Lex & Yacc

Introduction from Lex

Lex Specification File

Introduction from Yacc

Yacc Specification File

Chapter 7

Lex & Yacc

1. Introduction

First phase in the compiler development is tokenization. That is, the input or source program is decomposed into tokens which are also known as lexeme's(thus this phase is also called as lexical analysis). Each language will be having its own lexical rules. Only after extracting the tokens, syntactical analysis is carried out to test the validity of the expression in terms of that language specific grammatical rules. After this, actual transformation to machine language (via assembly language in the case of C language) takes place.

Lex library is widely used for lexical analysis. However, in the recent years, under Free Software Foundation license, Flex (fast lexical analyzer) is distributed along with Gnu compiler package which can be effectively used for lexical analysis purpose. In compiler construction terminology these SW which are used for lexical analysis are called as scanners.

In addition to scanner development, Lex/Flex is also used for some other applications in system administration where text processing is needed.

1.1 Lex Specification File

In essence, while using lex or flex we have to first create a specification file (used to specify the tokenization rules, i.e regular expressions to represent the tokens of the language and also C code, calleds as rules) and has to be presented to **lex** command which generates a C language file known as **lex.yy.c** (in which yylex() and other functions given in Table 2 are defined) which when compiled with gcc (with –lfl option) we get an executable file which does the required tokenization.

The flex input or specification file consists of three sections namingly definitions, rules and user code.

```
%{


 %}

definitions
```

```
%%
rules
%%

user code
```

1.1.1 The Definitions Section

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start conditions*, which are explained in a later section.

Name definitions have the form:

Name definition

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

1.1.2 The Rules Section

The *rules* section of the flex input contains a series of rules of the form:

pattern   action

where the pattern must be unindented and the action must begin on the same line.

See below for a further description of patterns and actions.

1.1.3 The User Code Section

The user code section is simply copied to `lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

In the definitions and rules sections, any *indented* text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{‘ and ‘%}’ removed). The `%{‘, and ‘%}" must appear unindented on lines by themselves.

In the rules section, any indented or %{ } text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for POSIX compliance; see below for other such features).

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

1.1.4 Patterns

The patterns in the input are written using an extended set of regular expressions. These are:

`x'

match the character `x'

`.'

any character (byte) except newline

`[xyz]'

a "character class"; in this case, the pattern matches either an `x', a `y', or a `z'

`[abj-oZ]'

a "character class" with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'

`[^A-Z]'

a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

`[^A-Z\n]'

any character EXCEPT an uppercase letter or a newline

`*r**'

zero or more *r*'s, where *r* is any regular expression

`*r*+'

one or more *r*'s

`*r*?'

zero or one *r*'s (that is, "an optional *r*")

`*r*{2,5}'

anywhere from two to five *r*'s

`*r*{2,}'

two or more *r*'s

`*r*{4}'

exactly 4 *r*'s

`{*name*}'

the expansion of the "*name*" definition (see above)

`"[xyz]\"foo"'

the literal string: `[xyz]"foo'

`*x*'

if *x* is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \*x*. Otherwise, a literal `*x*' (used to escape operators such as `*')

`\0'

a NUL character (ASCII code 0)

`\123'

the character with octal value 123

`\x2a'

the character with hexadecimal value 2a

`(*r*)'

match an *r*; parentheses are used to override precedence (see below)

`*rs*'

the regular expression *r* followed by the regular expression *s*; called "concatenation"

`*r*|*s*'

either an *r* or an *s*

`*r*/*s*'

an *r* but only if it is followed by an *s*. The text matched by *s* is included when determining whether this rule is the *longest match*, but is then returned to the input before the action is executed. So the action only sees the text matched by *r*. This type of pattern is called *trailing context*. (There are some combinations of `*r*/*s*' that flex cannot match correctly; see notes in the Deficiencies / Bugs section below regarding "dangerous trailing context".)

`^*r*'

an *r*, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

`*r*$'

an *r*, but only at the end of a line (i.e., just before a newline). Equivalent to "*r*/\n". Note that flex's notion of "newline" is exactly whatever the C compiler used to compile flex interprets '\n' as; in particular, on some DOS systems you must either filter out \r's in the input yourself, or explicitly use *r*/\r\n for "r$".

`<*s*>*r*'

an *r*, but only in start condition *s* (see below for discussion of start conditions) <*s1*,*s2*,*s3*>*r* same, but in any of start conditions *s1*, *s2*, or *s3*

`<*>*r*'

an *r* in any start condition, even an exclusive one.

`<<EOF>>'

an end-of-file

`<*s1*,*s2*><<EOF>>'

an end-of-file when in start condition *s1* or *s2*

Note that inside of a character class, all regular expression operators lose their special meaning except escape ('\') and the character class operators, '-', ']', and, at the beginning of the class, '^'.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

foo|bar*
is the same as
(foo)|(ba(r*))
since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('|'). This pattern therefore matches *either* the string "foo" *or* the string "ba" followed by zero-or-more r's. To match "foo" or zero-or-more "bar"'s, use:

foo|(bar)*
and to match zero-or-more "foo"'s-or-"bar"'s:
(foo|bar)*

In addition to characters and ranges of characters, character classes can also contain character class *expressions*. These are expressions enclosed inside `[': and `:'] delimiters (which themselves must appear between the '[' and ']' of the character class; other elements may occur inside the character class, too). The valid expressions are:

[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]

These expressions all designate a set of characters equivalent to the corresponding standard C `isXXX' function. For example, `[:alnum:]' designates those characters for which `isalnum()' returns true - i.e., any alphabetic or numeric. Some systems don't provide `isblank()', so flex defines `[:blank:]' as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If your scanner is case-insensitive (the `-i' flag), then `[:upper:]' and `[:lower:]' are equivalent to `[:alpha:]'.

Some notes on patterns:

- A negated character class such as the example "[^A-Z]" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]"). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match the entire input unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character. The following are illegal:

  ```
  foo/bar$
  <sc1>foo<sc2>bar
  ```

  Note that the first of these, can be written "foo/bar\n". The following will result in '$' or '^' being treated as a normal character:

  ```
  foo|(bar$)
  foo|^bar
  ```

  If what's wanted is a "foo" or a bar-followed-by-a-newline, the following could be used (the special '|' action is explained below):

  ```
  foo     |
  bar$    /* action goes here */
  ```

  A similar trick will work for matching a foo or a bar-at-the-beginning-of-a-line.

1.1.5 How the input is matched

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the `flex` input file is chosen.

Once the match is determined which satisfying one of the regular expression or rule, the text corresponding to the match (called the *token*) is made available in the global character pointer **`yytext` (see Table 1 for other lex specific variables)**, and its length in the global integer `yyleng`. The *action* corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the *default rule* is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal `flex` input is: (see Example 1).

