on every line which began with the letter b, and changing magic to third on every line which 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:

```
         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;
           }
         }
```

should be adequate.

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 can be abbreviated to s or S. The conditions can be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which 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, use:

```
BEGIN 0;
```

or

```
BEGIN INITIAL
```

which resets the initial condition of the `lex` automaton interpreter.

A rule may be active in several start conditions:

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

is a legal prefix. Any rule not beginning with the `<>` prefix operator is always active while in the normal state or any `%s` state. To specify that a rule is active *only* in the normal state, prefix it with `<INITIAL>`. Note that `INITIAL` is predefined by `lex`, and should not be included in a **`%start`** declaration.

The same example as before can be written:

```
%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");
```

where the logic is exactly the same as in the previous method of handling the problem, but `lex` does the work rather than the user's code.

The `lex` command also allows the definition of *exclusive start conditions*. The syntax is similar to that for *start conditions*, but the conditions are declared using %x or %X. For example:

%x *name1*, *name2*, ...

Exclusive start conditions differ from *start conditions* in how rules not beginning with the <> prefix operator are handled. When in a %x state, only rules explicitly prefixed by that <*state*> are active. Any rule not beginning with the <> prefix operator is not active.

The following example uses the `%x` exclusive start state. In this example, the scanner is looking for the keywords `first` and `second`; however, the symbol `##` puts the scanner into a "literal" mode where the normal patterns are not recognized. In this case, everything is just echoed (the default action) until another `##` symbol is reached that will put the scanner back into the initial state.

```
%x  LITERAL
%%
first  printf("FIRST");
second printf("SECOND");
## { BEGIN LITERAL; }
<LITERAL>##  { BEGIN INITIAL; }
```

3

Character Set

The programs generated by `lex` handle character I/O only through the routines `input`, `output`, and `unput`. Thus the character representation provided in these routines is accepted by `lex` and employed to return values in *yytext*. For internal use, a character is represented as a small integer which, 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 which translate the characters, `lex` must be told about it, by giving 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}
```

which indicate the value associated with each character. Thus the next example maps the lowercase and uppercase letters `a` through `z` together into the integers 1 through 26, the newline character into 27, `+` and `-` into 28 and 29 respectively, and the digits 0 through 9 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 can be assigned the number 0, and no character can be assigned a number that exceeds the size of the hardware character set.

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

Summary of Source Format

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

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

The definitions section contains a combination of

1. Definitions, in the form "name space translation".
2. Included code, in the form "space code".
3. Included code, in the form

    ```
    %{
    code
    %}
    ```

4. Start conditions, given in the form

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

5. Character set tables, in the form

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

6. 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:

Letter	Parameter
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:

Table 3-1.

x	the character "x"
"x"	an "x", even if x is an operator.
$\backslash x$	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

Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not 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 executed, the user 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 the not-yet-processed input.

4

yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an *input language* which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

The yacc command provides a general tool for describing the input to a computer program. The yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. The yacc command turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

The yacc command is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, Ratfor, etc., yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

Introduction

The `yacc` command provides a general tool for imposing structure on the input to a computer program. The `yacc` user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. The `yacc` command then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

The `yacc` command is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of `yacc` follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

`date` : *month_name day* `','` *year*;

Here, `date`, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and `year` are defined elsewhere. The comma , is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user supplied routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal* *symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name :  'F' 'e' 'b' ;


. . .

month_name :  'D' 'e' 'c' ;
```

might be used in the previous example. The lexical analyzer would only need to recognize individual letters, and `month_name` would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond `yacc`'s ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a `month_name` was seen; in this case, `month_name` would be a token.

Literal characters such as , must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying yacc has been described elsewhere [2] [3] [4]. The yacc command has been extensively used in numerous practical applications, including `lint` [5], the Portable C Compiler [6], and a system for typesetting mathematics [7].

The next several sections describe the basic process of preparing a yacc specification.

