Since ambiguous pattern lists are acceptable as input to *lex*, we can simply insert more special cases as we encounter them.

Unlike in *yacc*, where only in the case of reduce/reduce conflicts does input order make a difference, it greatly matters in *lex* since conflicts in the sense of *yacc* are frequently encountered. This, unfortunately, often leads to illogical arrangements of the patterns. The order of patterns does imply something like a control structure.

2.5 "lex" programs

Input to *lex* consists of one file with three parts, separated by lines beginning in %%:

```
first part
%%
pattern          action
    ...
%%
third part
```

The first part is optional; it can contain lines controlling the dimensions of certain tables internal to *lex*, it can contain definitions for text replacements as we shall see in section 2.7, and it can contain (global) C code preceded by a line beginning in %{ and followed by a line beginning in %}. Even if the first part of the *lex* specification is empty, the separator %% between the first and second parts of the *lex* specification cannot be omitted.

The third part and the separator preceding it are optional. The third part contains C code which is used as is. As we shall see, it usually contains (local) functions which the second part uses.

The second part of the specification consists of a table of patterns and actions. This part of the specification is quite line-oriented. A pattern starts at the beginning of a line and extends to the first non-escaped white space. Following an arbitrary amount of white space, an action is specified which is thus associated with the pattern. The action is a single C statement, or several statements enclosed in a set of braces. The action may also consist of a bar |, indicating that the present pattern will use the same action as the next pattern.

lex input itself has no provisions for comments(!), but within braces, regular C comments can be written.

From the table, *lex* will construct a C function `yylex()` in the file *lex.yy.c*. If this function is linked with a program and called, standard input is read until the next, longest possible string represented by a pattern has been collected. The action associated with the pattern is then executed. If this action contains a `return` statement, `yylex()` will return, possibly with a function value as dictated by the `return` statement.

lex adds a default pattern and action to the patterns specified by the user in such a way that all otherwise unrecognized input characters are copied to standard output.

The following *lex* program will remove all upper case letters from its input:

```
%{
/*
 *      remove upper case letters
 */
%}

%%

[A-Z]+
```

Assuming that the program is contained in a file *exuc.l*, the following commands produce an executable program *exuc*:

```
lex exuc.l
cc lex.yy.c -ll -o exuc
```

`-ll` references the *lex* library, which contains a default `main()` function, which will just call `yylex()` once. This library must always be supplied when a function `yylex()` generated by *lex* is to be linked.

More useful actions need to have access to the input string recognized by the pattern with which they are associated. The char vector `yytext[]` contains this string, null-terminated, the int variable `yyleng` has `strlen(yytext)` as a value, and the int variable `yylineno` contains the number of the current input line. Let us look at a few marginally useful programs:

```
%{
/*
 *      line numbering
 */
%}

%%

\n      ECHO;
^.*$    printf("%d\t%s", yylineno, yytext);
```

This first program prints standard input and precedes each nonempty input line by its number and a tab character. `ECHO` is defined within the C program produced by *lex*; it causes `yytext[]` to be printed.

If we also want to number the empty lines, the following program can be used:

```
%{
/*
 *      line numbering
 */
%}

%%

^.*\n   printf("%d\t%s", yylineno-1, yytext);
```

Now we recognize line feed as part of the pattern. `yylineno` is already incremented to the next line when our action gets control.

The next program is an author's favorite utility:

```
%{
/*
 *      word count
 */

int     nchar, nword, nline;
%}

%%

\n              ++ nchar, ++ nline;
[^ \t\n]+       ++ nword, nchar += yyleng;
.               ++ nchar;

%%

main()
{
        yylex();
        printf("%d\t%d\t%d\n", nchar, nword, nline);
}
```

This example shows a use for the third part of a *lex* specification: we include our own `main()` function, which here prints the statistics gathered during execution of the `yylex()` function.