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

Note that `yytext` can be defined in two different ways: either as a character *pointer* or as a character *array*. You can control which definition `flex` uses by including one of the special directives `` `%pointer' `` or `` `%array' `` in the first (definitions) section of your flex input. The default is `` `%pointer' ``, unless you use the `` `-l' `` lex compatibility option, in which case `yytext` will be an array. The advantage of using `` `%pointer' `` is substantially faster scanning and no buffer overflow when matching very large tokens (unless you run out of dynamic memory). The disadvantage is that you are restricted in how your actions can modify `yytext` (see the next section), and calls to the `` `unput()' `` function destroys the present contents of `yytext`, which can be a considerable porting headache when moving between different `lex` versions.

The advantage of `` `%array' `` is that you can then modify `yytext` to your heart's content, and calls to `` `unput()' `` do not destroy `yytext` (see below). Furthermore, existing `lex` programs sometimes access `yytext` externally using declarations of the form:

```
extern char yytext[];
```

This definition is erroneous when used with `` `%pointer' ``, but correct for `` `%array' ``.
`` `%array' `` defines `yytext` to be an array of `YYLMAX` characters, which defaults to a fairly large value. You can change the size by simply #define'ing `YYLMAX` to a different value in the first section of

your `flex` input. As mentioned above, with `` `%pointer' `` yytext grows dynamically to accommodate large tokens. While this means your `` `%pointer' `` scanner can accommodate very large tokens (such as matching entire blocks of comments), bear in mind that each time the scanner must resize `yytext` it also must rescan the entire token from the beginning, so matching such tokens can prove slow. `yytext` presently does *not* dynamically grow if a call to `` `unput()' `` results in too much text being pushed back; instead, a run-time error results.

Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded.

Table Lex variables

yyin	Of the type FILE*. This points to the current file being parsed by the lexer.
yyout	Of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
yytext	The text of the matched pattern is stored in this variable (char*).
yyleng	Gives the length of the matched pattern.
yylineno	Provides current line number information. (May or may not be supported by the lexer.)

Table Lex functions

yylex()	The function that starts the analysis. It is automatically generated by Lex.
yywrap()	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing.
yyless(int n)	This function can be used to push back all but first ‘n’ characters of the read token.
yymore()	This function tells the lexer to append the next token to the current token.

Example 1

This contains no patterns and no actions. Thus, any string matches and default action, i.e printing takes place.

```
%{
%}
%%
%%
main()
{
yylex();
return 0;
}
```

Let the lex specification file as ex0.lex. Then, run the following sequence of commands.

```
lex   ex0.lex          (creates lex.yy.c)

gcc -o ex0  lex.yy.c  -lfl
```

To Use the generated program.

```
./ex0  < filename
```

This displays the content of the file “filename”.

Even this ex0 program can take standard input. That is, we can simply type its name along the command line.

```
./ex0
dd
ds
dads
^d
```

We can also say from this program “ex0” from go on take input till we type a specified string with the help of **“Here the document (<<)”**.

```
./ex0 <<END
```

```
dsdsa asdd
asdkads asdk
asdkd asd
asdsd
adsd sdd
END
```

This simply displays what ever we have typed on the screen again.

Example 2

```
A lex program which adds line numbers to the given file
and displays the same onto the standard output

%{
/*

*/

int lineno=1;
%}

line .*\n

%%
{line}    {    printf("%5d   %s",lineno++,yytext); }
%%
main()
{
yylex();
return 0;
}

If we assume this file name is "ex1.lex", at the command
prompt we have to execute the following commands to use
lex.

lex   ex1.lex          (creates lex.yy.c)

gcc -o ex1  lex.yy.c  -lfl
```

To use the program.

```
./ex1  < filename
```

This displays the content of the file “filename” along with line numbers.

This “ex1” program takes standard input also. Try at the command prompt.

```
./ex1
dd
ds
dads
^d
```

Also try at the command prompt the following.

```
./ex1 <<END
dsdsa asdd
asdkads asdk
asdkd asd
asdsd
adsd sdd
END
```

Example 3

This is also a lex specification program which adds line numbers to the given file and displays the same onto standard output Only difference is that main() is not included unlike previous example. However, automatically, main() is added by the lex.

```
%{
/*


*/
```

```
int lineno=1;
%}

line .*\n

%%
{line}    {    printf("%5d   %s",lineno++,yytext); }
%%
```

Try at the command prompt the following comamnds.

```
lex   ex2.lex          (creates lex.yy.c)

gcc -o ex2  lex.yy.c  -lfl

./ex2  < filename
```

Example 4

This is a lex specification program which adds line numbers to the given file and displays the same onto standard output. However, it explains about the use of external variable yyin. The resultant program takes filename to be tokenized as command line argument unlike previous programs ex0, ex1 and ex2.

```
%{
/*

*/

int lineno=1;
%}

line .*\n

%%
{line}    {    printf("%5d   %s",lineno++,yytext); }
```

```
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
return 0;
}
```

If we assume this file name is "ex3.lex", at the command prompt we have to execute the following commands to use lex.

```
lex   ex3.lex          (creates lex.yy.c)

gcc -o ex3  lex.yy.c  -lfl
```

How to Use the program.

```
./ex3  filename
./ex3  < filename
```

Both the commands displays the file “filename” content on the screen along with the line numbers.
Also, try at the command prompt and note the difference.

```
./ex3
dd
ds
dads
^d

./ex3 <<END
dsdsa asdd
asdkads asdk
asdkd asd
asdsd
adsd sdd
END
```

Example 5

This specification program is an attempt to emulate od command.

```
%{


%}

character .
newline        \n

%%
{character}    {    printf("%o ",yytext[0]); }
{newline}{    printf("%o ", '\n');}
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the specification file be “ex5.lex”. Run the following commands.

lex ex5.lex

gcc –o ex5 lex.yy.c –lfl

./ex5 filename

Compare the result with the following command.

```
od –t oC filename
```

Example 6

```
This program is an attempt to extract only comments from
a C program and display the same on standard output

%{

%}

comment       \/\*.*\*\/

%%
{comment}ECHO;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex6.lex”. Run the following commands.

lex ex6.lex

gcc –o ex6 lex.yy.c –lfl

./ex6 filename.c

Example 7

```
This lex specification program is an attempt to replace
all nonnull sequences of white spaces by a single blank
character. Here, pattern ‘ws” is specified as a series
of spaces of tab characters and action is specified as
return or print  a single space. Any other string is
returned as it is.


%{
%}
ws   [ \t]
```

```
%%
{ws}+    {printf(" "); }
.    {printf("%s",yytext);}

%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex7.lex”. Run the following commands.

lex ex7.lex

gcc –o ex7 lex.yy.c –lfl

./ex7 filename

Example 8

```
This specification program replaces all the occurrences
of "rama" with "RAMA" and "sita" with "SITA". This
example is used from explain that we can use a string as
a direct pattern in the specification file.


