**Practical – 1**

**Introduction to LEX**

**The Lex**

* A lexical analyzer takes input streams and divides into tokens. This division into units is known as lexical analysis. Lex takes set of rules for valid tokens and produce C program which we call lexical analyzer or lexer that can identify these tokens.
* Lex is a lexical analyzer generator-a tool for programming that recognizes lexical patterns in the input with the help of Lex specifications.
* Lex is generally used in the manner as shown below.



* First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, the lex.l is run through the Lex compiler to produce a c program lex.yy.c. This program consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l, together with a standard routine that uses the table to recognize lexemes.
* The action associated with regular expression in lex.l is pieces of C code and are carried over directly to lex.yy.c.
* Finally lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

**Structure of Lex Program**

Declarations

%%

Rules Section

%%

User Code(Auxiliary) Section

**Definition Section**

* It contains different user defined Lex options used by the lexer. It also creates an environment for the execution of the Lex program.
* The definition section creates an environment for the lexer, which is a C code. This area of the Lex specification is separated by “ %{ “ , and it contains C statements, such as global declarations, commands, including library files and other declarations, which will be copied to the lexical analyzer(lex.yy.c) when it passed through the Lex tool.
* The definition section provides an environment for the Lex tool to convert the Lex specifications correctly and efficiently to a lexical analyzer. This section mainly contains declarations of simple name definitions to simplify the scanner specifications and declarations of start condition. The statement in this section will help the Lex rules to run efficiently.

**Example:**

%{

#include "calc.h"

#include <stdio.h>

#include <stdlib.h>

char name[10];

%}

/\* Regular expressions \*/

/\* ------------------- \*/

white [\t\n ]+

letter [A-Za-z]

digit [0-9]

identifier {letter}(\_|{letter}|{digit10})\*

**Rule Section**

* It contains the patterns and actions that specify the lex specifications. A pattern is in the form of a regular expression to match the largest possible string.
* Once the pattern is matched, the corresponding action part is invoked. The action part contains normal C language statements.
* They are enclosed within braces ( “{“ and “}”), if there is more than one statement then make these component statements into single block of statements.

**Example**

%%

{LETTER}({LETTER}| {DIGIT})\* {

printf(“\n It is a Identifier: %s \n”, yytext);

}

%%

* Always use braces to make the code clear, if the action has more than one statement or more than one line large.
* The lexer always tries to match the largest possible string, but when there are two possible rules that match the same length, the lexer uses the first rule in the Lex specification to invoke its corresponding action.

**User defined section**

* This section contains any valid C code. Lex copies the contents of this section into generated lexical analyzer as it is.

Example

/\* ……………User define Function………………..\*/

void display(int a)

{

………

………..

}

main()

{

yylex();

}

* Lex itself does not produce an executable program; instead, it translates the Lex specifications into a file containing a C subroutine called yylex().
* All the rules in the rule section will automatically be converted into C statements by the Lex tool and will be put under the function name of yylex().
* Whenever, we call the function yylex, C statements corresponding to the rules will be executed.
* That is we called the function yylex() in main function, even though we have not defined it anywhere in the program.

**Compiling and executing the Lex program**

%{

%}

%%

%%

main()

{

yylex();

}

Let above program be in file called practical.l. To create or generate a lexical analyzer we must enter the following command

**$ lex practical.l**

When, the above command is executed, Lex translates the Lex specification into a C source file called lex.yy.c, which is a lexical analyzer. Any lexical analyzer can be compiled using the following command

**$ cc lex.yy.c –ll**

This will compile the lexical analyzer, lex.yy.c, using any C compiler by linking it with Lex library using the extension –ll. After compilation, the output, by default, will write to “a.out” file.

The resulting program is executed using the following command

**$ ./a.out**

**or**

**$ ./a.out < filename**

**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 |

**Yytext and yyleng**

* 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.
* 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**, and its length in the global integer yyleng.
* The actioncorresponding to the matched pattern is then executed and then the remaining input is scanned for another match.
* If no match is found, then the default ruleis executed: the next character in the input is considered matched and copied to the standard output.
* yytext can be defined in two different ways: either as a character pointeror as a character array.
* We can control which definition lex uses by including one of the special directives `%pointer' or `%array' in the first (definitions) section of lex input.
* The default is `%pointer' and the advantage of using `%pointer' is substantially faster scanning and no buffer overflow when matching very large tokens (unless run out of dynamic memory).
* The disadvantage is that `input()' 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 we can then modify yytext to your heart's content, and calls to `unput()' do not destroy yytext.
* 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. We can change the size by simply

#define YYLMAX <constant number>.