Sections in this Chapter	**Description**
"Basic Specifications"	Explains the preparation of grammar rules
"Actions"	Covers the preparation of the user supplied actions associated with the grammar rules
"Lexical Analysis"	Explains the preparation of lexical analyzers
"How the Parser Works"	Covers the operation of the parser
"Ambiguity and Conflicts"	Gives reasons why yacc may be unable to produce a parser from a specification, and what to do about it
"Precedence and Associativity"	Provides a simple mechanism for handling operator precedences in arithmetic expressions

Sections in this Chapter	Description
"Error Handling"	Covers error detection and recovery
"The yacc Environment"	Covers the operating environment and special features of the parsers yacc produces
"Hints for Debugging"	Explains how to debug a yacc grammar
"Hints for Preparing Specifications"	Gives some suggestions which should improve the style and efficiency of the specifications
"Advanced Topics"	Covers advanced features of yacc
"yacc Examples, Input Syntax, and Support"	Provides yacc examples, input syntax, and support information
"Acknowledgements"	Gives credit to those people who contributed to yacc

Basic Specifications

Names refer to either tokens or nonterminal symbols. The yacc command requires token names to be declared as such. In addition, for reasons discussed in the section "Lexical Analysis," it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent %% marks. (The percent % is generally used in yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), underscore (_), and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes. As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized. Thus

`'\n'`	`newline`
`'\r'`	`return`
`'\''`	``single quote ``''''``
`'\\'`	``backslash ``\''``
`'\t'`	`tab`
`'\b'`	`backspace`
`'\f'`	`form feed`
`'\`*xxx*`'`	``` ``*xxx*'' in octal ```

For a number of technical reasons, the NULL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar (|) can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D   ;
A       :       E  F   ;
A       :       G   ;
```

can be given to yacc as

```
A       :       B  C  D
        |       E  F
        |       G
;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :    ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. For more information, see the sections "Lexical Analysis", "Ambiguity and Conflicts", and "Precedence." Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the `%start` keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see the section "Lexical Analysis," below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces { and }. For example:

```
A       :       '('  B  ')'   {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                        {       printf("a message\n");
                                        flag = 25;   }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" $ is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable $$ to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ... , which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :        '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr      :          '('   expr   ')'               {  $$ = $2 ;   }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A         :          B     ;
```

frequently need not have an explicit action. This last rule is equivalent to

```
A:          B
              { $$ = $1; }
```

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. The yacc command permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A         :          B
                             {  $$ = 1;   }
                     C
                 {   x = $2;    y = $3;   }
          ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. The yacc command actually treats the above example as if it had been written:

```
$ACT      :          /* empty */
                             {  $$ = 1;   }
          ;


A         :          B  $ACT  C
                             {   x = $2;    y = $3;   }
          ;
```

A good understanding of how yacc handles actions can be important when interpreting conflict messages for such rules (see the section "Ambiguity and Conflicts"). For example, conflicts in the grammar specification occur when an interior action occurs in a rule before the parser can be sure which rule is being reduced.

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr    :       expr  '+'  expr
                         { $$ = node('+', $1, $3); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The yacc parser uses only names beginning in yy for parser variables; the user should avoid such names.

In these examples, all the values returned by actions and values of tokens are integers; a discussion of values of other types will be found in the section "Advanced Topics."

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called `yylex`. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable `yylval`.

4

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc`, or chosen by the user. In either case, the **`# define`** mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name `DIGIT` has been defined in the declarations section of the `yacc` specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                        . . .
        case '0':
        case '1':
          . . .
        case '9':
                 yylval = c-'0';
                 return( DIGIT );
                 . . .
                 }
        . . .
```

The intent is to return a token number of `DIGIT`, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier `DIGIT` will be defined as the token number associated with the token `DIGIT`.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names `if` or `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **`error`** is reserved for error handling, and should not be used naively (see the section "Error Handling").

As mentioned above, the token numbers may be chosen by `yacc` or by the user. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

The `lex` program is a very useful tool for constructing lexical analyzers. Lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. `lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

How the Parser Works

The `yacc` command turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by `yacc` consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called **`shift`**, **`reduce`**, **`accept`**, and **`error`**. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls `yylex` to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The **`shift`** action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
    IF          shift 34
```

which says, in state 56, if the lookahead token is `IF`, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The `reduce` action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
    .   reduce 18
```

refers to grammar rule 18, while the action