%{
%}

%%
"rama"    {printf("RAMA");}

"sita"    {printf("SITA"); }
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
```

```
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex8.lex”. Run the following commands.

lex ex8.lex

gcc –o ex8 lex.yy.c –lfl

./ex8 filename

Example 9

```
This lex specification program is used to count all
occurrences of "rama" and "sita" in a given file.

%{
int count=0;

%}


%%
"rama"    {count++;}

"sita"    {count++; }
.    ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("No of Occurrences=%d\n",count);
return 0;
}
```

Let the above specification file be “ex9.lex”. Run the following commands.

lex ex9.lex

gcc –o ex9 lex.yy.c –lfl

./ex9 filename

Example 10

```
This lex specification program is used from generate a C
program which removes all the occurences of “sita” and
“rama” in the given file.

%{
%}

%%
"rama"
"sita"
.    ECHO;

%%
```

Let the above specification file be “ex9a.lex”. Run the following commands.

lex ex9a.lex

gcc –o ex9a lex.yy.c –lfl

./ex9a< filename

Example 11

This lex specification program is used to count and print the number of pages, lines, words and characters in a given file.

```
%{

int lines=0,words=0,characters=0,pages=0;
```

```
%}
%START    InWord
NewLine   [\n]
WhiteSpace     [\t ]
NewPage        [\f]


%%

{NewPage}{BEGIN 0; characters++;pages++;}
{NewLine}{BEGIN 0; characters++;lines++;}
{WhiteSpace}  {BEGIN 0; characters++;}
<InWord>.characters++;
.         {BEGIN   InWord; characters++; words++;}
%%
int main()
{
yylex();
printf("%d %d %d %d\n",lines,words,characters,pages);
}
```

Let the above specification file be “ex10.lex”. Run the following commands.

lex ex10.lex

gcc –o ex10 lex.yy.c –lfl

./ex10 filename

Example 12

This lex specification program also can be used from find no of lines, words and characters in a given file. Here, yyleng indicates the length of the string yytext.

```
%{

int lines=0,words=0,characters=0;
```

```
%}
word [^ \t\n]+
eol  \n
%%
{word}    {words++;characters+=yyleng;}
{eol}     {characters++;lines++;}
.         { characters++; }
%%
int main()
{
yylex();
printf("%d %d %d \n",lines,words,characters);
}
```

Let the above specification file be “ex11.lex”. Run the following commands.

lex ex11.lex

gcc –o ex11 lex.yy.c –lfl

./ex11 filename

Example 13

```
This lex specification program is to replace  all the
occurences of the word "username" with users login name.

%{
%}

%%
username  printf("%s",getlogin() );
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex12.lex”. Run the following commands.

lex ex12.lex

gcc –o ex12 lex.yy.c –lfl

./ex12 filename

Example 14

```
This lex specification program is to  extract all html
tags in the given file.

%{
%}

%%
"<"[^>]*>{printf("%s\n", yytext); }
.    ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex13.lex”. Run the following commands.

lex ex13.lex

gcc –o ex13 lex.yy.c –lfl

./ex13 filename.html

Example 15

```
This lex specification program  is from generate a
program which simulates  cat command to create files.
```

While giving input first "start" word has from be typed and at the end "end" word has from be typed.

```
%{
%}

%%
"start"    ;
"end"     exit(-1);
.    ECHO ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyout=fopen(argv[1],"w");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex14.lex”. Run the following commands.

lex ex14.lex

gcc –o ex14 lex.yy.c –lfl

./ex14 filename

Example 16

This lex specification program is to eliminate multiple spaces and tabs and replace with a single space and remove empty lines. Here, yytext is processed in our actions.

```
%{

#include<stdlib.h>
#include<stdio.h>
```

```
int emptyline=0;
%}

SPACES    [ \t]
eol  \n

%%
{SPACES}+   {printf(" "); }
\n|.      {
           char c=yytext[0];
          if(!isspace(c))  {
                    emptyline=1;
                    putchar(c);
                    }
          if(c=='\n')
               {
               if(emptyline)putchar(c);
               emptyline=0;
               }
             }
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Example 17

This lex specification program is to display only C comments in a given C file. Here, whenever “/*” pattern is encountered in the input, we have written code to process next characters in the input till we encounter the pattern “*/”. Whenever, “/*” pattern is encountered the C code will be executed. Here, we have used lex specific function (see Table)input() is used to read

characters from the lex input buffer and print them till it encounters “*/” pattern.

```
%{
%}

%%
"/*" {char c;
     int done=0;
     ECHO;
     do
     {
          while((c=input())!='*') putchar(c);
     putchar(c);
     while((c=input())=='*') putchar(c);
     putchar(c);
     if(c=='/') done=1;
     }    while(!done);
     }

.    ;
\n   ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Example 18

This lex specification file is to display a file’s content by replacing “:” with \t

```
%{

%}

%%
":"  {printf("\t");}

\n|. ECHO ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("\n");
return 0;
}
```

Let the above specification file be “ex17.lex”. Run the following commands.

lex ex17.lex

gcc –o ex17 lex.yy.c –lfl

./ex17 /etc/passwd

`REJECT` directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in "How the Input is Matched", and `yytext` and `yyleng` set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the `flex` input file, or one which matched less text

Example 19

For example, to count all instance of `she` and `he`, including the instances of `he` that are included in `she`, use the following action:

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

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

%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("No of ocurrence of he including in he in she=%
d\n",s);
return 0;
}
```

After counting the occurrences of `she`, the **lex** command rejects the input string and then counts the occurrences of `he`. Because `he` does not include `she`, a **REJECT** action is not necessary on `he`.

Example 20

The following lex specification file is used to generate a C program which counts number of words in a file other than the word “incl”.

```
%{
int nw=0;
%}
%%
incl       nw--; REJECT;
[^ \t\n]+ nw++;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
printf("No of words other than the word incl=%d\n",nw);
return 0;
}
```

Example 21

The following lex specification program generates a C program which takes a string “abcd” and prints the following output. From terminate the program enter ^d.

abcd

abc

ab

a

```
%{
%}
%%
a|ab|abc|abcd printf("%s\n",yytext);REJECT
.|\n

%%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
return 0;
}
```

Example 22

The following lex specification file generates a C program which extract http, ftp or telnet tags from the given file.

```
%{
%}
%%
(ftp|http|telnet):\/\/[^ \n<>"]*
.|\n
%%
main(int argc, char*argv[])
```

```
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
return 0;
}
```

The above program is supposed extract URL's with capitol letters such as HTTP, FTP or TELENT then we tell flex to build a case-insensitive lexer using the **"-i"** option.

Example 23

The following lex program generates a C program which takes standard input as output of Unix date and gives either of the following messages

Good Morning

Good Afternoon

Good Evening

```
%{
%}
%%
Morning         [ ](00|01|02|03|04|05|06|07|08|09|10|11)[:]
Afternoon       [ ](12|13|14|15|16|17)[:]
Evening         [ ](18|19|20|21|22|23)[:]

%%
{Morning}       printf("Good Morning ");
{Afternoon}     printf("Good Afternoon ");
{Evening}       printf("Good Evening ");
.               ;
```

If we assume that executable file name of the generated C program is "greet" then we can run the following command from see the output.

```
date | greet
```

BEGIN followed by the name of a start condition places the scanner in the corresponding start condition .

