



CHAPTER 7

MAKE: A PROGRAM MAINTAINER

CONTENTS

7.1	Introduction.....	7-1
7.2	Description Files and Substitutions.....	7-5
7.3	Command Usage.....	7-7
7.4	Implicit Rules.....	7-8
7.5	Example.....	7-10
7.6	Suggestions and Warnings.....	7-11
7.7	Suffixes and Transformation Rules.....	7-13

## 7.1 Introduction

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. Make provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell Make the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the make command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of Make is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file defines the graph of dependencies. Make does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc or Lex). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

Make

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

make

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

Make is most useful for medium-sized programming projects. It does not solve the problems of maintaining multiple source versions or of describing huge programs.

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

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

```
#include "defs"
```

The following text describes the relationships and operations:

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

x.o y.o: defs
```

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

make

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

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that prog depends on three ".o" files. Once these object files are current, the second line describes how to load them to create prog. The third line says that x.o and y.o depend on the file defs. From the file system, Make discovers that there are three ".c" files corresponding to the needed ".o" files, and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of Make's default rules:

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

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

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

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

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

```
make
```

would just announce this fact and stop. If, however, the defs file had been edited, x.c and y.c (but not z.c) would be recompiled, and then prog would be created from the new ".o" files. If only the file y.c had changed, only it would be recompiled, but it would still be necessary to reload prog.

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

```
make x.o
```

would recompile x.o if x.c or defs had changed.

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

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

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

The last two invocations are identical. \$\$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: \$\*, \$@, \$?, and \$<. They will be discussed later. The following fragment shows the use:

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

The command

```
make
```

loads the three object files with the Lex -lln library. The command

```
make "LIBES=-lln -lm"
```

loads them with both the Lex (-ll) and the math (-lm)

libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in XENIX commands.)

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

## 7.2 Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

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

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

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the Make command line (see below).

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

```
target ... :[:] [dependent ...] [; commands] [# ...]
[(tab) commands] [# ...]
...
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "\*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

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

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed. Otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line. If the target is out of date with any of the files on a particular line, then the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

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

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

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



### 7.3 Command Usage

The `make` command takes four kinds of arguments: macro definitions, flags, description file names, and target file names. The syntax is as follows:

```
make [ flags ] [ macro definitions ] [ targets ]
```

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

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

Next, the flag arguments are examined. The permissible flags are

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

in the current directory is read. The contents of the description files override the built-in rules if they are present).

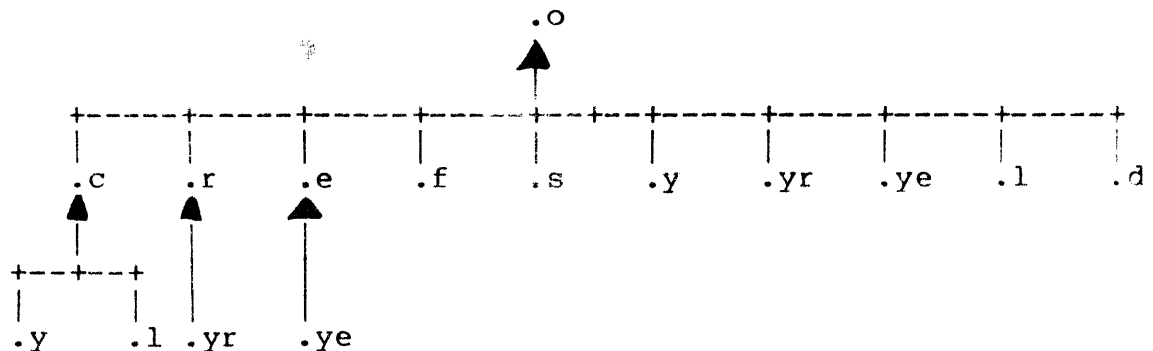
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

#### 7.4 Implicit Rules

The Make program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

```
.o      Object file
.c      C source file
.e      Efl source file
.r      Ratfor source file
.f      Fortran source file
.s      Assembler source file
.y      Yacc-C source grammar
.yr     Yacc-Ratfor source grammar
.ye     Yacc-Efl source grammar
.l      Lex source grammar
```

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



If the file x.o is needed and there is an x.c in the description or directory, it is compiled. If there is also

an x.l, that grammar would be run through Lex before compiling the result. However, if there is no x.c but there is an x.l, Make then discards the intermediate C-language file and uses the direct link in the graph above.

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

```
make CC=newcc
```

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

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

## 7.5 Example

As an example of the use of Make, we will present the description file used to maintain the `make` command itself. The code for Make is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = lpr
FILES = Makefile vers.c defs main.c doname.c misc.c files.c dosys.c\
      gram.y lex.c
OBJECTS = vers.o main.o ... dosys.o gram.o
LIBES=
LINT = lint -p
CFLAGS = -O

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

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

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

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

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

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

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

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

Make usually prints out each command before issuing it. The

following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

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

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

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

```
make print "P = lpr"
```

or

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

## 7.6 Suggestions and Warnings

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

recreate x.o.)

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

```
make -n
```

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

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of Make and destroys all memory of the previous relationships.

The debugging flag `-d` causes Make to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

## 7.7 Suffixes and Transformation Rules

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

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; Make looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, Make acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".r" file to a ".o" file is thus ".r.o". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro \$\* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro \$< is the name of the dependent that caused the action.

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

is to be changed).