* 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 runtime error results.

**Special Directives**

There are number of special directives which can be included within an action. Directives, like keywords in C, are those words whose meaning has been already predefined in Lex tool. Mainly we have three directives in Lex.

1. ECHO:- It copies yytext to the scanner’s output. That is, whatever token we have recently found will be copied to the output.
2. BEGIN:- The directive BEGIN followed by the name of the start symbol, places the scanner in the corresponding rules. Lex activates the rules using the directive BEGIN and a start condition.
3. REJECT:- It directs the scanner to proceed on to the "scanned best" rule which matched the input (or a prefix of the input). That is, as soon as REJECT statement is executed in the action part, the last letter, will be treated (or prefixed) from the recently matched token and will go ahead with the prefixed input for next best rule.

**Example**

%{

%}

%%

[a-z]+ {

printf(“\n String contains only lower case letters= ”);

ECHO;

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters= ”);

ECHO; REJECT;

}

%%

main()

{

yylex();

}

$ ./a.out

asDF

**Start Conditions**

* Start conditions are declared in the definition section of the Lex program using unintended lines beginning with either %s or %x, followed by a list of names called start symbols.
* A start condition rule is activated using the directives BEGIN. Until the next BEGIN action is executed, rules with the given start conditions will be active and those with other conditions will be inactive.
* If a start condition is declared with %s, then it is called an inclusive start condition. If the start condition is declared as inclusive, then all rules without any start condition and rules with corresponding start condition will be active.
* If start condition is declared with %x, then it is called an exclusive start condition. If start condition is declared as exclusive, then an only rule that is/are qualified with the start condition will be active.

**Example**

%{

%}

%s SM SMBG

%%

# BEGIN(SM);

## BEGIN(SMBG);

[0-9]+ {

printf(“\n It’s a Digit ”);

}

<SMBG>[A-Z]+ {

printf(“\n String contains upper case letters ”);

}

<SM>. {

printf(“\n Exiting from # start condition ”);

BEGIN(INITIAL);

}

<SM, SMBG>[a-z]+ {

printf(“\n String contains lower case letters ”);

}

<SMBG>.+ {

printf(“\n Exiting from ## start condition ”);

}

.+ {

printf(“\n No action to execute ”);

}

%%

main()

{

printf(“\n Enter # when u r executing digits and small case letter strings ”);

printf(“\n Enter ## when u r executing only upper and small case letter strings ”);

yylex();

}

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

%{

%}

%%

%%

main()

{

yylex();

return 0;

}

**3. LEX program to show the message when enter key is pressed**

%{

%}

%%

[\n] {

printf(“\n Hi…..Good….Morning\n”);

}

%%

main()

{

yylex();

return 0;

}

**4. LEX program to print the name of user with message when enter key is pressed**

%{

#include<stdio.h>

char name[20];

%}

%%

[\n] {

printf(“\n Hi…..Good….Morning\n”);

}

%%

main()

{

char ch;

do

{

printf(“\n Enter your name\n”);

scanf(“%s”, &name);

yylex();

printf(“\n Press any key to continue(Y/y):”);

scanf(“%c”, &ch);

}while((ch= =’Y’) || (ch= =’y’))

}

**5. LEX program to check whether the string is in smallcase letter,uppercase letter or contains mixed letter**

%{

%}

%%

[a-z]+ {

printf(“\n String contains only lower case letters ”);

}

[A-Z]+ {

printf(“\n String contains only upper case letters ”);

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters”);

}

%%

main()

{

yylex();

}

**6. LEX program to print the name of the user with message when the enter key is pressed using the function**

%{

void display(char\*)

%}

%%

[\n] {

char name[20];

printf(“\n Enter your name\n”);

scanf(“%s”, &name);

display(&name[20]);

return;

}

%%

void display(char\* in)

