UE19CS351 - Compiler Design

Session: Jan 2022 – May 2022

MANUAL

Semester	:	VI
Problem Statement	:	Mini-Compiler Part-I Introduction to Lex & Yacc tool. - To validate the syntax of a C program which consists of simple type declaration, if, if-else, and while constructs, Arithmetic/relational expressions
Course Anchor	:	Prof. Preet Kanwal
Teaching Assistants	:	Ananya Angadi, Alex Johnson & Joseph Dominic

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering
Session : Jan-May, 2022

UE19CS351 – COMPILER DESIGN (4-0-0-0-4)

Table of Contents

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering
Session : Jan-May, 2022

UE19CS351 – COMPILER DESIGN (4-0-0-0-4)

1. Introduction

Lexical analysis and syntactic analysis are the first two phases of a compiler. Lexical phase is responsible for dividing the input file into tokens, which are then passed to the syntax analyzer. **Lex is a tool used to write the lexical analyzer.** Lex file consists of a set of patterns and actions. Patterns are usually regular expressions. The lexer matches strings in the input based on the patterns, and performs the corresponding action.

Syntax analyzer uses the grammar of the language to validate the syntax of the input program. **Yacc tool is used to generate the parsing program for context free grammars.** Lex and Yacc are designed to complement each other. Tokens identified by Lex are passed on to Yacc which fits them into the grammar of the language.

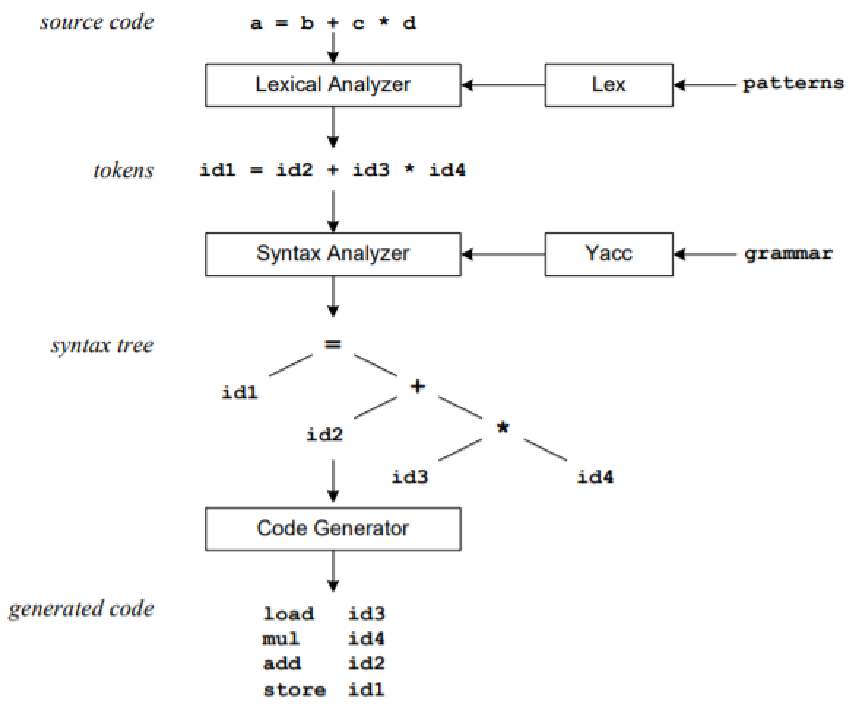


Fig 1. Phases of a compiler

2. Installation

i. Linux

```
$ sudo apt-get update
$ sudo apt-get install flex      (for lex)
$ sudo apt-get install bison  (for yacc)
```

ii. Windows

1. Download Flex 2.5.4a
2. Download Bison 2.4.1
3. Download DevC++
4. Install Flex at “C:\GnuWin32“
5. Install Bison at “C:\GnuWin32“
6. Install DevC++ at “C:\Dev-Cpp“
7. Open Environment Variables.
8. Add “C:\GnuWin32\bin;C:\Dev-Cpp\bin;”

iii. MacOS

1. Open the terminal.

1a. Install homebrew:

ruby -e "$(curl -fsSL $ https://raw.githubusercontent.com/Homebrew/install/master/install)"

2. Install Lex: **brew install flex**

3. Install Yacc: **brew install bison**

3. Structure of a lex program

The general outline of a lex program is as follows:

... definitions ...
%%
... rules …
%%
... subroutines ...