```
    IF          shift 34
```

refers to state 34.

Suppose the rule being reduced is

```
    A           :  x  y  z     ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is, in effect, a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
    A          goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable `yylval` is copied onto the value stack. After the return from the user code, the reduction is carried out. When the `goto` action is done, the external variable `yyval` is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

4 The other two parser actions are conceptually much simpler. The `accept` action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The `error` action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in the section "Error Handling."

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme    :       sound  place
         ;
sound    :       DING  DONG
         ;
place    :       DELL
         ;
```

When yacc is invoked with the -v option, a file called y.output is produced, with a human-readable description of the parser. The y.output file corresponding to the previous grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error
```

4

```
state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is `state 0`. The parser needs to refer to the input in order to decide between the actions available in `state 0`, so the first token, `DING`, is read, becoming the lookahead token. The action in `state 0` on `DING` is `shift 3`, so `state 3` is pushed onto the stack, and the lookahead token is cleared. Then `state 3` becomes the current state. The next token, `DONG`, is read, becoming the lookahead token. The action in `state 3` on the token `DONG` is `shift 6`, so `state 6` is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In `state 6`, without even consulting the lookahead, the parser reduces by rule 2.

```
    sound   :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering `state 0`. Consulting the description of `state 0`, looking for a `goto` on `sound`,

```
    sound   goto 2
```

is obtained; thus `state 2` is pushed onto the stack, becoming the current state.

In `state 2`, the next token, `DELL`, must be read. The action is `shift 5`, so `state 5` is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In `state 5`, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one `state,` 5, is popped off, and `state 2` is uncovered. The goto in `state 2` on *place*, the left side of rule 3, is `state 4`. Now, the stack contains 0, 2, and 4. In `state 4`, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering `state 0` again. In `state 0`, there is a goto on `rhyme` causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by `$end` in the `y.output` file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as `DING DONG DONG`, `DING DONG`, `DING DONG DELL DELL`, etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr        :       expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
(  expr  -  expr  )  -  expr
```

or as

```
expr  -  (  expr  -  expr  )
```

(The first is called *left association*, the second *right association*).

The yacc command detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr  -  expr  -  expr
```

When the parser has read the second expr, the input that it has seen:

```
expr  -  expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

```
-  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr  -  expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr  -  expr  -  expr
```

It could then apply the rule to the rightmost three symbols, reducing them to `expr` and leaving

```
expr  -  expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr  -  expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a **`shift / reduce`** conflict. It may also happen that the parser has a choice of two legal reductions; this is called a `reduce / reduce` conflict. Note that there are never any **`shift/shift`** conflicts.

When there are `shift/reduce` or **`reduce/reduce conflicts`**, `yacc` still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

The `yacc` command invokes two disambiguating rules by default:

- In a `shift/reduce` conflict, the default is to do the shift.
- In a `reduce/reduce` conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but `reduce/reduce` conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than `yacc` can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, `yacc` always reports the number of `shift/reduce` and `reduce/reduce` conflicts resolved by rule 1 and rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, `yacc` will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an `if-then-else` construction:

```
stat    :       IF  '('  cond  ')'  stat_
        |       IF  '('  cond  ')'  stat_ELSE  stat
        ;
```

In these rules, `IF` and `ELSE` are tokens, `cond` is a nonterminal symbol describing conditional (logical) expressions, and `stat` is a nonterminal symbol describing statements. The first rule will be called the `simple-if` rule, and the second the `if-else` rule.