{

printf(“\n Hi…..Good….Morning\n”);

}

main()

{

printf(“\n Press enter key…..\n”);

yylex();

}

**7. LEX program to check whether the given word is vowel or not using the function.**

%{

void display(int)

%}

%%

[a|e|i|o|u][a-zA-Z]+ {

int flag=1;

display(flag);

return;

}

.+ {

int flag=0;

display(flag);

return;

}

void display(int flag)

{

if(flag = =0)

printf(“\n The given word is vowel\n”);

else

printf(“\n The given word is not vowel\n”);

}

main()

{

printf(“\n Enter the word\n”);

yylex();

}

**8. LEX program which demonstrate the use of yymore function**

This function will output the yytext, when execution of the action part of any rule that invoked yymore() ends.

**Example**

%{

%}

%%

[a-z]+ {

printf(“\n String contains only lower case letters= ”);

ECHO; printf(“ Beginning of the first yymore”);

yymore(); printf(“ End of the first yymore”);

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters= ”);

ECHO; printf(“ Beginning of the second yymore”);

yymore(); printf(“ End of the second yymore”);

}

%%

main()

{

yylex();

}

$ ./a.out

Good Morning

**9. LEX program which demonstrate the use of yyless function**

This function yyless(n) returns all characters, except the first n characters of the current tokens, back to the input stream, where they will be re-scanned when the scanner looks for the next match.

**Example**

%{

%}

%%

[a-z]+ {

printf(“\n The string is= ”);

ECHO;

yyless(2);

printf(“ The string after yyless= ”); ECHO;

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters= ”);

ECHO;

}

%%

main()

{

yylex();

}

$ ./a.out

Nice morning

**10. LEX program which demonstrate the use of unput function**

This function unput(a) puts or returns character a back into the input stream and it will be the nect character to be scanned.

**Example1**

%{

%}

%%

“un” {

printf(“\n The unput char = ”);

ECHO;

}

[a-z]+ {

printf(“\n String contains only lower case letters= ”);

ECHO; unput(‘n’); unput(‘u’);

printf(“\n The string after unput= ”); ECHO;

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters= ”);

ECHO;

}

%%

main()

{

yylex();

}

$ ./a.out

good Day

**Example2**

%{

#define YYLMAX 10

%}

%array yytext

%%

“un” {

printf(“\n The unput char = ”);

ECHO;

}

[a-z]+ {

printf(“\n String contains only lower case letters= ”);

ECHO; unput(‘n’);

printf(“\n The string after first unput= ”); ECHO;

unput(‘u’);

printf(“\n The string after second unput= ”); ECHO;

}

[a-zA-Z]+ {

printf(“\n String contains both lower & upper case letters= ”);

ECHO;

}

%%

main()

{

yylex();

}

$ ./a.out

good Day

**11. LEX program which demonstrate the use of input function**

This function reads the next character from the input stream. The read character will not be made available to the scanner.

**Example**

%{

%}

%%

[a-zA-Z0-9]+ {

printf(“\n String contains mixed letters= ”);

ECHO;

}

“/\*” {

printf(“\n The comment begins ”);

char c;

while(( c=input() )!= ‘\*’);

if ((c=input()) = = ’/’)

printf(“\n The comment ends ”);

%%

main()

{

yylex();

}

$ ./a.out

This program is written by /\* Name \*/

**12. LEX program to recognizes the keyword if, begin and identifier which is defined as any string starts with letter and followed by letter or digit.**

%{

#include<stdio.h>

%}

Letter[a-zA-Z]

Digit[0-9]

%%

begin {

printf(“\n It is a keyword:%s\n”,yytext);

}

if {

printf(“\n It is a keyword:%s\n”,yytext);

}

{Letter}({Letter} || {Digit})\* {

printf(“\n It is a Identifier:%s\n”,yytext);

}

%%

main()

{

yylex();

}

**13**. **LEX program to recognizes the keyword if, begin and identifier which is defined as any string starts with letter and followed by letter or digit and count the number of identifiers, keywords and operators encountered in the input.**

%{

Int k=0,op=0,id=0;

%}

Letter[a-zA-Z]

Digit[0-9]

%%

(begin|if|else|while|do|then) {

k++;

