Khulna University of Engineering & Technology (KUET)

Report on

C/C++ Like Mini Compiler

Course No : **CSE 3212**

Course Name : **Compiler Design Laboratory**

Submitted By

Md. Jibon Mia

Roll: 1707038

Dept. of Computer Science and Engineering (CSE)

KUET - 9203

Under the Supervision of

Dola Das & Md. Ahsan Habib Sir

Lecturers

Dept. of Computer Science and Engineering

KUET - 9203

Submission Date: **14th June, 2021**

Introduction:

This is the project of a simple mini c/c++ like compiler that basically implements variable declarations, initializations, various loops (like for,

while and do..while) and print statements. To do so, I used **flex** and **bison** tools to generate tokens and validate them according to the program input. So what are these tools and what did they do? Let's take a closer look...

Flex/Lex: Flex is a fast lexical analyzer. It takes a program written in a combination of Flex and C, and it writes out a file (called **lex.yy.c**) that holds a definition of function **yylex(),** with the prototype,

int yylex(void)

yylex() reads from the file stored in a variable called **yyin** .So we can open a file for reading and store it into **yyin** before calling **yylex()** . Each time a program calls **yylex()** , it returns the next token .

When **yylex()** is finished, it calls another function named **yywrap()** . if **yywrap()** returns 1 , then **yylex()** returns 0 to its caller . That means EOF (End Of File) .

If **yywrap()** returns 0 , then **yylex()** assumes that a different file is stored in **yyin**. So it starts reading that file.

Pattern Matching primitives:

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal `"a+b"` (C escapes still work)
`[]`	character class

Some Flex Functions:

yywrap() - wraps the above rule section

yyin - takes the file pointer which contains the input

yylex() - this is the main flex function which runs the Rule Section

yytext - is the text in the buffer

input() - function reads the characters and makes them unavailable to the scanner. It basically truncates characters.

yylessn(n) - accepts n characters of the token and then they will be re-scanned for finding the next match. It basically keeps reducing n characters and returns the string for re-scanning .

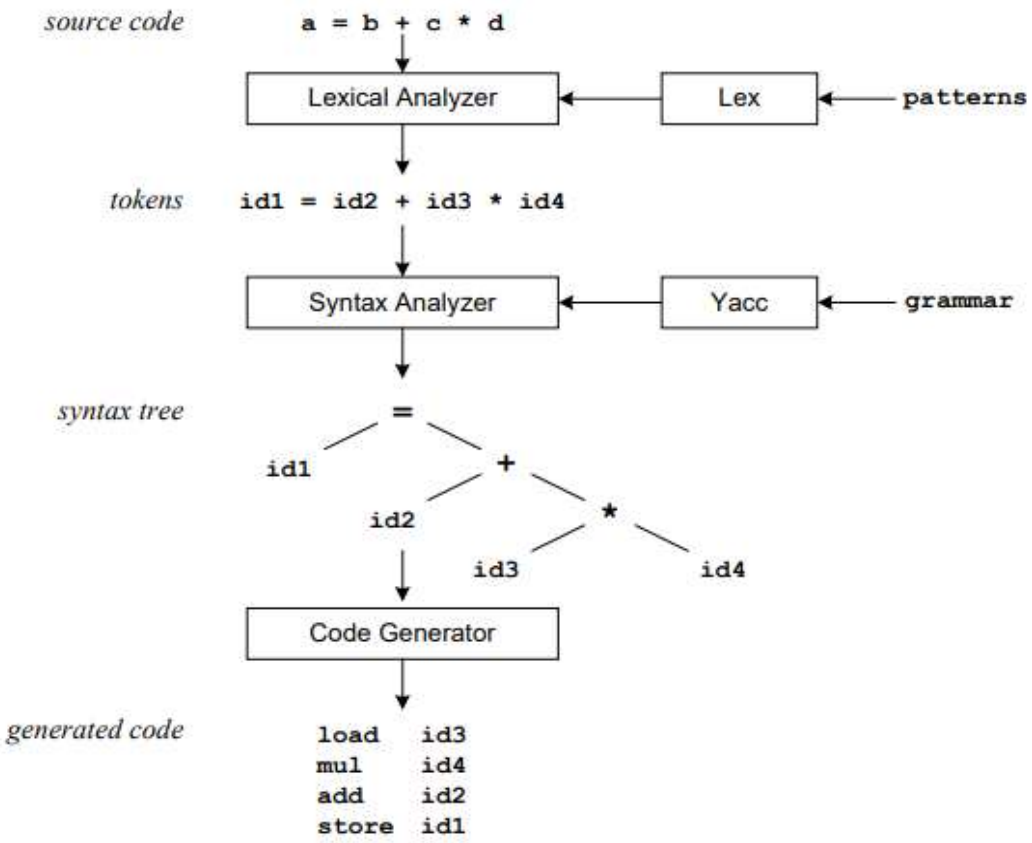
yymore() - function will output yytext, when the action part of any rule which has yymore() is finished. It basically outputs the matched input only after the rule has been executed.

yyterminate() - function ends the execution of the program as soon as it is called.