These two rules form an ambiguous construction, since input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways:

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              }
ELSE  S2
```

or

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              ELSE  S2
              }
```

The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding "un-ELSE'd" IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a `shift/reduce` conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This `shift/reduce` conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

            stat  :  IF  (  cond  )  stat_            (18)
            stat  :  IF  (  cond  )  stat_  ELSE  stat

            ELSE     shift 45
            .        reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. Part of the description for state 45 is the line:

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

since the ELSE will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not ELSE, the parser reduces by grammar rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

Once again, notice that the numbers following **shift** commands refer to other states, while the numbers following **reduce** commands refer to grammar rule numbers. In the `y.output` file, the rule numbers are printed after those rules which can be reduced. In most states, there will be at most one **reduce** action possible in the state, and this will be the default command. The user who encounters unexpected **shift/reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[234] might be consulted; the services of a local guru might also be appropriate.

Precedence and Associativity

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr  :  expr  OP  expr
```

and

```
expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow `yacc` to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a `yacc` keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity: the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left  '+'  '-'
%left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than asterisk (*) and slash (/), which are also left associative. The keyword `%right` is used to describe right associative operators, and the keyword `%nonassoc` is used to describe operators, like the operator `.LT.` in FORTRAN, that may not associate with themselves; thus,

```
A  .LT.  B  .LT.  C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in yacc. As an example of the behavior of these declarations, the description

```
%right  '='
%left   '+'   '-'
%left   '*'   '/'

%%

expr    :       expr   '='   expr
        |       expr   '+'   expr
        |       expr   '-'   expr
        |       expr   '*'   expr
        |       expr   '/'   expr
        |       NAME
        ;
```

might be used to structure the input

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '−'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr     :       expr  '+'  expr
         |       expr  '-'  expr
         |       expr  '*'  expr
         |       expr  '/'  expr
         |       '-'  expr      %prec  '*'
         |       NAME
         ;
```

A token declared by `%left`, `%right`, and `%nonassoc` need not be, but may be, declared by `%token` as well.

The precedences and associativities are used by `yacc` to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a `reduce/reduce` conflict, or there is a `shift/reduce` conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a `shift/reduce` conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (`shift` or `reduce`) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of `shift/reduce` and `reduce/reduce` conflicts reported by `yacc`. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The `y.output` file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, `yacc` provides a simple, but reasonably general, feature. The token name `error` is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token `error` is legal. It then behaves as if the token `error` were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
    stat        :       error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
   stat    :       error  ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input :  error  '\n'  {  printf( "Reenter last line: " );  }  input
                      {  $$  =  $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
   yerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
   input :  error  '\n'
                   {       yyerrok;
                           printf( "Reenter last line: " );    }
            input
                   {       $$  =  $4;  }
           ;
```

As mentioned above, the token seen immediately after the `error` symbol is the `input` token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
    yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by `yylex` would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
    stat    :       error
                            {       resynch();
                                    yyerrok ;
                                    yyclearin ;    }
            ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to yacc, the output is a file of C programs, called y.tab.c. The function produced by yacc is called yyparse. It is an integer valued function. When it is called, it in turn repeatedly calls yylex, the lexical analyzer supplied by the user (see Chapter 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, yyparse returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called main must be defined, that eventually calls yyparse. In addition, a routine called yyerror prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using yacc, a library has been provided with default versions of main and yyerror. The name of this library is system dependent; on HP-UX systems the library is accessed by a -ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
      return( yyparse() );
      }
```

and

```
# include stdio.h

yyerror(s) char *s; {
                    fprintf( stderr, "%s\n", s );
                    }
```

The argument to yyerror is a string containing an error message, usually the string syntax error. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.) the yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

The yacc command provides command line options to allow some user modifications to the yacc environment. The -b option allows the user to change the default output filenames. For example:

```
    yacc -b xx gram.y
```

would produce a file xx.tab.c rather than y.tab.c.

The -p option specifies a prefix to replace yy in naming external procedures and variables. This can aid the linking of multiple parsers into a single program. Note that if this renaming did not take place, multiple declaration errors would occur at link time. The names changed are yyparse, yylex, yyerror, yylval, yychar, and yydebug. The renaming is done by emitting a set of #defines in the y.tab.c and y.tab.h files.