}

[+\*-/<>=] {

op++;

}

(<=|>=|!=) {

op++;

}

[. ; \ .] ;

{Letter}({Letter} || {Digit})\* {

id++;

}

%%

main()

{

yylex();

printf(“\n Number od ID’s=%d\t, Keywords=%d\t, Operators=%d\t”, id,k,op);

}

**14. Program using LEX to count the number of characters, words, spaces and lines in a given input file.**

%{

int ch=0, bl=0, ln=0, wr=0;

%}

%%

[\n] {ln++;wr++;}

[\t] {bl++;wr++;}

[" "] {bl++;wr++;}

[^\n\t] {ch++;}

%%

int main()

{

FILE \*fp;

char file[10];

printf("Enter the filename: ");

scanf("%s", file);

yyin=fp;

yylex();

printf("Character=%d\nBlank=%d\nLines=%d\nWords=%d", ch, bl, ln, wr);

return 0;

}

**15. Program using LEX to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.**

%{

#include<stdio.h>

int a=0,s=0,m=0,d=0,ob=0,cb=0;

int flaga=0, flags=0, flagm=0, flagd=0;

%}

id [a-zA-Z]+

%%

{id} {printf("\n %s is an identifier\n",yytext);}

[+] {a++;flaga=1;}

[-] {s++;flags=1;}

[\*] {m++;flagm=1;}

[/] {d++;flagd=1;}

[(] {ob++;}

[)] {cb++;}

%%

int main()

{

printf("Enter the expression\n");

yylex();

if(ob-cb==0)

{

printf("Valid expression\n");

}

else

{

printf("Invalid expression");

}

printf("\nAdd=%d\nSub=%d\nMul=%d\nDiv=%d\n",a,s,m,d);

printf("Operators are: \n");

if(flaga)

printf("+\n");

if(flags)

printf("-\n");

if(flagm)

printf("\*\n");

if(flagd)

printf("/\n");

return 0;

}

**16. LEX program to check whether the parenthesis in the statement is missing or not**

%{

int flag=0,ln=1;

%}

%%

“(“ {

flag++;

}

“)” {

flag--;

}

[\n] {

if(flag= =0)

printf(”\n the statement in the line %d has no parenthesis is missing\n”,ln);

else

printf(“\n Error…..in the line:%d”,ln);

if(flag < 0)

printf(”\n It has missed ( parenthesis or extra ) parenthesis \n”);

else if(flag > 0)

printf(”\n It has missed ) parenthesis or extra ( parenthesis \n”);

flag=0; ln++;

%%

main()

{

Char filename[20];

printf(“\n Enter the file name:”\n);

scanf(“%s”, filename)

yyin=fopen(filename,”r+”);

yylex();

}

**17. Lex program which replaces all the occurances of “rama” with “RAMA” and “sita” with “SITA”. It demonstrate the use of 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;

}

**18. Lex program 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;

}

**19.LEX program which removes all occurrences “ rama “ and “ sita “ in a given file**

%{

int count=0;

%}

%%

“rama”

“sita”

. ECHO ;

%%

main( )

{

yylex( );

}

**20. Lex program, to count all instances of she and he, including the instances of he that are included in she**.

%{

int count=0;

%}

%%

she {

count++;

REJECT;

}

he {

count++;

}

. ;

%%

main(int argc, char\* argv[ ])

{

Extern FILE\* yyin;

yyin = fopen(argv[1],”r”);

yylex();

printf(“ No of occurrences of he including he in she = %d\n”, count);

return 0;

}

**21. Lex 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 occurrences of other than words incl = %d\n”, nw);

return 0;

}

**22. Lex program which take string “abcd ” and print the following output**

**abcd**

**abc**

**ab**

**a**

%{

%}

%%

a | ab | abc | abcd {

printf(“%s\n”, yytext);

REJECT;

}

%%

main(int argc, char\* argv[ ])

{

Extern FILE\* yyin;

yyin = fopen(argv[1],”r”);

yylex();

return 0;

}

**23. Lex program that changes all numbers to hexadecimal in input file while ignoring all others.**

%}

%}

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;

}

**24. Lex program to recognize whether a given sentence is simple or compound**

%{

int flag =0;