The following is an excerpt from the default rules file:

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





CHAPTER 8  
AS: AN ASSEMBLER

CONTENTS

8.1	Introduction.....	8-1
8.2	Invocation.....	8-1
8.3	Invocation Options.....	8-2
8.4	Source Program Format.....	8-3
8.4.1	Label Field.....	8-4
8.4.2	Opcode Field.....	8-4
8.4.3	Operand-Field.....	8-5
8.4.4	Comment Field.....	8-5
8.5	Symbols and Expressions.....	8-5
8.5.1	Symbols.....	8-5
8.5.2	Assembly Location Counter.....	8-8
8.5.3	Program Sections.....	8-9
8.5.4	Constants.....	8-9
8.5.5	Operators.....	8-11
8.5.6	Terms.....	8-12
8.5.7	Expressions.....	8-12
8.6	Instructions and Addressing Modes.....	8-13
8.6.1	Instruction Mnemonics.....	8-13
8.6.2	Operand Addressing Modes.....	8-14
8.7	Assembler Directives.....	8-17
8.7.1	.ascii .asciz.....	8-17
8.7.2	.blkb .blkw .blkl.....	8-18
8.7.3	.byte .word .long.....	8-19
8.7.4	.end.....	8-19
8.7.5	.text .data .bss.....	8-19
8.7.6	.globl .comm.....	8-20
8.7.7	.even.....	8-21
8.8	Operation Codes.....	8-22
8.9	Error Messages.....	8-23

## 8.1 Introduction

This chapter describes the use of the XENIX assembler, named `as`, for the Motorola MC68000 microprocessor. It is beyond the scope of this manual to describe the instruction set of the 68000 or to discuss assembly language programming in general. For information on these topics, refer to the MC68000 16-Bit Microprocessor User's Manual, 3rd Edition, Englewood Cliffs, N.J: Prentice-Hall, Inc., 1982.

This chapter is organized as follows:

### The Command Line

Discusses assembler invocation and command line options.

### Source Program Format

Discusses the proper layout of an assembly language program, including specification of the label, opcode, operand, and comment fields.

### Symbols and Expressions

Discusses the symbols and expressions used in writing assembly language programs.

### Instructions and Addressing Modes

Discusses the available instructions and addressing modes.

### Assembler Directives

Discusses assembler directives.

### Operation Codes

Lists the available 68000 operation codes.

### Error Messages

Lists error messages that can be generated by `as`.

## 8.2 Invocation

`As` can be invoked with one or more arguments options. Except for option arguments, which must appear first on the command line, arguments may appear in any order on the command line. The source filename argument should be named filename.s. If a filename does not have the ".s" extension, the assembler prints a warning message, but still assembles the specified file. Note, that except as specified below, flags may be grouped. For example

```
as -glo that.o this.s
```

will have the same effect as

```
as -g -l -o that.o this.s
```

### 8.3 Invocation Options

The various flags and their function are:

**-o relname** The default output name is filename.o if assembling on an MC68000, and filename.b if cross assembling. This can be overridden by giving **as** the **-o** flag and giving the new filename in the argument following the **-o**. The **-o** must be the last argument in a flag bunch. Subsequent flags are ignored. For example

```
as -o that.o this.s
```

assembles the source this.s and puts the output in the file that.o.

**-l** By default, no output listing is produced. A listing may be produced by giving the **-l** flag. The listing filename extension is **L**". The filename for the list file is based on the output file. So the command line

```
as -l -o output.x input.s
```

produces a listing named output.L.

**-e** By default, all symbols go into the symbol table of the a.out(5) format file that is produced by the assembler, including locals. If you want only symbols that are defined as **.globl** or **.comm** to be included, you can give the **-e** (externals only) flag.

**-g** By default, if a symbol is undefined in an assembly, an error is flagged. This may be changed with the **-g** flag. If this is done, undefined symbols will be interpreted as external.

## 8.4 Source Program Format

An as program consists of a series of statements, each of which occupies exactly one line, i.e. a sequence of characters followed by the newline character. Form feed, ASCII <CONTROL-L>, also serves as a line terminator. Continuation lines are not allowed, and the maximum line length is 132 characters. However, several statements may be on a single line, separated by semicolons. Remember though, that anything after a comment character is considered a comment. The format of an as assembly language statement is:

```
[label-field] [opcode [operand-field] [;]] [| comment]
```

Most of the fields may be omitted under certain circumstances. In particular:

1. Blank lines are permitted.
2. A statement may contain only a label field. The label defined in this field has the same value as if it were defined in the label field of the next statement in the program. As an example, the two statements

```
name:
addl    d0,d1
```

are equivalent to the single statement

```
name:    addl    d0,d1
```

3. A line may consist of only the comment field; the two statements below are allowed as comments occupying full lines:

```
| This is a comment field.
| So is this.
```

4. Multiple statements may be put on a line by separating them with a semicolon (;). Remember, however, that anything after a comment character (including statement separators) is a comment.

In general, blanks or tabs are allowed anywhere in a statement; that is, multiple blanks are allowed in the operand field to separate symbols from operators. Blanks are significant only when they occur in a character string (e.g., as the operand of an `.ascii` pseudo-op) or in a character constant. At least one blank or tab must appear between the opcode and the operand field of a statement.

### 8.4.1 Label Field

A label is a user-defined symbol that is assigned the value of the current location counter; both of which are entered into the assembler's symbol table. The value of the label is relocatable.

A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. A maximum of 10 labels may be defined by a single source statement. The collection of label definitions in a statement is called the label-field.

The format of a label-field is:

```
symbol: [symbol:] ...
```

Examples:

```
start:
name: name2:      | Multiple symbols
7$:              | A local symbol, defined below
```

### 8.4.2 Opcode Field

The opcode field of an assembly language statement identifies the statement as either a machine instruction, or an assembler directive (pseudo-op). One or more blanks (or tabs) must separate the opcode field from the operand field in a statement. No blanks are necessary between the label and opcode fields, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. Some conventions used in `as` for instruction mnemonics are described later in a later section. A complete list of the opcodes is also presented.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data.

`As` is case sensitive. Operators and operands may only be lower case.

### 8.4.3 Operand-Field

A distinction is made between operand-field and operand in as. Several machine instructions and assembler directives require one or more arguments, and each of these is referred to as an "operand". In general, an operand field consists of zero, one, or two operands, and in all cases, operands are separated by a comma. In other words, the format for an operand-field is:

[operand [ , operand]...]

The format of the operand field for machine instruction statements is the same for all instructions. The format of the operand field for assembler directives depends on the directive itself.

### 8.4.4 Comment Field

The comment delimiter in as is the vertical bar, (|), not the semicolon, (;). The semicolon is the statement separator.