Hints for Debugging

Debugging a yacc grammar can be a challenge since the user's code (the yacc grammar specification) is a level of abstraction away from the C code that is being debugged. There are generally two areas where run time debugging is needed:

- determining that the grammar is parsing its input as expected
- debugging action code.

To aid in debugging the parsing, yacc provides a trace facility. The trace is enabled by compiling y.tab.c with the preprocessor symbol YYDEBUG defined to be nonzero. This is easily done by using the -t option of yacc:

```
yacc -t grammar source
```

Then the trace is turned on by setting the variable yydebug to a nonzero value. This can be done either by editing y.tab.c before compiling it, or while debugging in xdb or cdb. The trace gives information about which parser shift and reduce actions are performed.

The action code and the overall program can be debugged using the symbolic debugger xdb or cdb, but some special work must be done to remove # line constructs that confuse the debuggers. Normally, yacc inserts # line constructs into the generated y.tab.c file so that compile time errors refer to lines in the yacc source file rather than the y.tab.c file. However, this causes problems for the symbolic debuggers xdb and cdb because the line numbers may not be in increasing order. Once the y.tab.c file has been successfully compiled, then run yacc with the -l option so that no # line constructs are generated. This y.tab.c file can then be debugged using either xdb or cdb. Remember that the code lines in either xdb or cdb refer to lines in the y.tab.c file, not the original yacc source file.

Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the section "`yacc` Examples, Input Syntax, and Support" are written following this style, as are the examples in the text of this chapter (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable. It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :      /* empty */ |  seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ...  other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

    ...  other rules ...
```

The flag `dflag` is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of `backdoor` approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like `if`, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of `yacc`; it is difficult to pass information to the lexical analyzer telling it "this instance of '`if`' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

Advanced Topics

This section discusses a number of advanced features of yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes yyparse to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; yyerror is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider:

```
sent    :       adj  noun  verb  adj  noun
               {
look at the sentence
 . . .  }
        ;

adj     :       THE     {       $$ = THE;  }
        |       YOUNG   {       $$ = YOUNG;  }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG;  }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                                }
                                $$ = CRONE;
                        }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol noun in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The yacc command can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack (see the section "How the Parser Works") is declared to be a union of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as lint[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the yacc value stack, and the external variables yylval and yyval, to have type equal to this union. If yacc was invoked with the -d option, the union declaration is copied onto the y.tab.h file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of `%{` and `%}`.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a `union` member name. If this follows one of the keywords `%token`, `%left`, `%right`, and `%nonassoc`, the `union` member name is associated with the tokens listed. Thus, saying

```
%left  optype  '+'  '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate `union` member names with nonterminals. Thus, one might say

```
%type  nodetype  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no `a priori` type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa  {  $<intval>$  =  3;  } bbb
                     {       fun( $<intval>2, $<other>0 );  }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in the section "An Advanced Example." The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $*n* or to refer to something with no defined type is diagnosed. If these facilities are not triggered, the `yacc` value stack is used to hold `int`'s, as was true historically.

For parser efficiency, when an arbitrary `union` is defined for `YYSTYPE`, all of the members of the `union` should be kept to the size of an integer (for example, an integer or pointer). This is because the `yacc` value stack will be an array of these `union` types. If some `union` members are large, the stack will be large and copying stack values will be inefficient.

When larger structures are needed (for example, tree nodes in a compiler), it is recommended that allocation and deallocation of structures be handled by user supplied routines, and that the `union` member `YYSTYPE` be a pointer to the appropriate structure type.

yacc **Examples, Input Syntax, and Support**

This section contains the following information:

- An example of the yacc specification for a small desk calculator.
- An example of a grammar using some of yacc's advanced features.
- A description of the yacc input syntax.
- Old yacc features that are supported but not encouraged.