%}

%%

( [aA][nN][dD] ) {

flag=1;

}

“or “ {

flag =1;

}

“nevertheless” {

flag =1;

}

“inspite” {

flag =1;

}

. ;

%%

main()

{

printf(“\n enter the sentence \n”);

yylex();

if ( flag = =0 )

printf”\n Sentence is simple”);

else

printf(“\n Sentence is compound”);

}

**25. LEX program to implement the positive closure of 10**

%{

%}

Digit[0-9]

%%

Digit+ {

printf(“\n It is a positive closure\n”);

}

.+ {

printf(“\n It is not a positive closure\n”);

}

main()

{

yylex();

}

**26. LEX program that accept the language L = { a n-1 b n+m, where n > =1 and m> =0 }**

%{

int check(int,char\*);

%}

%%

[ab]\* {

int leng=check(yyleng,yytext);

if(leng==1)

printf("\n accepted\n");

else

printf("\n not accepted\n");

}

%%

main()

{

yylex();

}

int check(int leng,char\*token)

{

int i, flag=0,ca=0,cb=0;

char ch;

for(i=1,ch=token[0]; ch=='a' && i<leng; i++)

{

ca++;

ch=token[i];

}

for(i=ca,ch=token[ca];ch=='b'&& i<leng; i++)

{

cb++;

ch=token[i];

}

if(((ca < cb) && (ca+cb)==leng))

flag=1;

return(flag);

}

**27. LEX program that accept the language L = { 1 n-1 0 n, where n > =1 }**

%{

int check(int,char\*);

%}

%%

[ab]\* {

int leng=check(yyleng,yytext);

if(leng==1)

printf("\n accepted\n");

else

printf("\n not accepted\n");

}

%%

main()

{

yylex();

}

int check(int leng,char\*token)

{

int i, flag=0,c0=0,c1=0;

char ch;

for(i=1,ch=token[0]; ch=='1' && i<leng; i++)

{

c1++;

ch=token[i];

}

for(i=c1;ch=='0'&& i<leng; i++)

{

C0++;

ch=token[i];

}

if(((c0 + 1 ) = = c1) && (c0 + c1) = =leng))

flag=1;

return(flag);

}

**28. LEX program that implement the simple desktop calculator**

%{

flaot op1=0,op2=0,ans=0;

char oper;

int f1=0,f2=0;

void eval();

%}

Digit [0-9]

Num {Digit}+

Op [+/\\*-]

%%

{Num} {

if (f1 = = 0)

{

Op1=atof(yytext);

f1=1;

}

else if (f2 = = -1)

{

Op2=atof(yytext);

f2=1;

}

if(( f1 = =1) && (f2= =1))

eval();

}

{Op} {

oper = (char)\* yytext;

f2 = -1;

}

[\n] {

if(( f1 = =1) && (f2 = =1))

eval();

f1=0; f2=0;

}

%%

void eval()

{

f1=0; f2=0;

switch(oper)

{

case ‘+’:ans=op1 + op2;

break;

case ‘-’:ans=op1 - op2;

break;

case ‘\*’:ans=op1 \* op2;

break;

case ‘/’:ans=op1 / op2;

break;

default:

printf(“\n program is not supporting the %c”, oper);

break;

}

printf(“\n the answer is = %f”, ans);

}

**Practical - 29**

**Introduction to Yacc**

**Introduction**

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.

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. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

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

month\_name : 'J' 'a' 'n' ;

month\_name : 'F' 'e' 'b' ;

……….

month\_name : 'D' 'e' 'c' ;

might be used in the 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 realively 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.

**Basic Specifications :**

Names refer to either tokens or non-terminal symbols. Yacc requires token names to be declared as such. In addition, 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 and PL/I.

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

A : BODY ;

A represents a non-terminal 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 non-terminal symbols.

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

'\n' newline

'\r' return

'\'' single quote ``'''

'\\' backslash ``\''

'\t' tab

'\b' backspace

'\f' form feed

'\xxx' ``xxx'' in octal

For a number of technical reasons, the 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 non-terminal 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. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminal 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 end-marker. If the tokens up to, but not including, the end-marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the end-marker is seen in any other context, it is an error.

**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 non-terminal 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.