The comment field consists of all characters on a source line following and including the comment character. These characters are ignored by the assembler. Any character may appear in the comment field, with the obvious exception of the newline character, which starts a newline.

## 8.5 Symbols and Expressions

This section describes the various components of as expressions: symbols, numbers, terms, and expressions.

### 8.5.1 Symbols

A symbol consists of 1 to 32 characters, with the following restrictions:

1. Valid characters include A-Z, a-z, 0-9, period (.), underscore (\_), and dollar sign (\$).
2. The first character must not be numeric, unless the symbol is a local symbol.

There is no limit to the size of symbols, except the practical issue of running out of symbol memory in the

assembler. However, be aware that the current C compiler only emits 8 characters so a 9 or more character symbol that you think is the same in C and assembly may not match. Upper and lower cases are distinct, ("Name" and "name" are separate symbols). The period (.) and dollar sign (\$) characters are valid symbol characters, but they are reserved for system software symbols such as system calls and should not appear in user-defined symbols.

A symbol is said to be "declared" when the assembler recognizes it as a symbol of the program. A symbol is said to be "defined" when a value is associated with it. With the exception of symbols declared by a `.globl` directive, all symbols are defined when they are declared. A label symbol (which represents an address in the program) may not be redefined; other symbols are allowed to receive a new value.

There are several ways to declare a symbol:

1. As the label of a statement
2. In a direct assignment statement
3. As an external symbol via the `.globl` directive
4. As a common symbol via the `.comm` directive
5. As a local symbol

#### 8.5.1.1 Direct Assignment Statements

A direct assignment statement assigns the value of an arbitrary expression to a specified symbol. The format of a direct assignment statement is:

```
symbol = [symbol = ] ... expression
```

Examples of valid direct assignments are:

```
vect_size =      4
vectora =      /fffe
vectorb =      vectora-vect_size
CRLF =        /0D0A
```

Any symbol defined by direct assignment may be redefined later in the program, in which case its value is the result of the last such statement. A local symbol may be defined by direct assignment; a label or register symbol may not be redefined.



If the expression is absolute, then the symbol is also absolute, and may be treated as a constant in subsequent expressions. If the expression is relocatable, however, then symbol is also relocatable, and it is considered to be declared in the same program section as the expression. See the discussion in a later section for an explanation of absolute and relocatable expressions.

#### 8.5.1.2 Register Symbols

Register symbols are symbols used to represent machine registers. Register symbols are usually used to indicate the register in the register field of a machine instruction. The register symbols known to the assembler are given at the end of this chapter.

#### 8.5.1.3 External Symbols

A program may be assembled in separate modules, and then linked together to form a single program (see ld(1)). External symbols are defined in each of these separate modules. A symbol which is declared (given a value) in one module may be referenced in another module by declaring the symbol to be external in both modules. There are two forms of external symbols: those defined with the .globl directive and those defined with the .comm directive. See Section 8.7.6 for more information on these directives.

#### 8.5.1.4 Local Symbols

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of local symbols reduces the possibility of multiply-defined symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules.

Local symbols are of the form n\$ where n is any integer. Valid local symbols include:

27\$  
394\$

A local symbol is defined and referenced only within a single local symbol block (lsb). A new local symbol block is entered when either 1) a label is declared; or 2) a new program section is entered. There is no conflict between local symbols with the same name that appear in different local symbol blocks.

### 8.5.2 Assembly Location Counter

The assembly location counter is the period character (.); hence its name dot. When used in the operand field of any statement, dot represents the address of the first byte of the statement. Even in assembly directives, it represents the address of the start of the directive. A dot appearing as the third argument in a .byte directive would have the value of the address where the first byte was loaded; it is not updated "during" the pseudo-op.

For example:

```
movl .,dl      | load value of program counter into dl
```

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

. = expression

This expression must not contain any forward references, must not change from one pass to another, and must not have the effect of reducing the value of dot. Note that setting dot to an absolute position may not have quite the effect you expect if you are linking as's output file with other files, since dot is maintained relative to the origin of the output file and not the resolved position in memory. Storage area may also be reserved by advancing dot. For example, if the current value of dot is 1000, the direct assignment statement:

```
TABLE:  . = . + /100
```

would reserve 100 (hex) bytes of storage, with the address of the first byte as the value of TABLE. The next instruction would be stored at address 1100. Note that ".blkb 100" is a substantially more readable way of doing the same thing.

Note that the :p operator allows you to assemble values that are location relative both locally (within a module) and across module boundaries, without needing to do explicit address arithmetic.

### 8.5.3 Program Sections

As in XENIX, programs to `as` are divided into two sections: text and data. The normal interpretation of these sections is: instruction space and initialized data space, respectively.

In the first pass of the assembly, `as` maintains a separate location counter for each section, thus for code like

```

        .text
LABEL1: movw    d1,d2
        .data
LABEL2: .word 27
        .text
LABEL3: addl   d2,d1
        .data
LABEL4: .byte  4

```

in the output, LABEL1 will immediately precede LABEL3, and LABEL2 will immediately precede LABEL4. At the end of the first pass, `as` rearranges all the addresses so that the sections will be output in the following order: text, then data. The resulting output file is an executable image file with all addresses correctly resolved, with the exception of `.comm`'s and undefined `.globl`'s. For more information on the format of the output file, consult a.out(5).

### 8.5.4 Constants

All constants are considered absolute quantities when appearing in an expression.

#### 8.5.4.1 Numeric Constants

Any symbol beginning with a digit is assumed to be a number, and will be interpreted in the default decimal radix. Individual numbers may be evaluated in any of the five valid radices: decimal, octal, hexadecimal, character, and binary. The default decimal radix is only used on "bare" numbers, i.e. sequences of digits. Numbers may be represented in other radices as defined by the following table.

The other three radices require a prefix:

Radix	Prefix	Example	
octal	^ (up-arrow)	^17 equals 15. base 10.	
octal	0	^017 equals 15. base 10.	
hex	/ (slash)	/A1 equals 161. base 10.	
hex	0x	0xA1 equals 161. base 10.	
char	' (quote)	'a equals 97 base 10.	
char	' (quote)	'\n equals 10 base 10.	
binary	% (percent)	%11011 equals 27. base 10.	