Example 24

The `yymore()' tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of yytext rather than replacing it. First, `yymore()' depends on the value of yyleng correctly reflecting the size of the current token, so you must not modify yyleng if you are using `yymore()'. Second, the presence of `yymore()' in the scanner's action entails a minor performance penalty in the scanner's matching speed.

Example 25

The yyless(n)' returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. yytext and yyleng are adjusted appropriately (e.g., yyleng will now be equal to *n*). An argument of 0 to yyless will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using BEGIN, for example), this will result in an endless loop. Note that yyless is a macro and can only be used in the flex input file, not from other source files.

Example 26

The following lex specification file is used to generate scanner program for a toy Pascal-like language which extracts integers type of numbers, float type of numbers, key words such as if, procedure etc.

```
 %{
#include <math.h>
#include<stdlib.h>
 %}

 DIGIT    [0-9]
 ID       [a-z][a-z0-9]*

 %%
```

```
{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*         {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n",
yytext );

"{"[\^{$\;$}}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n",
yytext );

%%

main( argc, argv )
int argc;
char **argv;
{
++argv, --argc;  /* skip over program name */
if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
else
      yyin = stdin;

yylex();
```

```
  }
```

Example 27
A lex input file that changes all numbers to hexadecimal in input file while ignoring all others.

```
%{

#include<stdlib.h>
 %}

digit  [0-9]
number {digit}+
%%
{number} { int n = atoi(yytext); printf("%x", n); }

• {;}
 %%
main(int argc, char*argv[])
{
extern FILE *yyin;
yyin=fopen(argv[1],"r");
yylex();
return 0;
}
```

Example 28

The lex specification file generates a C program which counts number of word with length 1 character, 2 characters vice versa in the given input.

```
%{
int leng[100];

%}
%%
[a-z]+    leng[yyleng]++;
.|\n ;
%%
main(int argc, char*argv[])
{
```

```
extern FILE *yyin;
int i;
for(i=1;i<100;i++) leng[i]=0;
yyin=fopen(argv[1],"r");
yylex();
printf("Word Length Frequency\n");
for(i=1;i<100;i++)
printf("%11d %10d\n",i,leng[i]);
return 0;
}
```

Example 29

```
%{
int leng[100];

%}
%%
[a-z]+   leng[yyleng]++;
.|\n ;
%%
int yywrap()
{
int i;
printf("Word Length   Frequency\n");

for(i=1;i<100;i++)
if(leng[i]) printf("%11d %10d\n",i,leng[i]);
return 1;
}
main(int argc, char*argv[])
{
extern FILE *yyin;
int i;
for(i=1;i<100;i++) leng[i]=0;
yyin=fopen(argv[1],"r");
yylex();
return 0;
}
```

Example 30

This lex specification file is used to generate a C program which counts how many times a given alphabet is next to another alphabet in the given input (This frequency table is called as digram in natural language processing terminology and also in algorithms.

```
%{
#include<stdlib.h>
int F[26][26];

%}
%%
[A-Za-z][A-Za-z] {
     F[toupper(yytext[0])-65][toupper(yytext[1])-65]
++;REJECT;
     }
.|\n ;
%%
main(int argc, char*argv[])
{
extern FILE *yyin;
int i,j;
for(i=0;i<26;i++)
for(j=0;j<26;j++)F[i][j]=0;

yyin=fopen(argv[1],"r");
yylex();
for(i=0;i<26;i++){
for(j=0;j<26;j++) printf("%d", F[i][j]);
printf("\n");
}
return 0;
}
```

If we give the following input to the program which is created compiling lex.yy.c file generated from the above lex specification file.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

```
Result is:

02000000000000000000000000
00200000000000000000000000
00020000000000000000000000
00002000000000000000000000
00000200000000000000000000
00000020000000000000000000
00000002000000000000000000
00000000200000000000000000
00000000020000000000000000
00000000002000000000000000
00000000000200000000000000
00000000000020000000000000
00000000000002000000000000
00000000000000200000000000
00000000000000020000000000
00000000000000002000000000
00000000000000000200000000
00000000000000000020000000
00000000000000000002000000
00000000000000000000200000
00000000000000000000020000
00000000000000000000002000
00000000000000000000000200
00000000000000000000000020
00000000000000000000000002
10000000000000000000000000
```

Example 31

In the C program generated by the lex specification file, when `yylex()` finished with the first file, it calls `yywrap()`, which opens the next file, and `yylex()` continues. When `yywrap()` has exhausted all the command line arguments, it returns 1, and `yylex()` returns with value 0 (but we don't use the return value).

```
Like previous examples, this also extracts URL's from
the given files whose names are given along the command
line to the above program.
```

```
%{
#include <stdio.h>
#include <errno.h>
int file_num;
int file_num_max;
char **files;
extern int errno;
%}
%%
(ftp|http):\/\/[^ \n<>"]*      printf("%s\n",yytext);
.|\n                ;
%%
int main(int argc, char *argv[]) {
      file_num=1;
      file_num_max = argc;
      files = argv;
      if ( argc > 1 ) {
            if ( (yyin = fopen(argv[file_num],"r")) ==
0 ) {
                  perror(argv[file_num]);
                  exit(1);
            }
      }
      while( yylex() )
            ;
      return 0;
}

int yywrap() {
      fclose(yyin);
      if ( ++file_num < file_num_max ) {
            if ( (yyin = fopen(files[file_num],"r")) ==
0 ) {
                  perror(files[file_num]);
                  exit(1);
            }
            return 0;
      } else {
            return 1;
```

```
        }
}
```

Start Conditions

`Also, flex` provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc". For example,

```
<STRING>[^"]*          {                                    }
```

will be active only match is found when the scanner is in the "STRING" start condition. Also, if we want same action to be done for a given regular expression under different states, we may have to enter all the states names like the following manner

<STRING, ERROR, WARNING>"*" ;

where STRING, ERROR, and WARNING are different states.

Example 32

This example is used to remove C comments from the file. It uses states facility available in the lex. Initially, lex is assumed to be in INITIAL state and when "/*" pattern is found it will be changing to COMMENT state. When it is in COMMENT state then pattern "*/" puts the lex in again INITIAL state. When it is in COMMENT state, whatever it encounters it will be eaten whereas if it is in INITIAL state simply the same is displayed.

```
%{
%}


%s COMMENT

%%
<INITIAL>"//".*          ;
<INITIAL>"/*"      BEGIN COMMENT;
```

```
<INITIAL>.          ECHO;
<INITIAL>[\n]       ECHO;
<COMMENT>"*/"       BEGIN INITIAL;
<COMMENT>.          ;
<COMMENT>[\n]       ;
%%

main() {
  yylex();
}
```

Example 33

The following lex specification file is used to generate a C program which is used to generate a html file from a data file.