A Simple Example

This example gives the complete yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled a through z, and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start  list
```

```
%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '-'
%left  '*'  '/'  '%'
%left  UMINUS      /*  supplies  precedence  for  unary  minus  */

%%                 /*  beginning  of  rules  section  */

list    :       /*  empty  */
        |       list  stat  '\n'
        |       list  error  '\n'
                        {       yyerrok;  }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 );  }
        |       LETTER  '='  expr
                        {       regs[$1]  =  $3;  }
        ;

expr    :       '('  expr  ')'
                        {       $$  =  $2;  }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  '-'  expr
                        {       $$  =  $1  -  $3;  }
        |       expr  '*'  expr
                        {       $$  =  $1  *  $3;  }
        |       expr  '/'  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  '%'  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  '&'  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  '|'  expr
                        {       $$  =  $1  |  $3;  }
        |       '-'  expr         %prec  UMINUS
                        {       $$  =  -  $2;  }
        |       LETTER
```

```
                        {       $$  =  regs[$1];  }
        |       number
        ;

number  :       DIGIT
                        {       $$ = $1; base  =  ($1==0)  ?  8  :  10;  }
        |       number  DIGIT
                        {       $$  =  base * $1  +  $2;  }
        ;

%%                  /*  start  of  programs  */

yylex() {               /*  lexical  analysis  routine  */
              /* returns LETTER for a lower case letter, yylval = 0
through  25  */
              /* return DIGIT for a digit, yylval = 0 through 9 */
              /* all other characters are returned immediately */

        int  c;

        while(  (c=getchar())  ==  ' '  )  {    /*  skip  blanks  */  }

        /*  c  is  now  nonblank  */

        if(  islower(  c  )  )  {
                                yylval  =  c  -  'a';
                                return  (  LETTER  );
                                }
        if(  isdigit(  c  )  )  {
                                yylval  =  c  -  '0';
                                return(  DIGIT  );
                                }
        return(  c  );
        }
```

Advanced Example

This is an example of a grammar using some of the advanced features discussed in the section "Advanced Topics." The desk calculator example in the section "A Simple Example" in this section is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations `+`, −, ∗, `/`, unary −, and `=` (assignment), and has 26 floating point variables, `a` through `z`. Moreover, it also understands *intervals*, written