Lex tool converts the lex.l into a C file (yy.lex.c) containing the definition of the yylex() function, responsible for performing the lexical analysis. Statements which are part of the definitions section have scope throughout the program, and are akin to global declarations in the generated C file.
The rules section consists of the regex pattern and the corresponding action.
The subroutines section allows us to specify any additional subroutines required.
A sample lex program:

```
/*lex program to match identifiers*/

%{
#include<stdio.h>
int i = 0;
%}

/* Rules Section*/
%%

([a-zA-Z0-9])*    {printf("Identifier\n");}
.                 {printf("%s\n",yytext);}

%%

/* Subroutines section */
int main()
{
    // The function that starts the analysis
    yylex();
    return 0;
}
```

Definitions are enclosed inside %{ … %}

After these definitions, we can specify the regular definitions, which are commonly occurring regex patterns, so as to avoid specifying them repeatedly.

Following these is the %%, signaling the beginning of the rules section.

In this section, the regex for each token is given, followed by the action to be performed when there is a match: in this case, display 'Identifier'.

'.' is a metacharacter which can match any character.

'yytext' is a variable which holds the currently matched lexeme.

Here, printing yytext is equivalent to displaying the currently matched character as it is.

The second %% is followed by the subroutines.

The main() function defines the main function of the generated C file.

Main makes a call to the yylex() function, which performs the lexical analysis.

4. Structure of a yacc program

The general outline of a yacc program is almost the same as that of a lex program.

Variable declarations and other C code external to the definition of the parser are included within the %{ and %}, forming the definition part.

After the definitions, before the first %% is where we define keywords using the **%token** keyword.

Eg: **%token INT FLOAT ID NUM**

Following this, we specify the production rules for the grammar.

Eg:

```
Decl : Type ListVar ';'
    ;
Type : INT
      | FLOAT
      ;
ListVar : ID
        | ID ',' ListVar
        ;
```

; is used to mark the end of all productions from the non terminal

| indicates 'OR'

The rules section is concluded by the second %%.

The rest of the code includes function definitions for every function needed in the rules part. It also defines the main function, which in turn calls the yyparse() function, responsible for parsing.

5. Execution

Linux and Mac

Lex only

```
$ lex hello.l  // creates lex.yy.c
$ gcc lex.yy.c
$ ./a.out < input
```

Yacc only:

```
$ yacc -d prog.y     // creates y.tab.h, y.tab.c
$ gcc y.tab.c
$ ./a.out < input
```

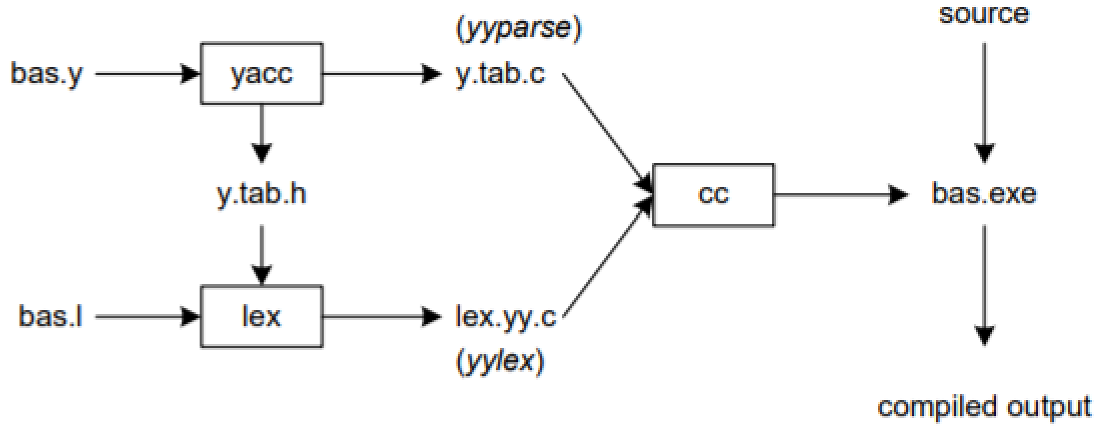
Both lex and yacc:

```
$ yacc -d prog.y
$ lex hello.l
$ gcc y.tab.c lex.yy.c -ll -ly   // link, load & execute
$ ./a.out < input
```