The input format is that of a textual spreadsheet. Each spreadsheet entry is numbered using an alphabetical character (indicating the row) and an integer (indicating the column). To make the task easier, we can assume several things. First there will not be more than 26 columns. Second, that the entries will be ordered in the obvious way, i.e. A1, followed by B1, ... then A2, followed by B2, etc. Third, there will only be one entry per line and that all entries exist in the file. Finally, we are going to assume that there are no spreadsheet equations.

```
An example input might be

A0 = "Name"
B0 = "SSN"
C0 = "HW1"
D0 = "HW2"
A1 = "Scott Smith"
B1 = "123-44-5678"
C1 = 82
D1 = 44.2
A2 = "Sam Sneed"
B2 = "999-88-7777"
C2 = 92
D2 = 84
```

For output, we what to generate the appropriate HTML table. In html, tables are surrounded by <table> ... </table>. Each row of the table is surrounded by <tr> ... </tr> and each entry in a row by <td> ... </td>. For the above input we would want to output:

```
<table>
<tr>
<td> Name </td>
<td> SSN </td>
<td> HW1 </td>
<td> HW2 </td>
</tr>
<tr>
<td> Scott Smith </td>
<td> 123-44-5678 </td>
<td> 82 </td>
<td> 44.2 </td>
</tr>
<tr>
<td> Sam Sneed </td>
<td> 999-88-7777 </td>
<td> 92 </td>
<td> 84 </td>
</tr>
</table>
```

When viewed with a viewer, this looks like:

Name	SSN	HW1	HW2
Scott Smith	123-44-5678	82	44.2
Sam Sneed	999-88-7777	92	84

Lex specification for the conversion

```
%%
```

```
^A0       {printf("<table>\n<tr>\n");}
^A[0-9]*  {printf("</tr>\n<tr>\n");}
^[B-Z][0-9]*  ;
[0-9]*        {printf("<td> "); ECHO; printf("
</td>\n");}
[0-9]*"."[0-9]*    {printf("<td> "); ECHO; printf("
</td>\n");}
\"[^"]*\"{printf("<td> "); yytext[yyleng-1] = ' ';
printf("%s </td>\n",yytext+1);}
=         ;
[ \n]          ;
.         ECHO;
%%

main() {
  printf("<html>\n");
  yylex();
  printf("</tr>\n</table>\n</html>\n");
}
```

Example 34

The following specification file is for the opposite purpose. That is, given html languages <table> spefication such the following it has to extract only field's information.

Sample Input:

```
<table>
<tr>
<td> Name </td>
<td> SSN </td>
<td> HW1 </td>
<td> HW2 </td>
</tr>
<tr>
<td> Scott Smith </td>
<td> 123-44-5678 </td>
<td> 82 </td>
<td> 44.2 </td>
</tr>
```

```
<tr>
<td> Sam Sneed </td>
<td> 999-88-7777 </td>
<td> 92 </td>
<td> 84 </td>
</tr>
</table>
```

Sample output:

```
 Name  SSN  HW1  HW2  Scott Smith  123-44-5678  82  44.2
Sam Sneed  999-88-7777  92  84
```

Lex Specification File

```
%{
%}


%s  TABL REC DATA

%%
<INITIAL>"<table>"     BEGIN TABL;
<TABL>"<\/table>"      BEGIN INITIAL;
<TABL><tr>    BEGIN REC;
<REC><\/tr>   BEGIN TABL;
<REC><td> BEGIN DATA;
<DATA><\/td>  BEGIN REC;
<DATA>.       ECHO;
<DATA>[ \t\n]     ;
<REC>[ \t\n]  ;
<TABL>[ \t\n] ;
%%

main() {
  yylex();
}
```

Example 35

The above specification file can be also written as the following by combining last three state conditions.

```
%{
%}


%s  TABL REC DATA

%%
<INITIAL>"<table>"          BEGIN TABL;
<TABL>"<\/table>"           BEGIN INITIAL;
<TABL><tr>  BEGIN REC;
<REC><\/tr>  BEGIN TABL;
<REC><td>   BEGIN DATA;
<DATA><\/td>        BEGIN REC;
<DATA>.             ECHO;
<DATA,REC,TABL>[ \t\n]          ;
%%

main() {
  yylex();
}
```

Example 36

This example is used to explain yymore() and other pattern usage. Run the program by giving input like BOMBAYB.

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

%%
B[^B]*    { if (flag == 0) {
                    flag = 1;
                    yymore();
                }
```

```
            else     {
                flag = 0;
        printf("%s", yytext);
                printf("%d\n",yyleng);
              }
            }
%%

int main()
{
yylex();
}
```

1.2Yacc - A Parser Generator

Normally, syntax analysis is employed to validate or check the syntax of any program. SW systems which are used for this purpose is referred as parsers. It *is* also possible to create a simple parser using Lex alone by making extensive use of the user-defined states (ie start-conditions). However, such a parser quickly becomes unmaintainable, as the number of user-defined states tends to explode.

Once our input file syntax contains complex structures, such as "balanced" brackets, or contains elements which are context-sensitive, compiler-compilers such as **yacc (yet another compiler compiler)** is best available alternative. "Context-sensitive" in this case means that a word or symbol can have different interpretations, depending on *where* it appears in the input language. For example in C, the '*' character is used for both multiplication, and to specify indirection (ie to dereference a pointer to a piece of memory). That is, it's meaning is "context-sensitive".

Yacc provides a general tool for imposing structure on the input to a computer program. That is, yacc user prepares a specification of the input process (grammar rules) as explained in detail in the following sections. Then, when yacc command is used with this specification file as input, it generates a C language program which checks the grammar (specified in the **.y** file) in the given input file. This generated C language program in turn calls lexical analyser, probably generated by using Lex command, to pick up the basic items (called tokens) and test their syntactical validity according to the specified gramatical rules. Thus, both lex and yacc commands are used while writing compilers. For detailed discussion on compilers one can refer [Aho, 1985].

To summarise, the steps in developing compilers (parsers) using Yacc and Lex are:

- Write the grammar in a .y file (also specify the actions here that are to be taken in C).
- Write a lexical analyzer to process input and pass tokens to the parser whenever it is needed. This can be done using Lex command as explained in previous section. That is, we may prepare Lex specification file also.

- Write error handling routines (like yyerror()).
- Run yacc command on .y file such that it gives **y.tab.c** file and **y.tab.h** file.
- Run lex command on lex specification file such that it gives **lex.yy.c** file.
- Compile code produced by Yacc and lex as well as any other relevant source files.
- Test the resulting executable file by giving input file.

1.2.1 The Yacc Specification Rules

Like lex, **yacc** has it's own specification language. A yacc specification is structured along the same lines as a Lex specification. By convention, a Yacc file has the suffix .y. The Yacc compiler is invoked from the compile line as **yacc -dv file.y**

