Study Assignment

Assignment No 3: Write a program to recognize infix expression using LEX and YACC.

Objective: Understand the implementation of Lex and YACC Specification with simple & compound sentences.

Theory :

LEX

We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.

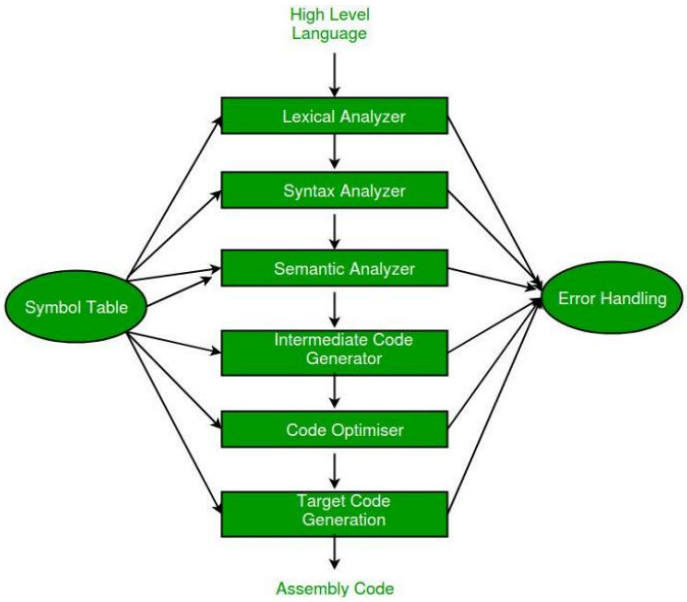


Fig:- Phases of Complier

The general format of Lex source is:

{definitions}

%%

{rules}

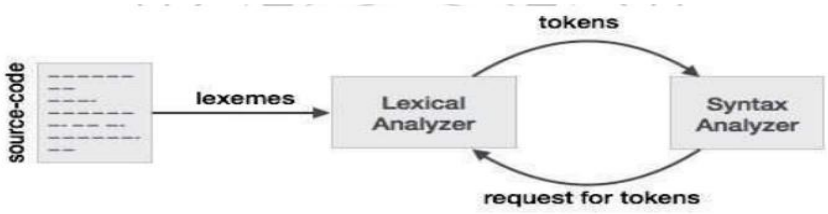
%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus %% (no definitions, no rules) which translates into a program which copies the input to the output unchanged.

Lexical Analysis:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any white space or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

Example of tokens:

▪ Type token (id, number, real, . . .)

▪ Punctuation tokens (IF, void, return, . . .)

▪ Alphabetic tokens (keywords)

Basic Functions of Lexical Analysis:

1.Tokenization i.e. Dividing the program into valid token.

2. Remove white space characters.

3. Remove comments.

4. It also provides help in generating error message by providing row number & column number.

YACC

Syntax Analyzer:

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.

Parse Tree: Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings. A parse tree is an entity which represents the structure of the derivation of a terminal string from some non-terminal (not necessarily the start symbol). The definition is as in the book. Key features to define are the root ∈ V and yield ∈ Σ* of each tree

- For each σ ∈ Σ, there is a tree with root σ and no children; its yield is σ
- For each rule A → ε, there is a tree with root A and one child ε; its yield is ε
- If t1, t2, ..., tn are parse trees with roots r1, r2, ..., rn and respective yields y1, y2, ..., yn, and A → r1r2...rn is a production, then there is a parse tree with root A whose children are t1, t2, ..., tn. Its root is A and its yield is y1y2...yn

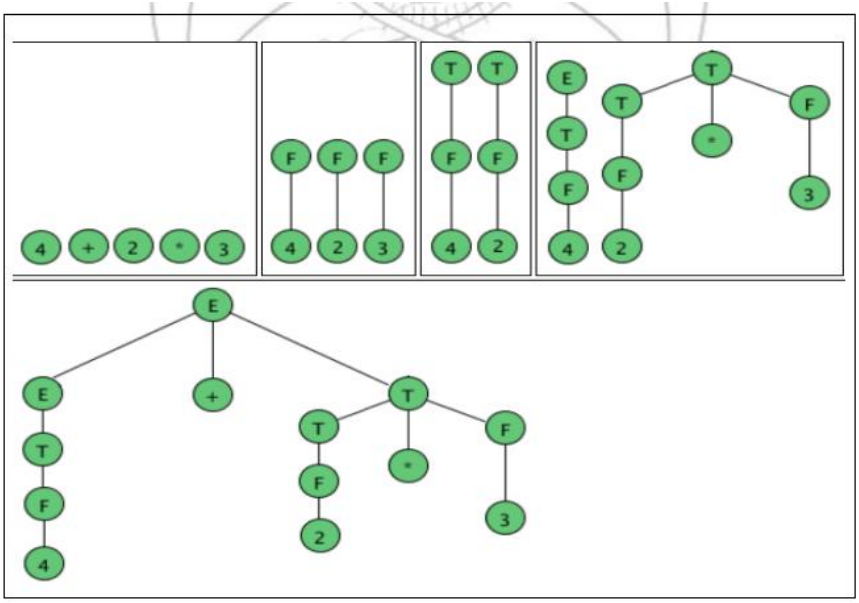
Observe that parse trees are constructed from bottom up, not top down. The actual construction of "adding children" should be made more precise, but we intuitively know what's going on. As an example, here are all the parse (sub) trees used to build the parse tree for the arithmetic expression 4 + 3 * 2 using the expression grammar

E → E + T | E - T | T

T → T * F | F

F → a | (E)

Where a represents an operand of some type, be it a number or variable. The trees are grouped by height.



Steps for Execution of Program :

1. Write lex program and save as filename. l and yacc program as filename. y.
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex filename. l to compile lex program.
4. type yacc filename. y to compile yacc program.
5. then c code will generate for both automatically.
6. type to compile both codes cc lex. yy. c y. tab. h -ll.
7. Type to run final code ./a. out.

Program

Infix.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ { yylval.dval=atoi(yytext); return NUMBER;}
[0-9]*"."[0-9]+ { yylval.dval=atof(yytext); return NUMBER;}
[a-zA-Z] { return LETTER; }
"+" { return PLUS;}
"-" { return MINUS;}
"*" { return MULTIPLY;}
"/" { return DIVIDE;}
"(" { return OPEN;}
")" { return CLOSE;}
"\n" { return ENTER;}
"$" { return 0;}
%%
```

Infix.y

```
%{
#include<stdio.h>
#include<math.h>
%}
%union {
double dval;
char symbol;
}
%token<dval>NUMBER
%token<symbol>LETTER
%token PLUS MINUS MULTIPLY DIVIDE OPEN CLOSE ENTER
%left PLUS MINUS
%left DIVIDE MULTIPLY
%nonassoc UMINUS
%type<dval>E
%%
print: E ENTER { printf("\n\v VALID INFIX EXP.....\n"); exit (0); }
;
E:E PLUS E
|
E MINUS E
|
E MULTIPLY E
```

```
|
E DIVIDE E
|
MINUS E %prec UMINUS {$$=-$2;}
|
OPEN E CLOSE { $$=$2;}
|
NUMBER { $$=$1; }
|
LETTER {$$=$1;}
;
%%

int main()
{
printf("\n Enter infix expression: ");
yyparse();
return 0;
}
void yyerror( char *msg)
{
printf("\n INVALID INFIX EXPRESSION.....: ");
}
int yywrap(){return(1);}
```

Conclusion: We learn how to recognize infix expression using LEX and YACC programs in linux. Also we learn how to recognize infix expression using LEX and YACC Programs.