Windows

```
$ bison -dy prog.y
$ flex hello.l
$ gcc y.tab.c lex.yy.c
$ a.exe
```

6. Interaction between lex and yacc



The lex compiler converts the lex file to a C program in which the yylex() function is defined. This subroutine is responsible for lexical analysis.

```
$ lex hello.l
```

The above command generates the **lex.yy.c** file.

Similarly, the yacc compiler generates 2 files from the yacc code: **y.tab.h**, where tokens are declared, and **y.tab.c**, containing the yyparse function which drives the parsing.

```
$ yacc -d prog.y
```

The above command generates **y.tab.h** and **y.tab.c**.

We know that the parser drives the lexical analysis. Hence, the parser must be aware of the function which performs the lexical analysis. Hence, it is important to declare the **yylex()** function in the definitions part of the yacc file.

Similarly, since we expect the lex file to generate tokens, it must know their declarations. Hence, we must include the **y.tab.h** file in the definitions part of the lex file.

Finally, we must link the C program for lexer and C program for parser, and execute.

This is done by the following command:

```
$ gcc y.tab.c lex.yy.c -ll -ly
```

-ll: linking the lex file

-ly: linking the yacc file

This results in the **a.out** executable file.

PES University, Bangalore
(Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering
Session : Jan-May, 2022

UE19CS351 – COMPILER DESIGN (4-0-0-0-4)

7. **Mini-compiler**

lexer.l -

```
%{
   /* DEFINITIONS */
    #define YYSTYPE char* // specify the data type for variable yylval
    #include "y.tab.h" // y.tab.h contains token definitions
    #include <stdio.h>
    extern void yyerror(const char *); // declare the error handling
function

%}

/* Regular definitions */
digit      [0-9]
letter     [a-zA-Z]
id    {letter}({letter}|{digit})*
digits     {digit}+
opFraction      (\.{digits})?
opExponent      ([Ee][+-]?{digits})?
number     {digits}{opFraction}{opExponent}

%%

\/\/(.*) ; // ignore comments
[\t\n] ; // ignore whitespaces
"int"           {return T_INT;}
"char"          {return T_CHAR;}
"double"   {return T_DOUBLE;}
"float"    {return T_FLOAT;}
"while"    {return T_WHILE;}
"if"       {return T_IF;}
"else"          {return T_ELSE;}
```

```
"do"        {return T_DO;}
"#include"       {return T_INCLUDE;}
"main"      {return T_MAIN;}
\".*\"           {return T_STRLITERAL;}
"=="        {return T_EQCOMP;}
"!="        {return T_NOTEQUAL;}
">="             {return T_GREATEREQ;}
“<="         {return T_LESSEREQ;}
"("         {return *yytext;}
")"         {return *yytext;}
"."              {return *yytext;}
","              {return *yytext;} // yytext contains the currently
matched lexeme
"{"              {return *yytext;} // single char tokens can be passed
directly
"}"              {return *yytext;} // (they are tokens in themselves)
"*"              {return *yytext;}
"+"              {return *yytext;}
";"              {return *yytext;}
"-"              {return *yytext;}
"/"              {return *yytext;}
"="              {return *yytext;}
">"              {return *yytext;}
"<"              {return *yytext;}
{number}    {return T_NUM;}
{id}\.h     {return T_HEADER;}  // ending in .h => header file name
{id}             {return T_ID;}
.           {} // anything else => ignore

%%
```

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering
Session : Jan-May, 2022

UE19CS351 – COMPILER DESIGN (4-0-0-0-4)

parser.y -

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

      void yyerror(char* s); // error handling function
      int yylex(); // declare the function performing lexical analysis
      extern int yylineno; // track the line number