**Translating, Compiling and Executing A Yacc Program**

The Lex program file consists of Lex specification and should be named <file name>.l and the Yacc program consists of Yacc sepecification and should be named <file name>.y. following command may be issued to generate the parser

Lex <file name>.l

Yacc –d <file name>.y

cc lex.yy.c y.tab.c –ll

./a.out

Yacc reads the grammar description in <file name>.yand generates a parser, function yyparse, in file y.tab.c . the –d option causes yacc to generate the definitions for tokens that are declared in the <file name>.y and palce them in file y.tab.h. Lex reads the pattern descriptions in <file name>.l, includes file y.tab.h, and generates a lexical analyzer, function yylex, in the file lex.yy.c

Finally, the lexer and the parser are compiled and linked (-ll) together to form the output file, a.out(by default).

The execution of the parser begins from the main function, which will be ultimately call yyparse() to run the parser. Function yyparse() automatically calls yylex() whenever it is in need of token .

**Lexical Analyzer for YACC**

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.

When Yacc generates, the parser(by default y.tab.c, which is C file), it will assign token numbers for all the tokens defined in Yacc program.Token numbers will be assigned using”#define”and will be copied, by default, to y.tab.h file. The lexical analyzer will reasd from this file or any furthe use.

**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. You specify as disambiguating rules 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 the **yacc** keywords **%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. **+** and **-** are left associative and have lower precedence than *****and */*, which are also left associative. The keyword **%right** is used to describe right associative operators. The keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. That is, because

A .LT. B .LT. C

is invalid in FORTRAN, **.LT.** 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) ) )

in order to achieve the correct precedence of operators. 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 minus.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword **%prec** changes the precedence level associated with a particular grammar rule. **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

%left '+' '-'

%left '*' '/'

%%

expr : expr '+' expr

| expr '-' expr

| expr '*' expr

| expr '/' expr

| '-' expr %prec '*'

| NAME

;

might be used to give unary minus the same precedence as multiplication.

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

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. 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** or **shift**-**reduce** conflict, and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given in the preceding 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 precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; 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.

**The yyerror() Function**

The yyerror function is called when Yacc encounters an invalid synatx. Whenver an invalid syntax finds error, it will move to already predefined error state. Moving to error state maens shifting (shift/reduce) to error, which is areserved token name for error handling.that is, any move to error state will cause to call function yyerror.the yyerror() is passed a single string of type char\* as argument. The basic yyerror() function is like this:

yyerror(char\* err)

{

fprintf(stderr,”%s\n”,err);

}

The above function just prints the error message when we call the function by passsing the argument.

**30. Program to recognize a valid arithmetic expression that uses operators +, -, \* and /.**

LEX

%{

#include"y.tab.h"

extern yylval;

%}

%%

[0-9]+ {yylval=atoi(yytext); return NUMBER;}

[a-zA-Z]+ {return ID;}

[\t]+ ;

\n {return 0;}

. {return yytext[0];}

%%

YACC

%{

#include<stdio.h>

%}

%token NUMBER ID

%left '+' '-'

%left '\*' '/'

%%

expr: expr '+' expr

|expr '-' expr

|expr '\*' expr

|expr '/' expr

|'-'NUMBER

|'-'ID

|'('expr')'

|NUMBER

|ID

;

%%

main()

{

printf("Enter the expression\n");

yyparse();

printf("\nExpression is valid\n");

exit(0);

}

int yyerror(char \*s)

{

printf("\nExpression is invalid");

exit(0);

}

**31. Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.**

LEX

%{

#include"y.tab.h"

extern yylval;

%}

%%

[0-9]+ {yylval=atoi(yytext); return DIGIT;}

[a-zA-Z]+ {return LETTER;}

[\t] ;

\n return 0;

. {return yytext[0];}

%%

YACC

%{

#include<stdio.h>

%}

%token LETTER DIGIT

%%

variable: LETTER|LETTER rest

;

rest: LETTER rest

|DIGIT rest

|LETTER

|DIGIT

;

%%

main()

{

yyparse();

printf("The string is a valid variable\n");

}

int yyerror(char \*s)

{

printf("this is not a valid variable\n");

exit(0);