```
( x , y )
```

where `x` is less than or equal to `y`. There are 26 interval valued variables `A` through `Z` that may also be used. The usage is similar to that in the section "A Simple Example" in this appendix; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of `yacc` and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as `double`'s. This structure is given a type name, `INTERVAL`, by using `typedef`. The `yacc` value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of `YYERROR` to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of `yacc` is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through `yacc`: 18 `shift/reduce` and 26 `reduce/reduce`. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the `,` is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Also, while this example illustrates the use of arbitrary `yacc` stack value types, the `union` member `INTERVAL` results in every element of the `yacc` value stack being the size of a `struct` interval. For large structures, this can be very inefficient, and pointers to structures should be used instead. The user code must then supply routines to explicitly allocate and deallocate the structures.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine atof is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

#  include  <stdio.h>
#  include  <ctype.h>

typedef  struct  interval  {
                            double  lo,  hi;
                            }  INTERVAL;

INTERVAL  vmul(),  vdiv();

double  atof();

double  dreg[ 26 ];
INTERVAL  vreg[ 26 ];

%}

%start    lines

%union    {
          int  ival;
          double  dval;
          INTERVAL  vval;
          }

%token  <ival>  DREG  VREG      /*  indices  into  dreg,  vreg  arrays  */

%token  <dval>  CONST           /*  floating  point  constant  */

%type  <dval>  dexp             /*  expression  */

%type  <vval>  vexp             /*  interval  expression  */
```

```
        /*  precedence  information  about  the  operators  */

%left   '+'  '-'
%left   '*'  '/'
%left   UMINUS        /*  precedence  for  unary  minus  */

%%

lines   :       /*  empty  */
        |       lines  line
        ;

line    :       dexp  '\n'
                        {  printf( "%15.8f\n", $1 );}
        |       vexp  '\n'
                        {  printf( "(%15.8f, %15.8f)\n", $1.lo, $1.hi );}
        |       DREG  '='  dexp  '\n'
                        {     dreg[$1]  =  $3;}
        |       VREG  '='  vexp  '\n'
                        {     vreg[$1]  =  $3;}
        |       error  '\n'
                        {  yyerrok;}
        ;

dexp    :       CONST
        |       DREG
                        {       $$  =  dreg[$1];  }
        |       dexp  '+'  dexp
                        {       $$  =  $1  +  $3;  }
        |       dexp  '-'  dexp
                        {       $$  =  $1  -  $3;  }
        |       dexp  '*'  dexp
                        {       $$  =  $1  *  $3;  }
        |       dexp  '/'  dexp
                        {       $$  =  $1  /  $3;  }
        |       '-'  dexp       %prec  UMINUS
                        {       $$  =  - $2;  }
        |       '('  dexp  ')'
                        {       $$  =  $2;  }
        ;
```

```
vexp    :       dexp
                        {       $$.hi  =  $$.lo  =  $1;  }
        |       '('  dexp  ','  dexp  ')'
                        {
                        $$.lo  =  $2;
                        $$.hi  =  $4;
                        if($$.lo > $$.hi){
                                        printf("interval out of order\n");
                                        YYERROR;
                                        }
                        }
        |       VREG
                        {       $$  =  vreg[$1];    }
        |       vexp  '+'  vexp
                        {       $$.hi  =  $1.hi  +  $3.hi;
                                $$.lo  =  $1.lo  +  $3.lo;}
        |       dexp  '+'  vexp
                        {       $$.hi  =  $1  +  $3.hi;
                                $$.lo  =  $1  +  $3.lo;}
        |       vexp  '-'  vexp
                        {       $$.hi  =  $1.hi  -  $3.lo;
                                $$.lo  =  $1.lo  -  $3.hi;}
        |       dexp  '-'  vexp
                        {       $$.hi  =  $1  -  $3.lo;
                                $$.lo  =  $1  -  $3.hi;}
        |       vexp  '*'  vexp
                        {       $$  =  vmul(  $1.lo,  $1.hi,  $3  );}
        |       dexp  '*'  vexp
                        {       $$  =  vmul(  $1,  $1,  $3  );}
        |       vexp  '/'  vexp
                        {       if(  dcheck(  $3  )  )  YYERROR;
                                $$  =  vdiv(  $1.lo,  $1.hi,  $3  );}
        |       dexp  '/'  vexp
                        {       if(  dcheck(  $3  )  )  YYERROR;
                                $$  =  vdiv(  $1,  $1,  $3  );}
        |       '-'  vexp       %prec  UMINUS
                        {       $$.hi  =  -$2.lo;    $$.lo  =  -$2.hi;}
        |       '('  vexp  ')'
                        {       $$  =  $2;}
        ;
```

```
%%

# define BSZ 50    /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                              yylval.ival = c - 'A';
                              return( VREG );
                              }
        if( islower( c ) ){
                              yylval.ival = c - 'a';
                              return( DREG );
                              }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

            for( ; (cp-buf)&<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                         if(dot++ || exp) return( '.' );
                                       /* will cause syntax error */
                                           continue;
                                         }

                        if( c == 'e' ){
                                          if( exp++ ) return('e');
                                         /* will cause syntax error */
                                            continue;
                                            }

                            /*  end  of  number  */
                            break;
                            }
                  *cp  =  '\0';
                  if((cp-buf) >= BSZ) printf("constant too long: truncated\n");
                  else ungetc( c, stdin );  /* push back last char read */
                  yylval.dval  =  atof( buf );
                  return(  CONST  );
                  }
          return(  c  );
          }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
          /* returns the smallest interval containing a, b, c, and d */
          /*  used  by  *,  /  routines  */
          INTERVAL  v;

          if(  a>b  )  {  v.hi  =  a;    v.lo  =  b;  }
          else  {  v.hi  =  b;    v.lo  =  a;  }

          if(  c>d  )  {
                          if(  c>v.hi  )  v.hi  =  c;
                          if(  d<v.lo  )  v.lo  =  d;
                          }
          else            {
                          if(  d>v.hi  )  v.hi  =  d;
                          if(  c<v.lo  )  v.lo  =  c;
                          }
          return(  v  );
          }

INTERVAL  vmul(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
          return(  hilo(  a*v.hi,  a*v.lo,  b*v.hi,  b*v.lo  )  );
          }

dcheck(  v  )  INTERVAL  v;  {
          if(  v.hi  >=  0.  &&  v.lo  <=  0.  ){
                          printf(  "divisor  interval  contains  0.\n"  );
                          return(  1  );
                  }
        return(  0  );
        }

INTERVAL  vdiv(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
        return(  hilo(  a/v.hi,  a/v.lo,  b/v.hi,  b/v.lo  )  );
        }
```