The next example is typical of a large class of problems, where a special action — here it is file inclusion — needs to be taken when just one pattern is recognized. Most of the input is just passed through. Similar applications include, e.g., adorning reserved words for the publication of a program, gathering a table of contents, etc.

```
%{
/*
 *      .so filename    file inclusion
 */

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

static include();
%}

%%

^".so".*\n      { yytext[yyleng-1] = '\0'; include(yytext+3); }

%%

static include(s)
        char * s;
{       FILE * fp;
        int i;

        while (*s && isspace(*s))
```

```
                ++ s;
        if (fp = fopen(s, "r"))
        {       while ((i = getc(fp)) != EOF)
                        output(i);
                fclose(fp);
        }
        else
                perror(s);
}
```

output() is the routine which *lex* uses to write all output. The program can only handle one level of file inclusion.

The last example deals with depositing output in several files, selected by options in the input. This is a variant of the *split* utility.

```
%{
/*
 *      .di filename    file splitting
 */

#include <ctype.h>
#include <stdio.h>
static divert();
%}

%%

^".di".*\n      { yytext[yyleng-1] = '\0'; divert(yytext+3); }

%%

static divert(s)
        char * s;
{
        while (*s && isspace(*s))
                ++ s;
        if (! freopen(s, "w", stdout))
                perror(s);
}
```

This example is quite difficult to handle with tools such as *sed* or *awk*, where the number of output files is severely limited. Since *lex* can pick the file name from within a complex context, this solution is significantly easier to modify than a hand-written C program.

2.6 Testing a lexical analyzer

For use as a lexical analyzer in a compiler, yylex() should be designed as a function returning an int value. Upon each call, the next terminal symbol should be collected from the input, encoded, and returned as the function value. Each pattern now is designed to recognize one or more terminal symbols, and the associated action will contain a return statement to produce the function value. We must make provisions so that *all* input characters are recognized, even if they neither belong to terminal symbols, nor are to be ignored silently, since yylex() is supposed to let all input

disappear in this case.

Testing such a function can be messy — the function values are numbers and as such usually not terribly mnemonic. We found a C programming trick quite helpful.[3] For debugging purposes we arrange for the *lex* program to have the following principal architecture:

```
%{
#include <assert.h>
#define token(x)        (int) "x"
main()
{       char * p;

        assert(sizeof(int) >= sizeof(char *));
        while (p = (char *) yylex())
                printf("%s is \"%s\"\n", p, yytext);
}
%}

%%

pattern         return token(MNEMONIC);
```

As the *sampleC* example will show, it is possible to build a lexical analyzer in such a way that it can always be conditionally compiled for testing purposes in this fashion.

The caller of `yylex()` receives the value `MNEMONIC` in printable form as a result value of `yylex()`. The simple `main()` routine will then display the input text `yytext` and a decoded representation of the returned value together for debugging purposes.

By convention, `yylex()` is expected to return zero(!) as an end of file indication. *lex* generates the function `yylex()` so that this happens internally; `main()` is coded to terminate once `yylex()` returns zero.

One last advice: the reserved words of a programming language are particularly easy to write down as patterns. Unfortunately, a long list of such self-representing patterns dramatically increases the size of the program generated by *lex*. It is much more efficient to collect reserved words and identifiers with the same, single pattern and to screen the results with a small C function. A standard approach to this problem is shown in the next section.

2.7 Example

From the *yacc* specification of *sampleC* we know which terminal symbols must be found by the lexical analysis routine for this compiler: `yylex()` needs to find all symbols mentioned in `%token` statements and all single-character terminal symbols quoted directly. At present, using the debugging technique introduced in the previous section, we do not need to worry about the exact function values which must be returned as encodings of the various terminal symbols.

---

[3] Unfortunately, this trick will not work on machines such as the MC 68000, where pointer values do not fit into `int` variables. The library macro `assert()` is employed to guard against this possibility; it will abort the program if the given condition is not met.

The patterns pose no problems, since we will eventually run the C preprocessor prior to our own compiler. The C preprocessor will remove comments(!), and it makes the usual #define, #include, and conditional compilation facilities available for our *sampleC* implementation.