```
%{
   /* C declarations and includes */
%}
   /* Yacc token and type declarations */
%%
   /* Yacc Specification
      in the form of grammer rules like this:
   */
   symbol    :    symbols tokens
                      { $$ = my_c_code($1); }
             ;
%%
   /* C language program (the rest) */
```

The Yacc Specification rules are the place where you "glue" the various tokens together that lex has conveniently provided to you.

Each grammar rule defines a symbol in terms of:

- other symbols
- tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

Terminal and non-terminal symbols

Terminal symbol: Represents a class of syntactically equivalent tokens. Terminal symbols are of three types:

Named token: These are defined via the %token identifier. By convention, these are all upper case.

Character token: A character constant written in the same format as in C. For example, + is a character token.

Literal string token: is written like a C string constant. For example, "<<" is a literal string token.

The lexer returns named tokens.

Non-terminal symbol: Is a symbol that is a group of non-terminal and terminal symbols. By convention, these are all lower case.

For example, in English one of the valid form of a sentence is the one having subject, verb and object.

Sentence: Subject Verb Object

Similarly, in US style, date is represented as:

Date: Month / Day / Year

Here, Date can be termed as Non-terminal and Month, '/', Day, and Year can be termed as terminals (i.e they are not further decomposable). In Yacc specification, the same rule can be specified as:

Date: MONTH '/' DAY '/' YEAR { /*actions */}

Actual values for MONTH, DAY and YEAR are returned from the lexical analyzer by tokenizing the given input.

Similarly, Context-free grammar production such as:

p->AbC

will have equivalent Yacc Rule as:

p : A b C { /* actions */}

The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case (Surprise!!. Exactly opposite to Automata Theory or compiler construction books notation's).

Also, we can use few Yacc specific declarations which begins with a %sign in yacc specification file such as:

1. **%union** It defines the Stack type for the Parser. It is a union of various datas/structures/ objects.

2. **%token** These are the terminals returned by the yylex function to the yacc. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member> tokenName.

3. **%type** The type of a non-terminal symbol in the Grammar rule can be specified with this.

The format is %type <stack member> non-terminal.

4. **%noassoc** Specifies that there is no associativity of a terminal symbol.

5. **%left** Specifies the left associativity of a Terminal Symbol

6. **%right** Specifies the right assocoativity of a Terminal Symbol.

7. **%start** Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.

8. **%prec** Changes the precedence level associated with a particular rule to that of the following token name or literal.

Let us discuss about how to write a parser to recognize US style date.

File date.y

```
%token NUMBER SLASH NL
%%
date :
     |NUMBER SLASH NUMBER SLASH NUMBER NL {printf("OK");
}
     ;
%%
void yyerror(char *s)
{
  printf(“Error\n”);
}
```

File date.lex

```
%{
```

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

%%
[0-9]+        return NUMBER;
"\/"      return SLASH;
'\n'      return NL;
```

Run the following command.

yacc –dv date.y

This command generates the files y.tab.c, y.tab.h and y.output.

File y.tab.h contains:

```
#ifndef YYERRCODE
#define YYERRCODE 256
#endif

#define NUMBER 257
#define SLASH 258
#define NL 259
```

The file y.tab.c contains the C code for the parser which recognizes the given grammatical rules (here it is date format) in a given input file.

Also, run the following commands to generate the final parser.

lex date.lex

gcc –o DATE y.tab.c lex.yy.c –ly –lfl

Run the program DATE and enter interactively a string other than of the format 12/31/1998 and observe the error message. Otherwise, it displays the message OK.

As mentioned earlier, Yacc uses symbolic tokens. In the above Yacc specification file we have declared symbolic Tokens such as NUMBER, NL, and SLASH using %token

declaration. When we run Yacc command on this specification file it generates the file y.tab.h (an example is shown above) in which for all these token a number assigned starting from 257.

Also, observe that these symbolic tokens are used in lex specification file, in which when a regular espression match occurs, yylex() returns this symbolic token. That is, the lexical analyser, yylex() generated by lex command takes responsibility of reading input stream and recognizing low level structures (regular expressins) historically called as terminal symbols and communicates these tokens to the parser which in turn recognizes the nonterminals, i.e grmmatical structures.

Tokens also will have assigned values during the scanning process and the same is assigned to variable yylval which is defined internally by Yacc.

Example 1

This example is to develop parser which recognizes strings of form $a^n b^n$, where n>=1.

Yacc Specification File (ab.y)

```
%token A B
%%
start: anbn '\n' {return 0;}
anbn :  A B
    | A anbn B
    ;
%%
#include"lex.yy.c"
```

Lex Specification File (ab.lex)

```
%%
a     return(A);
b     return(B);
.     return(yytext[0]);
\n    return('\n');
```

To create parser which recognizes strings of for $a^n b^n$, run the following commands

```
lex  ab.lex
yacc –dv ab.y
gcc –o anbn y.tab.c  lex.yy.c  -ly –lfl
```

Run the resulting program (anbn) and check by giving pattern such as:

aabb or aaaabbbb.

If we do not give matching input it gives error message “Syntax Error”. Otherwise it displays nothing.

Example 2

The following is a little modified version of the above program with error checking and better user interface when match occurs.

Yacc Specification File (ab1.y)

```
%token A B
%%
start: anbn '\n' { printf("  is in anbn\n");                  return 0; }
anbn:  A B
    | A anbn B
    ;
%%
#include "lex.yy.c"

yyerror(s)
char *s;
{
  printf("%s, it is not in anbn\n", s);
}
```

Lex Specification File(ab1.lex)

```
%%
```

```
a     return(A);
b     return(B);
.     return(yytext[0]);
\n    return('\n');
```

To create parser, run the following commands

```
lex  ab1.lex
yacc –dv ab1.y
gcc –o anbn1 y.tab.c  lex.yy.c  -ly –lfl
```

Run the resulting program (anbn1) and check by giving pattern such as:

aabb   or aaaabbbb

Example 3

This example is also to explain simple yacc example.