Letters in hex constants may be upper or lower case; e.g., /aa=/Aa=/AA=170. Illegal digits for a particular radix generate an error (e.g. ^018). While the C character constant syntax is supported, you cannot define character constants by a number, (e.g., '\27) as this is more easily represented in one of the other formats.

## 8.5.5 Operators

### 8.5.5.1 Unary Operators

There are three unary operators in as:

Operator	Function
+	unary plus, has no effect.
-	unary minus.
~	logical negation.
:p	program displacement

The ":p" operator is a suffix that can be applied to a relocatable expression. It replaces the value of the expression with the displacement of that value from the current location (not dot). This is implemented with displacement relocation, so that it also works across modules.

### 8.5.5.2 Binary Operators

Binary operators in as include:

Operator	Description	Example	Value
+	Addition	3+4	7.
-	Subtraction	3-4	-1., or /FFFF
*	Multiplication	4*3	12.
/	Division	12/4	3.
	Logical OR	%01101   %00011	%01111
&	Logical AND	%01101&%00011	%00001
*	Remainder	5^3	2.

Each operator is assumed to work on a 32-bit number. If the value of a particular term occupies only 8 or 16 bits, the sign bit is extended into the high byte.

Sometimes error messages in expressions can be fixed by breaking the expressions into multiple statements using direct assignment statements.

### 8.5.6 Terms

A term is a component of an expression. A term may be one of the following:

- A. A number whose 32-bit value is used
- B. A symbol
- C. A term preceded by a unary operator. For example, both "term" and "~term" may be considered to be terms. Multiple unary operators are allowed; e.g. "+--+A" has the same value as "A".

### 8.5.7 Expressions

Expressions are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only one byte, (e.g. .byte), then the low-order 8 bits are used.

Expressions are evaluated left to right with no operator precedence. Thus "1 + 2 \* 3" evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpreted as having a value of zero. In this case, an "Invalid Expression" error will be generated. An "Invalid Operator" error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error will be generated if an expression contains a symbol with an illegal character, or if an incorrect comment character was used.

Any expression, when evaluated, is either absolute, relocatable, or external:

- A. An expression is absolute if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a direct assignment statement, is absolute. A relocatable expression minus a relocatable term, where both items belong to the same program section is also absolute.
- B. An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into core. All

labels of a program defined in relocatable sections are relocatable terms, and any expression which contains them must only add or subtract constants to their value. For example, assume the symbol sym was defined in a relocatable section of the program. Then the following demonstrates the use of relocatable expressions:

```

sym          relocatable
sym+5        relocatable
sym-'A       relocatable
sym*2        Not relocatable
2-sym        Not relocatable, since the expression cannot
              be linked by adding sym's offset to it.
sym-sym2     Absolute, since the offsets added to sym and
              sym2 cancel each other out.

```

- C. An expression is external (or global) if it contains an external symbol not defined in the current program. The same restrictions on expressions containing relocatable symbols apply to expressions containing external symbols. Exception: the expression "sym-sym2" where both sym and sym2 are external symbols is not allowed.

## 8.6 Instructions and Addressing Modes

This section describes the conventions used in as to specify instruction mnemonics and addressing modes.

### 8.6.1 Instruction Mnemonics

The instruction mnemonics used by as are described in the previously mentioned user's manual with a few variations. Most of the MC68000 instructions can apply to byte, word or to long operands, thus in as the normal instruction mnemonic is suffixed with *b*, *w*, or *l* to indicate which length of operand was intended. For example, there are three mnemonics for the add instruction: addb, addw, and addl.

Branch and call instructions come in 3 forms: the **bra**, **jra**, **bsr** and For the **bra** and **bsr** forms, the assembler will always produce a long (16-bit) pc relative address. For the **jra** and **jbsr** forms, the assembler will produce the shortest form of

binary it can. This may be 8-bit or 16-bit pc relative, or 32-bit absolute. The 32-bit absolute is implemented for conditional branches by inverting the sense of the condition and branching around a 32-bit `jmp` instruction. The 32-bit form will be generated whenever the assembler can't figure out how far away the addressed location is; for example, branching to an undefined symbol or a calculated value such as branching to a constant location.

### 8.6.2 Operand Addressing Modes

These effective addressing modes specify the operand(s) of an instruction. For details of the effective addressing modes, see section 2.10 of the MC68000 User's Manual. Note also that not all instructions allow all addressing modes. Details are given in the MC68000 User's Manual in appendix B under the specific instruction.

In the examples that follow, when two examples are given, the first example is based on the assembly format suggested by Motorola. The second example is in what is called "Register Transfer Language" or RTL. This is the format used by MIT and a number of other M68000 UNIX vendors, and is used by Motorola to describe technically the register transfers that are occurring within the machine, so it is provided for compatibility. Either syntax is accepted, and it is permissible to mix the two types of syntax within a module or even within a line when two effective address fields are allowed. Be aware, that a warning message will be generated when the assembler notices such a mix.

Many of the effective address modes have other names, by which they may be more commonly known. These name or names appear to the right of the Motorola name in parenthesis.