Input Syntax

This is a description of the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
            /*  grammar  for  the  input  to  ``yacc''  */

        /*  basic  entities  */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal) followed by
colon    */
%token  NUMBER          /*    [0-9]+    */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /*  the  %%  mark  */
%token  LCURL   /*  the  %{  mark  */
%token  RCURL   /*  the  %}  mark  */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs  MARK  rules  tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
```

```
        |       /*  empty:  the  second  MARK  is  optional  */
        ;

defs    :       /*  empty  */
        |       defs  def
        ;

def     :       START  IDENTIFIER
        |       UNION  {  Copy union  definition  to  output  }
        |       LCURL  {  Copy  C  code  to  output  file   }  RCURL
        |       ndefs  rword  tag  nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /*  empty:  union  tag  is  optional  */
        |       '<'  IDENTIFIER  '>'
        ;

nlist   :       nmno
        |       nlist  nmno
        |       nlist  ','  nmno
        ;

nmno    :       IDENTIFIER          /* NOTE: literal illegal with
                                       %type  */
        |       IDENTIFIER  NUMBER  /* NOTE: illegal  with %type */
        ;

        /*  rules  section  */

rules   :       C_IDENTIFIER  rbody  prec
        |       rules  rule
        ;

rule    :       C_IDENTIFIER  rbody  prec
```

4

```
        |       '|'  rbody  prec
        ;

rbody   :       /*  empty  */
        |       rbody  IDENTIFIER
        |       rbody  act
        ;

act     :       '{'  {  Copy  action,  translate  $$,  etc.<<  }  '}'
        ;

prec    :       /*  empty  */
        |       PREC  IDENTIFIER
        |       PREC  IDENTIFIER  act
        |       prec  ';'
        ;
```

Old Features Supported but Not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

- Literals can also be delimited by double quotes `"`.
- Literals can be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

  The use of multi-character literals is likely to mislead those unfamiliar with `yacc`, since it suggests that `yacc` is doing a job which must be actually done by the lexical analyzer.
- Most places where `%` is legal, backslash `\` may be used. In particular, `\\` is the same as `%%`, `\left` the same as `%left`, etc.
- There are a number of other synonyms:
  - `%<` is the same as `%left`
  - `%>` is the same as `%right`
  - `%binary` and `%2` are the same as `%nonassoc`
  - `%0` and `%term` are the same as `%token`
  - `%=` is the same as `%prec`
- Actions can also have the form

  ```
  ={ ... }
  ```

  and the curly braces can be dropped if the action is a single C statement.
- C code between `%{` and `%}` used to be permitted at the head of the rules section, as well as in the declaration section.

Acknowledgements

B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder are to be recognized for their contribution of ideas to the current version of yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this chapter into English. Al Aho also deserves special credit for his help and favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys6*(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach. 18*(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Charry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach. 18* pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex – A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975). (See *HP-UX Concepts and Tutorials*, Vol. 1.)

Index

HEWLETT
PACKARD

Reorder No. or
Manual Part No.
B1864-90009

**Manufacturing
Part No.
B1864-90009**

B1864-90009