Yacc Specification File(two.y)

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

place   :       BELL
        ;
%%
#include "lex.yy.c"
```

Lex Specification File (two.lex)

```
%%
"ding"  return (DING);
"dong" return (DONG);
"bell"   return (BELL);
```

To create parser which accepts “ding dong bell”, run the following commands

```
lex  two.lex
yacc –dv two.y
gcc –o two y.tab.c  lex.yy.c  -ly –lfl
```

After creating parser if we give input “ding dong bell” it accepts otherwise rejects, i.e it gives an error message “Syntax Error”.

Example 4

This Yacc specification and lex specification program’s are for testing balanced parentheses.

Yacc Specification File (bp.y)

```
%{
#include <ctype.h>
#include <stdio.h>
#include "y.tab.h"
extern int yydebug;
%}
%token          OPEN  CLOSE
%%
lines    : s '\n' {printf("OK\n"); }
         ;
s        :
         | OPEN s CLOSE  s
      ;
%%
```

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

int yywrap(){return 1; }
int main(void) {
yydebug=1;
return yyparse();}
```

Lex Specification File (bp.lex)

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

%%
[ \t]     { /* skip blanks and tabs */ }
"("       return OPEN;
")"       return CLOSE;
\n|.      { return yytext[0]; }
```

To create parser which accepts strings having balanced parantheses, run the following commands

```
lex  bp.lex
yacc –dv bp.y
gcc –o bp y.tab.c  lex.yy.c  -ly –lfl
```

Test the generated parser (bp) by giving the following input.

(()) or (()(()()))

Example 5

The following Yacc and Lex specification files are used to generate parser which recognizes arithemetic expressions involving + and -.

Yacc Specification File (ath.y)

```
%{
%}

%token NAME NUMBER EQUL PLUS MINUS

%%

Stmt  :  NAME EQUL exp
      |  exp
      ;

exp   :  NUMBER PLUS NUMBER
      |  NUMBER MINUS NUMBER
      |  NUMBER MINUS exp
      |  NUMBER PLUS exp
      ;

%%

void yyerror(char * s)
{

  printf ( "%s\n", s);

}
int yywrap(){return 1;}

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

Lex Specification File (ath.lex)

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

%%
[a-zA-Z\_][a-zA-Z\_0-9]*      return NAME;
[0-9]+  return NUMBER;
"+"       return PLUS;
"-"       return MINUS;
"="       return EQUL;
```

To create parser which accepts arithemetic expressions with +, - operators, run the following commands

```
lex  ath.lex
yacc –dv ath.y
gcc –o ath y.tab.c  lex.yy.c  -ly –lfl
```

Run the command “ath” and enter the following expressions

1+2+3-2 or 1-2-3-5+4

Example 6

This Yacc and Lex specification files can be used to generate a tiny language which can be used (simulation only!!) to control a thermostat. It accepts commands such as “on”, “off” etc.

Yacc Specification File (thermo.y)

```
%{
#include <stdio.h>
#include <string.h>
void yyerror(const char *str)
{
```

```
	fprintf(stderr,"error: %s\n",str);
}
int yywrap()
{
	return 1;
}

main()
{
	yyparse();
}

%}

%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE
%%
commands	:
		| commands command
	;

command:
	heat_switch
	|
	target_set
	;

heat_switch:
	TOKHEAT STATE
	{
		printf("\tHeat turned on or off\n");
	}
	;

target_set:
	TOKTARGET TOKTEMPERATURE NUMBER
	{
		printf("\tTemperature set\n");
	}
```

```
;
```

Lex Specification File (thermo.lex)

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
 [0-9]+                return NUMBER;
heat                   return TOKHEAT;
on|off                 return STATE;
target                 return TOKTARGET;
temperature            return TOKTEMPERATURE;
\n                     /* ignore end of line */;
[ \t]+                 /* ignore whitespace */;
%%
```

To create parser which accepts the above language, run the following commands

```
lex  thermo.lex
yacc –dv thermo.y
gcc –o thermo y.tab.c  lex.yy.c  -ly –lfl
```

Example 7

The following Yacc and Lex specification files are used to recognize addresses which are in a specific format only.

Yacc Specification File (add.y)

```
%{
#include <stdio.h>
void yyerror(const char *str)
{

        fprintf(stderr,"error: %s\n",str);
}
```

```
int yywrap()
{
	return 1;
}

%}

%token CAPSTRING CAPLETTER NUMBER STATE ZIPPLUSFOUR COMMA
HASH DOT NEWLINE DOORNO

%%

sentence: firstline secondline thirdline { printf("Have a valid address.\n"); }
	;

firstline: firstname surname NEWLINE
	| firstname middlename surname NEWLINE
	;

secondline: DOORNO street NEWLINE
	| DOORNO street HASH NUMBER NEWLINE
	;

thirdline: city STATE COMMA zip NEWLINE
	;

firstname: CAPSTRING ;

middlename: CAPLETTER DOT ;

surname: CAPSTRING ;

street: CAPSTRING
	| CAPSTRING street
	;
```

```
city: CAPSTRING
   | CAPSTRING city
   ;

zip: ZIPPLUSFOUR
  ;

%%


int main( void ) {

     yyparse();
return 0;
}
```

Lex Specification File (add.lex)

```
%{
#include "y.tab.h"
%}
%%
 [\t ]+              /* ignore whitespace */;
 [A-Z][a-z]+     { return CAPSTRING; };
 [A-Z][A-Z]     {return STATE; }
 [A-Z]              {return CAPLETTER; }
 [0-9]+                      {return NUMBER; }
 [0-9]+-[0-9]+-[0-9]+            {return DOORNO; }
 PIN-[0-9][0-9][0-9][0-9][0-9][0-9]              {return ZIPPLUSFOUR; }

\,                  {return COMMA; }

#                   {return HASH; }

\.                  {return DOT; }

\n                  {return NEWLINE; }
```

%%

To create parser which accepts addresses, run the following commands

```
lex  add.lex
yacc –dv add.y
gcc –o add y.tab.c  lex.yy.c  -ly –lfl
```

Sample input which is accepted by the parser developed is:

```
Ravi Teja
12-33-33 First Street
Visakhapatnam AP, PIN-121212
```

1.2.3 Use of Pseudovariables

While writing actions for each grammar rule, we can make use of pseudovariables supported by the Yacc. As mentioned earlier, when a grammar rule is matched, every symbol in the rule will have a value which is returned by yylex(). Usually, this is assumed to be integer unless redefined by the user. These values are maintained as a seperate stack known as value stack in addition to parse stack which matains the symbols. The variable $$ represents the value of nonterminal and $1, $2, .. as the values of symbols on the right hand side of the nonterminal(rule). Thus, in the following example,

```
expr     : expr PLUS term        { $$ = $1 + $3; }
```

$$ refers to expression value and $1 and $3 refers to both the operands. That is sum of these operands are assigned to the expression.

In order to tell YACC about the new type of yylval, we add this line to the header of our YACC grammar:

```
#define YYSTYPE char *
extern YYSTYPE yylval;
```

Also, we can use the %union in yacc file such that we can declare that yylval to be a union of an integer, a string pointer, and a character.

```
%union { int integer_value; char *string_value; char op_value; }
```

Now, we can declare both terminals and nonterminals as either intger or char type using the following manner.

```
%token <int> OPRND1
%token<char> OPR1
%token<integer> exp
```

Run the Ycc command and check how the union is declared in the y.tab.h file.

Example 8

This Yacc specification file used to develop calculator which accepts single digit operands. Also, here we are not using any lexical specification file. The necessary lexical analysis program (yylex()) is written directly.