}

**32. Yacc program which identify the language L = Σ, where Σ = {1,0} if and only if the string starts with 10.**

**LEX File**

%{

#include”y.tab.h”

%}

%%

“0” {

return ZERO;

}

“1” {

return ONE;

}

[\n] {

return NL;

}

. ;

%%

**YACC File**

%{

%}

%token ONE ZERO NL

%%

str1: str2 nl {

}

;

Str2: ONE ZERO str3 {

}

| ONE ZERO {

}

;

Str3: ZERO str3 {

}

| ONE str3 {

}

| ONE {

}

| ZERO {

}

;

nl: NL {

printf(“\n The string is matched\n”);

return;

}

;

%%

main()

{

yyparse( );

}

void yyerror( )

{

printf(“\n Error……..Invalid String\n”);

return;

}

**33. Yacc program which accept the language L+, where the language L = AB and the set A = {10,11}, B= {000, 110}**

**LEX File**

%{

#include”y.tab.h”

%}

%%

“0” {

return ZERO;

}

“1” {

return ONE;

}

[\n] {

return NL;

}

. ;

%%

**YACC File**

%{

%}

%token ONE ZERO NL

%%

str1: str2 str3 str4 nl {

}

| str2 str3 nl {

}

;

Str2: ONE ZERO {

}

| ONE ONE {

}

;

Str3: ZERO ZERO ZERO {

}

| ONE ONE ZERO {

}

;

Str4: str2 str3 str4 {

}

| str2 str3 {

}

;

nl: NL {

printf(“\n The string is matched\n”);

return;

}

;

%%

main()

{

yyparse( );

}

void yyerror( )

{

printf(“\n Error……..Invalid String\n”);

return;

}

**34. Yacc program which accept the language L = { 1n 0n; where n > 0}**

**LEX File**

%{

#include”y.tab.h”

%}

%%

“0” {

return ZERO;

}

“1” {

return ONE;

}

[\n] {

return\* yytext;

}

. ;

%%

**YACC File**

%{

int count=0;

%}

%token ONE ZERO

%%

stmt1: stmt2 ’\n’ {

dispaly(count);

return;

}

;

stmt2: ZERO stmt3 {

count--;

}

| ZERO {

count--;

}

| ONE error {

yyerror();

return;

}

;

%%

main()

{

yyparse( );

}

void dispaly(int count)

{

if(count = = 0)

printf(“\n The string is matched\n”);

else

printf(“\n The string is not matched\n”);

count=0;

return;

}

void yyerror( )

{

printf(“\n Error……..Invalid Input\n”);

count=0;

return;

}

**35. Yacc program that read from the input file( where a c program will be the content) to check and identify all valid identifiers**

**LEX File**

%{

#include”y.tab.h”

int cline=1;

%}

Digit [0-9]

Letter [a-zA-Z]

Datatype int | char | float | long | double | signed| unsigned

Ident {Letter}({Letter}| {Digit})\*

%%

{Datatype} {

return TYPE;

}

[Ident] {

return IDEN;

}

[ ; ] {

return COLE ;

}

[ , ] {

return COMMA ;

}

[ \n] {

Cline++ ;

}

. ;

%%

**YACC File**

%{

#include<stdio.h>

Extern FILE\* yyin;

int cident=0;

%}

%token TYPE IDEN COLE COMMA

%%

stmt1: {

}

| stmt1 TYPE stmt2 {

printf(“\n No : of Variables”);

printf(“ in the line No: %d”, cline);

printf(“is : %d”, cident );

cident=0;

}

| stmt1 error {

}

;

stmt2: IDEN COMMA stmt2 {

cident++;

}

| IDET COLE {

cident++;

}

;

%%

main ( int argc, char\* argv[])

{

yyin=fopen(argv[1], “r+”);

yyparse( );

}

void yyerror( )

{

}

**Input File**

void main ()

{

int a, b, c, d, e;

long p, q;

p = a + b;

long float r, s;

e = add(d, r);

signed int w, z;

w = z + e \* n;

}

**36. Yacc program that read from the input file ( where a c program will be the content) to check and identify all valid C if statement structure;**

**LEX File**

%{

#include”y.tab.h”

int cline=1;

%}

Digit [0-9]