#### Data Register Direct

```
addl    d0,d1
```

#### Address Register Direct

```
addl    a0,a0
```

#### Address Register Indirect (indirect)

```
addl    (a0),d1
addl    a0@,d1
```

#### Address Register Indirect with Postincrement (autoinc)



```

movl    (a7)+,dl
movl    a7@+,dl

```

#### Address Register Indirect with Predecrement (autodec)

```

movl    dl,(a7)-
movl    dl,a7@-

```

#### Address Register Indirect with Displacement (indexed)

This form includes a signed 16-bit displacement. These displacements may be symbolic.

```

movl    l2(a6),dl
movl    a6@(l2),dl

```

#### Address Register Indirect with Index (double-indexed)

This form includes a signed 8-bit displacement and an index register. The size of the index register is given by following its specification with a ":w" or a ":l". If neither is specified, ":l" is assumed.

```

movl    l2(a6,d0:w),dl
movl    a6@(l2,d0:w),dl

```

#### Absolute Short Address

```

movl    xx:w,dl

```

#### Absolute Long Address (absolute)

Note that this is the address mode assumed should the given value be a constant. This is not true of branch and call instructions. Note also that the second example here is not RTL syntax, but it is provided because it is also allowed.

```

movl    xx,dl
movl    xx:l,dl

```

#### Program Counter with Displacement (pc relative)

When pc relative addressing is used, such as

```

pea name(pc)

```

the assembler will assemble a value that is equal to "name-.", where dot (.) is the position of the value, whether name is in the current module or not. You may also cause an expression to be pc relative by suffixing

it with a ":p". See also the displacement relocation mode in a.out(5).

```
movl    10(pc),dl
movl    pc@(10),dl
```

Note that if a symbol appears in the above addressing mode (where the 10 is in the example), the symbol's displacement from the extension word will be used in the instruction.

### Program Counter with Index

```
jmp     switchtab(pc,d0:1)
jmp     pc@(switchtab,d0:1)
switchtab:
```

### Immediate Data

Note that this is the way to get immediate data. If a number is given with no number sign (#), you get absolute addressing. This does not hold for `jsr` and `jmp` instructions. Also note that the second and third examples are not RTL syntax in particular.

```
movl    #47,dl
jmp     somewhere
moveq   #7,dl
```

In the `movem` instruction's register mask field, a special kind of immediate is allowed: the register list. Its syntax is as follows:

```
<reg [,reg]>
```

Here, reg is any register name. Register names may be given in any order. The assembler automatically takes care of reversing the mask for the auto-decrement addressing mode. Normal immediates are also allowed.

## 8.7 Assembler Directives

The following pseudo-ops are available in as:

<code>.ascii</code> <code>.asciz</code>	stores character strings
<code>.blkb</code> <code>.blkw</code> <code>.blkl</code>	saves blocks of bytes/words/longs
<code>.byte</code> <code>.word</code> <code>.long</code>	stores bytes/words/longs
<code>.end</code>	terminates program and identifies execution address
<code>.text</code> <code>.data</code> <code>.bss</code>	Text psect Data psect Bss psect
<code>.globl</code> <code>.comm</code>	declares external symbols declares communal symbols
<code>.even</code>	forces location counter to next word boundary

### 8.7.1 `.ascii` `.asciz`

The `.ascii` directive translates character strings into their 7-bit ascii (represented as 8-bit bytes) equivalents for use in the source program. The format of the `.ascii` directive is as follows:

```
.ascii    "character-string"
```

where character-string contains any character valid in a character constant. Obviously, a newline must not appear within the character string. (It can be represented by the escape sequence `"\n"` as described below). The quotation mark (") is the delimiter character, which must not appear in the string unless preceded by a backslash (\).

The following escape sequences are also valid as single characters:

X	Value of X
<code>\b</code>	<code>&lt;backspace&gt;</code> , hex /08
<code>\t</code>	<code>&lt;tab&gt;</code> , hex /09
<code>\n</code>	<code>&lt;newline&gt;</code> , hex /0A
<code>\f</code>	<code>&lt;form-feed&gt;</code> , hex /0C
<code>\r</code>	<code>&lt;return&gt;</code> , hex /0D
<code>\nnn</code>	hex value of nnn

Several examples follow:

<u>Hex Code Generated:</u>	<u>Statement:</u>
22 68 65 6C 6C 6F 20 74	<code>.ascii "hello there"</code>
68 65 72 65 22	
77 61 72 6E 69 6E 67 20	<code>.ascii "Warning-\007\007 \n"</code>
2D 07 07 20 0A	

The `.asciz` directive is equivalent to the `.ascii` directive with a zero (null) byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string. Null terminated strings are sometimes used as arguments to XENIX system calls.

### 8.7.2 `.blkb` `.blkw` `.blk1`

The `.blkb`, `.blkw`, and `.blk1` directives are used to reserve blocks of storage: `.blkb` reserves bytes, `.blkw` reserves words and `.blk1` reserves longs.

The format is:

```
[label:] .blkb  expression
[label:] .blkw  expression
[label:] .blk1  expression
```

where expression is the number of bytes or words to reserve. If no argument is given a value of 1 is assumed. The expression must be absolute, and defined during pass 1 (i.e. no forward references).

This is equivalent to the statement `".=.expression"`, but has a much more transparent meaning.

### 8.7.3 .byte .word .long

The `.byte`, `.word`, and `.long` directives are used to reserve bytes and words and to initialize them with values.

The format is:

```
[label:] .byte    [expression] [,expression]...
[label:] .word    [expression] [,expression]...
[label:] .long    [expression] [,expression]...
```

The `.byte` directive reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression. Note that multiple expressions must be separated by commas. A blank expression is interpreted as zero, and no error is generated.

For example,

```
.byte a,b,c,s    reserves 4 bytes.
.byte ,,,,      reserves five bytes, each with
                 a value of zero.
.byte           reserves a single byte, with a
                 value of zero.
```

The semantics for `.word` and `.long` are identical, except that 16-bit or 32-bit words are reserved and initialized. Be forewarned that the value of dot within an expression is that of the beginning of the statement, not of the value being calculated.

### 8.7.4 .end

The `.end` directive indicates the physical end of the source program. The format is:

```
.end
```

The `.end` is not really required; reaching the end of file has the same effect.

### 8.7.5 .text .data .bss

These statements change the "program section" where assembled code will be loaded.

### 8.7.6 .globl .comm

Two forms of external symbols are defined with the `.globl` and `comm` directives.

External symbols are declared with the `.globl` assembler directive. The format is:

```
.globl symbol [ , symbol ...]
```

For example, the following statements declare the array `TABLE` and the routine `SRCH` to be external symbols:

```
        .globl TABLE,SRCH
TABLE:  .blkw 10.
SRCH:   movw  TABLE,a0
```

External symbols are only declared to the assembler. They must be defined (i.e. given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

It is generally a good idea to declare a symbol as `.globl` before using it in anyway. This is particularly important when defining absolutes.