Yacc specification File (calc.y)

```
%{
#include<stdio.h>
#include<stdlib.h>
%}

%token PLUS MINUS MUL DIV NEWLINE RPAR LPAR
%token NUMBER

/* grammar rules & actions section */

%%

/* These two rules are for reading expressions from the keyboard */

lines   : lines line
        |
        ;
line    : expr NEWLINE              { printf("%d\n> ", $1); }
        | NEWLINE           { printf ("> "); }
```

```
        ;

/* Grammar rules for integer expressions evaluation */

expr    : expr PLUS term        { $$ = $1 + $3; }
        | expr MINUS term       { $$ = $1 - $3; }
        | term                  { $$ = $1; } /* default action */
        ;
term    : term MUL factor       { $$ = $1 * $3; }
        | term DIV factor       { if ($3 == 0)
                                yyerror("divide by zero");
                                  else
                                    $$ = $1 / $3;
                                }
        | factor            { $$ = $1; } /* default action */
        ;
factor  : LPAR expr RPAR    { $$ = $2; }
        | NUMBER               { $$ = $1; } /* default action */
        ;

%%


yylex() {
      /* My lexer */
         int c;
          do {
          c=getchar();
          switch (c) {
        case '0': case '1': case '2': case '3': case '4': case '5': case '6':
           case '7': case '8': case '9':
              yylval= c - '0';
              return NUMBER;
           case '+':  return PLUS;
           case '-':  return MINUS;
           case '*':  return MUL;
           case '/':  return DIV;
           case '(':  return LPAR;
```

```
            case ')':  return RPAR;
            case '\n':  return NEWLINE;
        }
   } while (c!= EOF);

          return(EOF);
}

main() {
          printf("> ");
          yyparse();
}
```

To generate the calculator program (executable file), run the following commands.

```
yacc –dv calc.y
gcc –o calc y.tab.c –ly
```

Example 9

This Yacc and Lex specification programs are used to generate a calculator which is flexible than the previous one. It accepts, integer and float type arguments.

Yacc Specification File (calculator.y)

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

%union{ double   real; /* real value */
       int   integer; /* integer value */
      }

%token <real> REAL
%token <integer> INTEGER
%token PLUS MINUS TIMES DIVIDE LP RP NL
```

```
%type <real> rexpr
%type <integer> iexpr

%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS

%%

lines: /* nothing */
    | lines line
    ;

line:  NL
    | iexpr NL
      { printf("%d) %d\n", lineno, $1);}
    | rexpr NL
      { printf("%d) %15.8lf\n", lineno, $1);}
    ;

iexpr: INTEGER
    | iexpr PLUS iexpr
      { $$ = $1 + $3;}
    | iexpr MINUS iexpr
      { $$ = $1 - $3;}
    | iexpr TIMES iexpr
      { $$ = $1 * $3;}
    | iexpr DIVIDE iexpr
      { if($3) $$ = $1 / $3;
        else { fprintf(stderr, "divide by zero\n");
               yyerror();
             }
      }
    | MINUS iexpr %prec UMINUS
      { $$ = - $2;}
    | LP iexpr RP
      { $$ = $2;}
    ;
```

```
rexpr: REAL
    | rexpr PLUS rexpr
      { $$ = $1 + $3;}
    | rexpr MINUS rexpr
      { $$ = $1 - $3;}
    | rexpr TIMES rexpr
      { $$ = $1 * $3;}
    | rexpr DIVIDE rexpr
      { if($3) $$ = $1 / $3;
        else { fprintf(stderr, "divide by zero\n");
               yyerror();
                 }
      }
    | MINUS rexpr %prec UMINUS
      { $$ = - $2;}
    | LP rexpr RP
      { $$ = $2;}
    | iexpr PLUS rexpr
      { $$ = (double)$1 + $3;}
    | iexpr MINUS rexpr
      { $$ = (double)$1 - $3;}
    | iexpr TIMES rexpr
      { $$ = (double)$1 * $3;}
    | iexpr DIVIDE rexpr
      { if($3) $$ = (double)$1 / $3;
        else { fprintf(stderr, "divide by zero\n");
               yyerror();
                 }
      }
    | rexpr PLUS iexpr
      { $$ = $1 + (double)$3;}
    | rexpr MINUS iexpr
      { $$ = $1 - (double)$3;}
    | rexpr TIMES iexpr
      { $$ = $1 * (double)$3;}
    | rexpr DIVIDE iexpr
      { if($3) $$ = $1 / (double)$3;
```

```
        else { fprintf(stderr, "divide by zero\n");
               yyerror();
                }
      }
    ;

%%
#include "lex.yy.c"
int lineno;
```

Lex Specification File (calculator.lex)

```
integer      [0-9]+
dreal        ([0-9]*\.[0-9]+)
ereal        ([0-9]*\.[0-9]+[Ee][+-]?[0-9]+)
real         {dreal}|{ereal}
nl           \n

%%
[ \t]        ;
{integer}    { sscanf(yytext, "%d", &yylval.integer);
               return INTEGER;
             }
{real}       { sscanf(yytext, "%lf", &yylval.real);
               return REAL;
             }
\+           { return PLUS;}
\-           { return MINUS;}
\*           { return TIMES;}
\/           { return DIVIDE;}
\(           { return LP;}
\)           { return RP;}

{nl}         { extern int lineno; lineno++;
               return NL;
             }
.            { return yytext[0]; }
```

To create parser which accepts arithemetic expressions with +, - operators, run the following commands

```
lex  calculator.lex
yacc –dv calculator.y
gcc –o calculator y.tab.c  lex.yy.c  -ly –lfl
```

Example 10

The following Yacc and Lex specification files are used to generate a program which identify the number of words in the given input file.

Yacc Specification File (words.y)

```
%{
#include<stdlib.h>
#include<string.h>
int yylex();
#include "words.h"
int nwords=0;
#define MAXWORDS 100
char * words[MAXWORDS];
%}

%token WORD
%%
text     : ;
         | text WORD; {
                        if($2<0) printf("New Word\n");
                        else  printf("Matched\n");
                    }
%%

int find_word(char *x)
{
int i;
```

```
for(i=0;i<nwords;i++) if(strcmp(x,words[i])==0) return i;

words[nwords++]=strdup(x);
return -1;
}

int main()
{
yyparse();
printf("No of Words=%d\n", nwords);
}

void yyerror(char *a)
{
}

int yywrap()
{
return 1;
}
```

Lex Specification File (words.lex)

```
%{
#include "y.tab.h"
int find_word(char *);
extern int yylval;
%}

%%

[a-zA-Z]+       {yylval=find_word(yytext); return WORD;}
.|\n       ;
%%
```

To create parser which counts the number of words in a given file, run the following commands

```
lex  words.lex
yacc –dv words.y
gcc –o words y.tab.c  lex.yy.c  -ly –lfl
```

Tracing the execution of a Yacc generated parser can be done by including the following lines to the Yacc specification file and while compiling give –DYYDEBUG option.

```
extern in yydebug;
yydebug=1;
```

1.3 Conclusions

This chapter discusses about the use of Lex and Yacc libraries for developing lexical analysis programs, file processing utilities and parsers.