The complete, final lexical analyzer is shown below. DEBUG must be defined when we compile the lexical analyzer for testing purposes. A few lines in the following file should therefore be ignored at present; they will be explained in chapter 3 where we integrate our language recognition program.

```
%{
/*
 *      sample c -- lexical analysis
 */

#ifdef  DEBUG           /* debugging version - if assert ok */

#       include <assert.h>

        main()
        {       char * p;

                assert(sizeof(int) >= sizeof(char *));

                while (p = (char *) yylex())
                        printf("%-10.10s is \"%s\"\n", p, yytext);
        }

        s_lookup() {}
        int yynerrs = 0;

#       define  token(x)        (int) "x"

#else   ! DEBUG         /* production version */

#       include "y.tab.h"
#       define  token(x)        x

#endif  DEBUG

#define END(v)  (v-1 + sizeof v / sizeof v[0])
static int screen();
%}

letter                          [a-zA-Z_]
digit                           [0-9]
letter_or_digit                 [a-zA-Z_0-9]
white_space                     [ \t\n]
blank                           [ \t]
other

%%

^"#"{blank}*{digit}+({blank}+.*)?\n     yymark();

">="                                    return token(GE);
```

```
"<="                                    return token(LE);
"=="                                    return token(EQ);
"!="                                    return token(NE);
"+="                                    return token(PE);
"-="                                    return token(ME);
"*="                                    return token(TE);
"/="                                    return token(DE);
"%="                                    return token(RE);
"++"                                    return token(PP);
"--"                                    return token(MM);

{letter}{letter_or_digit}*              return screen();

{digit}+                                { s_lookup(token(Constant));
                                          return token(Constant);
                                        }

{white_space}+

{other}                                 return token(yytext[0]);

%%

/*
 *      reserved word screener
 */

static struct rwtable {                 /* reserved word table */
        char * rw_name;                 /* representation */
        int rw_yylex;                   /* yylex() value */
        } rwtable[] = {                 /* sorted */
        "break",        token(BREAK),
        "continue",     token(CONTINUE),
        "else",         token(ELSE),
        "if",           token(IF),
        "int",          token(INT),
        "return",       token(RETURN),
        "while",        token(WHILE),
        };

static int screen()
{       struct rwtable * low = rwtable,
                       * high = END(rwtable),
                       * mid;
        int c;

        while (low <= high)
        {       mid = low + (high-low)/2;
                if ((c = strcmp(mid->rw_name, yytext)) == 0)
                        return mid->rw_yylex;
                else if (c < 0)
                        low = mid+1;
                else
                        high = mid-1;
        }
        s_lookup(token(Identifier));
```

```
        return token(Identifier);
}
```

Assuming that this text is in the file *samplec.l*, the lexical analyzer is compiled for testing purposes as follows:

```
lex samplec.l
cc -DDEBUG lex.yy.c -ll -o lexi
```

We can run some tests through this lexical analyzer to check if the proper terminal symbols are recognized. The simplest, complete program is as follows:

```
main() {}
```

For this program the lexical analyzer will return the following:

```
Identifier is "main"
yytext[0]  is "("
yytext[0]  is ")"
yytext[0]  is "{"
yytext[0]  is "}"
```

We took a very simple approach to recognizing single character operators and at the same time finding and signaling all illegal input characters: if no other pattern catches a character, it is returned itself as value of `yylex()`. In this fashion we obtain a compact lexical analyzer, and we will later deal with all input errors in a systematic way.

A few other features of the lexical analyzer are perhaps worth mentioning. *lex* has a rudimentary text replacement facility within the patterns. A line of the form

```
name    replacement
```

in the first part of the *lex* specification defines a replacement text for a name. The name can then be specified as

```
{name}
```

within a pattern, and the `replacement` will be substituted. We use this facility often, usually to make the patterns more transparent and mnemonic.

The first pattern deals with lines of the form

```
# linenumber filename
```

Such lines are produced by the C preprocessor as position stamps during file inclusion and selection for conditional compilation. As will be explained in chapter 3, `yymark()` is a function which we designed to record the relevant information for our own error messages.

As promised, we use a simple binary lookup function `screen()` to recognize reserved words as special cases of those terminal symbols recognized by the identifier pattern. `screen()` is really intended as a blueprint solution for such a screening problem. Notice how the `token` technique is even used in the reserved word table!

`s_lookup()` is a handle for symbol table management which we will use and explain beginning in chapter 5.