The other form of external symbol is defined with the `.comm` directive. The `.comm` directive reserves storage that may be communally defined, i.e., defined mutually by several modules. The link editor, `ld(1)` resolves allocation of `.comm` regions. The syntax of the `.comm` directive is:

```
.comm name constant-expression
```

which causes `as` to declare the name as a common symbol with a value equal to the expression. For the rest of the assembly this symbol will be treated as though it was an undefined global. `AS` does not allocate storage for common symbols; this task is left to the loader. The loader computes the maximum size of each common symbol that may appear in several load modules, allocates storage for it in the final bss section, and resolves linkages.

### 8.7.7 .even

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation to be on a word boundary after a `.byte` or `.ascii` directive. Note that many things may not be on an odd boundary in `as`, including instructions, and word and long data.

## 8.8 Operation Codes

Below are all opcodes recognized by as:

abcd	bmi	dbra	movb	rte
addb	bmis	dbt	movw	rtr
addw	bne	dbvc	movl	rts
addl	bnes	dbvs	movemw	sbcd
addqb	bpl	divs	moveml	scc
addqw	bpls	divu	movepw	scc
addql	bra	eorb	movepl	seq
addxb	bras	eorw	moveq	sf
addxw	bset	eorl	mulb	sge
addxl	bsr	exg	mulu	sgt
andb	bsrs	extw	nbcd	shi
andw	btst	extl	negb	sle
andl	bvc	jbsr	negw	sls
aslb	bvcs	jcc	negl	slt
aslw	bvs	jcs	negxb	smi
asll	bvss	jeq	negxw	sne
asrb	chk	jge	negxl	spl
asrw	clrb	jgt	nop	st
asrl	clrw	jhi	notb	stop
bcc	clrl	jle	notw	subb
bccs	cmpb	jls	notl	subw
bchg	cmpw	jlt	orb	subl
bclr	cmpl	jmi	orw	subqb
bcs	cmpmb	jmp	orl	subqw
bcss	cmpmw	jne	pea	subql
beq	cmpml	jpl	reset	subxb
beqs	dbcc	jra	rolb	subxw
bge	dbcs	jsr	rolw	subxl
bges	dbeq	jvc	roll	svc
bgt	dbf	jvs	rorb	svs
bgts	dbge	lea	rorw	swap
bhi	dbgt	link	rorl	tas
bhis	dbhi	lslb	roxlb	trap
ble	dble	lslw	roxlw	trapv
bles	dbls	lsl	roxll	tstb
bls	dbl	lsrb	roxrb	tstw
blss	dbmi	lsrw	roxrw	tstl
blt	dbne	lsrl	roxrl	unlk
blts	dbpl			



The following pseudo operations are recognized:

```
.ascii
.asciz
.blkb
.blkl
.blkw
.bss
.byte
.comm
.data
.end
.even
.globl
.long
.text
.word
```

The following registers are recognized:

```
d0 d1 d2 d3 d4 d5 d6 d7
a0 a1 a2 a3 a4 a5 a6 a7
sp pc cc sr
```

## 8.9 Error Messages

If there are errors in an assembly, an error message appears on the standard error channel (usually the terminal) giving the type of error and the source line number. If an assembly listing is requested, and there are errors, the error message appears before the offending statement. If there were no assembly errors, then there are no messages, thus indicating a successful assembly. Some diagnostics are only warnings and the assembly is successful despite the warnings.

If an assembly listing was not requested, any source lines which caused an assembly diagnostic are displayed on the terminal (the standard error file). In addition, a list of assembly errors and their description is also displayed on the terminal.

The common error codes and their probable causes, appear below:

### Invalid character

An invalid character for a character constant or character string was encountered.

**Multiply defined symbol**

A symbol has appeared twice as a label, or an attempt has been made to redefine a label using an = statement. This error message may also occur if the value of a symbol changes between passes.

**Offset too large**

A displacement cannot fit in the space provided for by the instruction.

**Invalid constant**

An invalid digit was encountered in a number.

**Invalid term**

The expression evaluator could not find a valid term that was either a symbol, constant or expression. An invalid prefix to a number or a bad symbol name in an operand will generate this.

**Non-relocatable expression**

Some instructions require relocatable expressions as operands. It was not provided.

**Invalid operand**

An illegal addressing mode was given for the instruction.

**Invalid symbol**

A symbol was given that does not conform to the rules for symbol formation.

**Invalid assignment**

An attempt was made to redefine a label with an = statement.

**Invalid opcode**

A symbol in the opcode field was not recognized as an instruction mnemonic or directive.

**Bad filename**

An invalid filename was given.

**Wrong number of operands**

An instruction has either too few or too many operands as required by the syntax of the instruction.

**Invalid register expression**

An operand or operand element that must be a register is not, or a register name is used where it may not be used. For example, using an address

register in a **moveq** instruction, which only allows data registers will produce this error message; as will using a register name as a label with a **bra** instruction.

**Odd address**

Something which must start at an even address does not.

**Inconsistent effective address syntax**

Both assembly and RTL syntax appear within a single module.

**Non-word memory shift**

An in memory shift instruction was given a size other than 16 bits.



## CHAPTER 9

### LEX: A LEXICAL ANALYZER

#### CONTENTS

9.1	Introduction.....	9-1
9.2	Lex Source.....	9-3
9.3	Lex Regular Expressions.....	9-4
	9.3.1 Character classes.....	9-5
	9.3.2 Arbitrary character.....	9-6
	9.3.3 Optional Expressions.....	9-6
	9.3.4 Repeated Expressions.....	9-6
	9.3.5 Alternation and Grouping.....	9-7
	9.3.6 Context Sensitivity.....	9-7
	9.3.7 Repetitions and Definitions.....	9-8
9.4	Lex Actions.....	9-9
9.5	Ambiguous Source Rules.....	9-13
9.6	Lex Source Definitions.....	9-16
9.7	Usage.....	9-17
9.8	Lex and Yacc.....	9-18
9.9	Left Context Sensitivity.....	9-22
9.10	Character Set.....	9-24
9.11	Summary of Source Format.....	9-25
9.12	Notes.....	9-27

## 9.1 Introduction

Lex 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 C program that recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written 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 written 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.