Condition < | > | <= | >= | = = | !=

Identifier [a-zA-Z][a-zA-Z0-9]\*

%%

“ if “ {

return IF;

}

[ ( ) ] {

Return\* yytext;

}

{Identifier} {

return IDEN ;

}

{Condition} {

return CON ;

}

{Digit} {

return DIGIT ;

}

[\*+/ \ -] {

return OPER ;

}

[\n] {

cline++;

}

. ;

%%

**YACC File**

%{

#include<stdio.h>

extern FILE\* yyin;

extern int cline=0;

%}

%token IDEN CON IF DIGIT OPER

%%

stmt1: {

}

| stmt1 stmt2 {

}

| stmt1 error {

}

;

stmt2: IF ’(’ stmt3 COND stmt4 ’)’ {

printf(“\n Line No: %d”, cline);

printf(“contains an valid IF statement”);

}

| IF error {

printf(“\n Line No: %d”, cline);

printf(“contains an Invalid IF statement”);

}

;

stmt3: IDEN {

}

| expre {

}

;

stmt4: IDEN {

}

| DIGIT {

}

| expre {

}

;

expre: IDEN OPER IDEN { }

| IDEN OPER DIGIT

| DIGIT OPER IDEN

| DIGIT OPER DIGIT

| ’ ( ’ EXPRE ’ ) ’

;

%%

%%

main ()

{

char filename[20];

printf(“\n Enter the file name:”);

scanf(“%s”, filename);

yyin=fopen(filename, “r+”);

yyparse( );

return;

}

void yyerror( )

{

}

**Input File**

main ()

{

int a, b;

if( a!= b)

{

b=1;

}

If (b = =20)

{

a=a + b;

b = 50;

}

If ( a\*b > b+i)

{

b = a++;

}

**37. Yacc program to implement desktop calculator**

**LEX File**

%{

#include”y.tab.h”

extern int yyval;

%}

DIGIT [0-9]+

%%

{DIGIT} {

yyval= atoi(yytext);

return NUM;

}

[+ \* / \ ^ \ - ] {

return\* yytext;

}

. ;

%%

**YACC File**

%{

#include<stdio.h>

int yyval;

%}

%token NUM

%left ‘+’ ‘-‘

%left ‘\*’ ‘/’

%left ‘^’

%right UMINUS

%%

stmt: expr ‘\n’ {

printf(“\n The Result: %d\n”, $1);

return;

}

;

expr: expr ‘^’ expr {

$$=expFun( $1, $3);

}

| expr ‘/’ expr {

$$ = $1/ $3;

}

| expr ‘+’ expr {

$$ = $1+ $3;

}

| expr ‘-’ expr {

$$ = $1- $3;

}

| expr ‘\*’ expr {

$$ = $1\* $3;

}

| ‘-‘expr %prec UMINUS {

$$ = -$2;

}

| NUM {

$$ = $1;

}

;

%%

main ( )

{

printf(“\n Enter the Expression:”);

yyparse( );

}

int expFUN( int n, int m)

{

int i, exponent=1;

for(i=0; i<m; i++)

exponent = exponent\*n;

return exponent;

}

void yyerror ( )

{

printf(“\n Error……..Invalid expression”);

return;

}

**38. YACC program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using the grammar (anbn, n>= 0).**

LEX

%{

#include"y.tab.h"

%}

%%

[a] return A;

[b] return B;

%%

YACC

%{

#include<stdio.h>

%}

%token A B

%%

S:A S B

|

;

%%

main()

{

printf("Enter the string\n");

if(yyparse()==0)

{

printf("Valid\n");

}

}

yyerror(char \*s)

{

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

}

**39. Program to recognize the grammar (anb, n>= 10).**

LEX

%{

#include"y.tab.h"

%}

%%

[a] return A;

[b] return B;

%%

YACC

%{

#include<stdio.h>

%}

%token A B

%%

stat:exp B

;

exp:A A A A A A A A A exp1

;

exp1:A exp2

|A

|A A exp2

|A A A exp2

|A A A A exp2

;

exp2:A

;

%%

main()

{

printf("Enter the string\n");

if(yyparse()==0)

{

printf("Valid\n");

}

}

yyerror(char \*s)

{

printf("error\n");

}