%}
/* declare tokens */
%token T_INT T_CHAR T_DOUBLE T_WHILE  T_INC T_DEC   T_OROR T_ANDAND
T_EQCOMP T_NOTEQUAL T_GREATEREQ T_LESSEREQ T_LEFTSHIFT T_RIGHTSHIFT T_NUM
T_ID T_PRINTLN T_STRING  T_FLOAT T_BOOLEAN T_IF T_ELSE T_STRLITERAL T_DO
T_INCLUDE T_HEADER T_MAIN


/* specify start symbol */
%start START

%%
START : PROG { printf("Valid syntax\n"); YYACCEPT; }   /* If program fits
the grammar,
                                                       syntax is valid */
        ;       /* Anything within {} is C code, it is the action
corresponding to the production rule */
```

```
PROG :  T_INCLUDE '<' T_HEADER '>' PROG     /* include header */
     |MAIN PROG                        /* main function   */
     |DECLR ';' PROG                   /* declarations    */
     | ASSGN ';' PROG                       /* assignments     */
     |                                 /* end of program */
     ;
/* Grammar for variable declaration */
DECLR : TYPE LISTVAR
     ;     /* always terminate with a ; */

LISTVAR : LISTVAR ',' T_ID
        | T_ID
        ;

TYPE : T_INT
       | T_FLOAT
       | T_DOUBLE
       | T_CHAR
       ;
/* Grammar for assignment */
ASSGN : T_ID '=' EXPR
     ;

EXPR : EXPR REL_OP E
       | E
       ;

REL_OP :   T_LESSEREQ
         | T_GREATEREQ
         | '<'
         | '>'
         | T_EQCOMP
         | T_NOTEQUAL

         ;
```

```
/* Expression Grammar */
E : E '+' T
    | E '-' T
    | T
    ;


T : T '*' F
    | T '/' F
    | F
    ;

F : '(' EXPR ')'
    | T_ID
    | T_NUM
    ;

/* Grammar for main function */
MAIN : TYPE T_MAIN '(' EMPTY_LISTVAR ')' '{' STMT '}';

/* argument list can be empty, or have a list of variables */
EMPTY_LISTVAR : LISTVAR
           |     /* similar to lambda */
           ;

/* statements can be standalone, or parts of blocks */
STMT : STMT_NO_BLOCK STMT
       | BLOCK STMT
       |
       ;

/* to give IF precedence over IF-ELSE */
%nonassoc T_IFX
%nonassoc T_ELSE
```

```
STMT_NO_BLOCK : DECLR ';'
       | ASSGN ';'
       | T_IF COND STMT %prec T_IFX    /* if loop */
       | T_IF COND STMT T_ELSE STMT    /* if else loop */
       | WHILE
       ;
BLOCK : '{' STMT '}';

/* Grammar for while loop */
WHILE : T_WHILE '(' COND ')' WHILE_2;

/* Condition can be an expression or an assignment */
COND : EXPR
       | ASSGN
       ;
// while loop may or may not have block of statements
WHILE_2 : '{' STMT '}'
        | ';'
        ;
%%

/* error handling function */
void yyerror(char* s)
{
      printf("Error :%s at %d \n",s,yylineno);
}

/* main function - calls the yyparse() function which will in turn drive
yylex() as well */
int main(int argc, char* argv[])
{
      yyparse();
      return 0;
}
```

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering
Session : Jan-May, 2022

UE19CS351 – COMPILER DESIGN (4-0-0-0-4)

To run:

```
$ lex lexer.l
$ yacc -d parser.y
$ gcc y.tab.c lex.yy.c -ll
$ ./a.out < test.c
```

Practice Questions:

1. Which among the lex and yacc files will you execute first? Why?
2. Write a lex program to find the number of words in the input file.
3. What is the significance of yytext, yylineno, yylval, yyerror and yywrap variables?
4. Write a regex to match comments in the input C file.

You can also try making further additions to the lex and yacc files for the mini compiler:

1. Support the 'for' construct
2. Include array declaration and definition
3. Support bitwise operators
4. Support initialization as part of definition

References:

1. https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
2. https://www.epaperpress.com/lexandyacc/
3. https://www.geeksforgeeks.org/introduction-to-yacc/