Lex 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 supported host language is C.

Lex turns the user's expressions and actions (called source in this section) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input here) and perform the specified actions for each expression as it is detected.

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

```
%%  
[ \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." 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.

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

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

In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. 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.

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

## 9.2 Lex Source

The general format of Lex source is:

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

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is 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 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 lex regular 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. Lex 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 is described later.

### 9.3 Lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (that match the corresponding characters in the strings being compared) and operator characters (these 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".

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 need not memorize the list above of current operator characters.

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

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. The quoting mechanism can also be used to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within brackets must be quoted. Several normal C escapes with \ are recognized:

```
\n newline
\t tab
\b backspace
\\ backslash
```

Since newline is illegal in an expression, a "\n" must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

### 9.3.1 Character classes

Classes of characters can be specified using brackets: [ and ]. 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 the backslash (\), the dash (-), and the up-arrow (^). The dash character indicates ranges. For example

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

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using the dash between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and causes a warning message. If it is desired to include the dash 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 up-arrow (^) 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 backslash (\) provides an escape mechanism within character class brackets, so that characters can be entered literally by preceding them with this character.

### 9.3.2 Arbitrary character

To match almost any character, the period (.) designates the class of all characters except a newline. Escaping into octal is possible although non-portable. For example

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

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

### 9.3.3 Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus

```
ab?c
```

matches either "ac" or "abc".

### 9.3.4 Repeated Expressions

Repetitions of classes are indicated by the asterisk (\*) and plus (+) operators. For example

```
a*
```

matches any number of consecutive "a" characters, including zero; while "a+" matches one or more instances of "a". For example,

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

matches all strings of lowercase letters, and

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

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

### 9.3.5 Alternation and Grouping

The vertical bar (|) operator indicates alternation. For example

```
(ab|cd)
```

matches either "ab" or "cd". Note that parentheses are used for grouping, although they are not necessary at the outside level. For example

```
ab|cd
```

would have sufficed in the preceding example. Parentheses can be used for more complex expressions, such as

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

which matches such strings as "abefef", "efefef", "cdef", and "cddd", but not "abc", "abcd", or "abcdef".

### 9.3.6 Context Sensitivity

Lex recognizes a small amount of surrounding context. The two simplest operators for this are the up-arrow (^) and the dollar sign (\$). If the first character of an expression is an up-arrow, then the expression is only matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of the up-arrow, complementation of character classes, since complementation only applies within brackets. If the very last character is dollar sign, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the slash (/) operator, 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 specifying start conditions as explained later. If a rule is only to be executed when the Lex automaton interpreter is in start condition "x", the rule should be enclosed in angle brackets:

```
<x>
```

If we considered "being at the beginning of a line" to be start condition ONE, then the up-arrow (^) operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

### 9.3.7 Repetitions and Definitions

The curly braces ({ and }) 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 the character "a".

Finally, an initial percent sign (%) is special, since it is the separator for Lex source segments.

## 9.4 Lex Actions

When an expression written as above 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. You 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 to use the repeat 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, you often want to know the actual text that matched some expression like

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

Lex leaves this text in an external character array named "yytext". Thus, to print the name found, a rule like

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

prints the string in "yytext". The C function printf accepts a format argument and data to be printed; in this

case, the format is "print string" (% indicating data conversion, and s indicating string type), and the data are the characters in "yytext". So this just places the matched string on the output. This action is so common that it may be written as ECHO. For example

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

is the same as the preceding example. 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 that 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 further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count of the number of characters matched. in the variable, "yyleng". To count both the number of words and the number of characters in words in the input, you might write

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

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

```
yytext[yylen-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, yyles(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 lookahead offered by the slash (/) operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that

to include a quotation mark in a string, it must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so that it might be preferable to write

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

which, when faced with a string such as

```
"abc\"def"
```

will first match the five characters

```
"abc\"
```

and 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 processing".

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

```
=-[a-zA-Z] {i
    printf("Operator (=-) ambiguous\n");
    yyless(yytext-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: The following performs the interpretation:



```

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

```

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. The possibility of "`==3`", however, makes

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

a still better rule.

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 lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in `+ * ?` or `$` or containing `/` implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100

character limit on backup.

Another Lex library routine that you 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.

## 9.5 Ambiguous Source Rules

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

- ⊕ The longest match is preferred.
- ⊕ Among rules 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) does not match the expression integer, so the identifier interpretation is used.

The principle of preferring the longest match makes certain constructions dangerous, such as the following:

```
.*
```

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 matches

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

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

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

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match a newline. Therefore, only no more than one line is ever matched by such expressions. Don't try to defeat this with expressions like

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

or their 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 the period (.) does not include the 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 to tell which input characters were in both classes.

Consider the two rules

```

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

```

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

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

```

%%
[a-z][a-z]      {digram[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.

## 9.6 Lex Source Definitions

Remember the format of the Lex source:

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

So far only the rules have been described. You will need additional options, though, to define variables for use in your 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 conventions of the C language.

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

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."

## 9.7 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 on a file named `lex.yy.c`. The I/O library is defined in terms of the C standard library.

```

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

```

The rule `[0-9]+` recognizes strings of digits; `atoi()` converts the digits to binary and stores the result in "k". 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:

```

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

```

Numerical strings containing a decimal point 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 makes histograms of word lengths, where a word is defined as a string of letters.

```

        int lengs[100];
%%
[a-z]+  lengs[yyval]++;
\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 parts of a program written to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```

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

```

An additional class recognizes white space:

```

W      [ \t]*

```

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

```

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

```

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



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

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of the caret (^) here. There follow 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 == 'D')
        *p+= 'e'-'d';
    ECHO;
}
```

After the floating point constant is recognized, it is scanned by the for loop to find the letter 'd' or 'D'. The program then adds 'e'-'d' which converts it to the next letter of the alphabet. 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}l{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.

## 9.9 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 circumflex (^) operator, for example, is a prior context operator, recognizing immediately preceding left context just as the dollar sign (\$) 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" may be abbreviated to "s" or "S". The conditions may be referenced at the head of a rule with angle brackets (< and >):

```
<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,

```
BEGIN 0;
```

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.

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.

## 9.10 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. For example:

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

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

### 9.11 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:

x	The character "x"
"x"	An "x", even if x is an operator.
\x	An "x", even if x is an operator.
[xy]	The character x or y.
[x-z]	The characters x, y or z.
[^x]	Any character but x.
.	Any character but newline.
^x	An x at the beginning of a line.

$\langle y \rangle x$	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.
$x\{m,n\}$	<u><math>m</math></u> through <u><math>n</math></u> occurrences of $x$ .

### 9.12 Notes

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.





CHAPTER 10  
YACC: A COMPILER-COMPILER

CONTENTS

10.1	Introduction.....	10-1
10.2	Basic Specifications.....	10-4
10.3	Actions.....	10-6
10.4	Lexical Analysis.....	10-9
10.5	How the Parser Works.....	10-11
10.6	Ambiguity and Conflicts.....	10-17
10.7	Precedence.....	10-22
10.8	Error Handling.....	10-25
10.9	The Yacc Environment.....	10-27
10.10	Hints for Preparing Specifications.....	10-28
10.11	Advanced Topics.....	10-32
10.12	A Simple Example.....	10-35
10.13	Yacc Input Syntax.....	10-38
10.14	An Advanced Example.....	10-40
10.15	Old Features.....	10-47

## 10.1 Introduction

Computer program input generally has some structure; 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 lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The name Yacc itself stands for "yet another compiler-compiler." The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc 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 class of specifications accepted is a very general one: LALR 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.

Yacc 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. Yacc 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.

Yacc 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 routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. 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 above 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 next several sections describe:

- The preparation of grammar rules

- ⊕ The preparation of the user supplied actions associated with the grammar rules
- ⊕ The preparation of lexical analyzers
- ⊕ The operation of the parser
- ⊕ Various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it.
- ⊕ A simple mechanism for handling operator precedences in arithmetic expressions.
- ⊕ Error detection and recovery.
- ⊕ The operating environment and special features of the parsers Yacc produces.
- ⊕ gives some suggestions which should improve the style and efficiency of the specifications.

## 10.2 Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed later, 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

For a number of technical reasons, the NUL 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. (See Sections 3, 5, and 6 for much more discussion). 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 section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

### 10.3 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.

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. Yacc 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. Yacc actually treats the above example as if it had been written:

```
$ACT : /* empty */
      { $$ = 1; }
      ;
A    : B $ACT C
      { x = $2; y = $3; }
      ;
```

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. Then 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"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in a later section.

#### 10.4 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.

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.

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. Hence, all lexical analyzers should be prepared to

return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is Lex, discussed in a previous section. These 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.

### 10.5 How the Parser Works

Yacc 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.

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 in a later section.

Consider the following example:

```
%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 above 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

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 underscore 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. 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 spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

## 10.6 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

$$\text{expr} : \text{expr} \text{ '-' } \text{expr}$$

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

$$\text{expr} - \text{expr} - \text{expr}$$

the rule allows this input to be structured as either

$$(\text{expr} - \text{expr}) - \text{expr}$$

or as

$$\text{expr} - (\text{expr} - \text{expr})$$

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

Yacc 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

$$\text{expr} - \text{expr} - \text{expr}$$

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

$$\text{expr} - \text{expr}$$

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:

$$- \text{expr}$$

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.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. 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. State 45 will have, as part of its description, 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 one states, there will be at most 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 might be consulted; the services of a

local guru might also be appropriate.

## 10.7 Precedence

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 star 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. The %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.

## 10.8 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 perform 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 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

```
yyerrok ;
```

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.

## 10.9 The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called `y.tab.c` on most systems. 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 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 many 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.

### 10.10 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.

- a. 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."
- b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
- c. 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.
- d. 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.
- e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (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 "back door" approach can be over done. Nevertheless, it represents a way of doing some things that are difficult 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.

For stylistic (and other) reasons, it is best that keywords be reserved; that is, be forbidden for use as variable names.



## 10.11 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. Yacc 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 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(1) 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 yyval 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 a later section. 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.

## 10.12 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 /* 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 lowercase 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 ) ) {

```

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

e

## 10.13 Yacc Input Syntax

This section has 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 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 { 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 /* Literal illegal with %type */
        | IDENTIFIER NUMBER /* Illegal with %type */
        ;

/* rules section */

rules    : C IDENTIFIER rbody prec
        | rules rule
        ;

rule     : C IDENTIFIER rbody prec
        | '|' rbody prec
        ;

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

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

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

```



## 10.14 An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example 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. 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 circumvented 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. However, 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.

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'
      | vexp '\n'
      { printf( "%15.8f\n", $1 ); }
      { printf( "(%15.8f, %15.8f )\n", $1.lo, $1.hi ); }

```

```

| DREG '=' dexp '\n'
  { dreg[$1] = $3; }
| VREG '=' vexp '\n'
  { vreg[$1] = $3; }
| error '\n'
  { yyerror; }
;

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 fp numbers */
```

```
/* lexical analysis */
```

```

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

    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( '.' );
                /* above causes syntax error */
            }
            continue;
        }
    }
}

```

```

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

        /* end of number */
        break;
    }
    *cp = '\0';
    if( (cp-buf) >= BSZ )
        printf( "constant too long: truncated\n" );
    else ungetc( c, stdin );
        /* above pushes 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 ) );  
}
```

## 10.15 Old Features

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

1. Literals may also be delimited by double quotes "".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or underscore, 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 that must be actually done by the lexical analyzer.
3. Most places where '%' is legal, backslash "\" may be used. In particular, "\\\" is the same as "%%", \left the same as %left, etc.
4. 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

```

5. Actions may also have the form

```

={ ... }

```

and the curly braces can be dropped if the action is a single C statement.

6. C code between %<sup>\*</sup>{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.