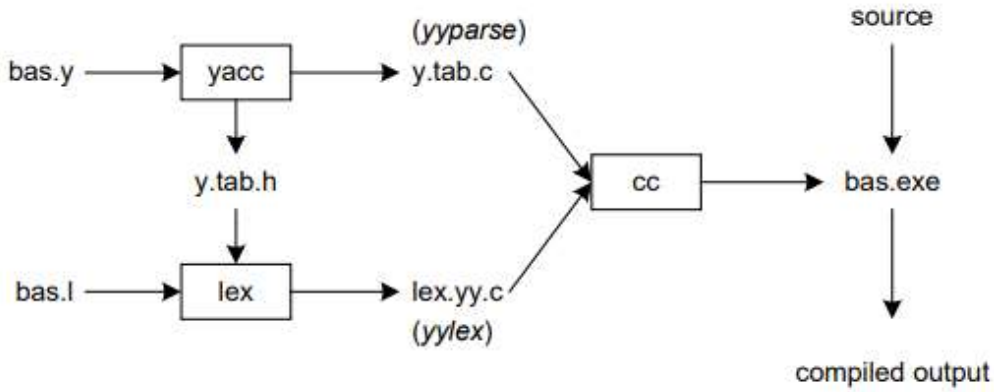
yy_flush_buffer() - unction flushes out the first two characters of the token by setting it to NULL('\0') character.

unput() - function reads the characters and basically replaces those characters.

Compilation sequence:



Compiler with flex/bison:



Bison/Yacc:

Grammars for yacc are described using a variant of Backus Naur Form (BNF). A BNF grammar can be used to express context-free languages. Most

constructs in modern programming languages can be represented in BNF.

For example, the grammar for an expression that multiplies and adds numbers is

E -> E + E

E -> E * E

E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier.

Input to **yacc** is divided into **three** sections. The **definitions section** consists of token declarations and **C code** bracketed by **"%{"** and **"%}".** The **BNF grammar** is placed in the rules section and user subroutines are added in the subroutines section.

%{

... **C definitions/declarations** ...

%}

... **bison declarations** like

%token **INTEGER, ...**

%%

... **grammer rules** ...

%%

... **subroutines/additional code** ...

Project In Depth:

Architecture: In this project, I implemented the followings..

1) **IF** statement

2) **IF...ELSE** statements

3) **IF...ELSE IF...ELSE** statements

4) **FOR()** loop

5) **WHILE()** loop

6) **DO..WHILE()** loop

7) **BOUND()** loop used for range of numbers

8) **TERNARY** Operations

=> Arithmatic expressions with **+, -, *, /, ++, --** are handled.

=> Boolean expressions with **>, <, >=, <=, ==, !=** are handled.

=> Error handling with **syntax error/errors** .

Design Stages and Implementation:

1) lexical Analysis:

=> FLEX tool was used to create a scanner for the language(c/c++).

=> The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.

=> All tokens included are of the form **T_<token-name>.**

Eg: **T_pl** for '**+**', **T_min** for '**-**', **T_lt** for '**<**' etc.

=> A global variable ‘yylavl’ is used to record the value of each lexeme scanned.

=> Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.

=> Scanning error is reported when the input string does not match any rule in the lex file.

=> The rules are regular expressions which have corresponding actions that execute on a match with the source input.

The following is the lex(.l) file...

```
%{
        #include<stdio.h>
        #include<stdlib.h>
        #include "pro.tab.h"
        extern int yylval;
%}
alpha   [A-Za-z_]
digit    [0-9]
%%
"int"    return INT;
"float" return FLOAT;
"char"  return CHAR;
"double"return DOUBLE;
"void"  return VOID;
"while"          return WHILE;
```

```
"for"     return FOR;
"if"      return IF;
"else"    return ELSE;
"cout"    return COUT;
"endl"    return ENDL;
"break"           return BREAK;
"continue"        return CONTINUE;
"do"      return DO;
"bound"           return BOUND;
"in"      return IN;
"..."     return T_dot;
"#include"        return INCLUDE;
"main()"          return MAINTOK;
"return"          return RETURN;

{digit}+{ yylval = atoi(yytext); return NUM; }
{alpha}({alpha}|{digit})*         { return ID; }
{alpha}{alpha}*".h"?      return H;
\".+\"    return STRING;

"<"       return T_lt;
">"         return T_gt;
"="         return T_eq;
"<="        return T_lteq;
">="        return T_gteq;
```

```
"=="        return T_eqeq;
"!="        return T_neq;
"+"         return T_pl;
"-"         return T_min;
"*"         return T_mul;
"/"         return T_div;
"++"        return T_incr;
"--"        return T_decr;
"!"         return T_neq;
"||"        return T_or;
"&&"        return T_and;
"."         return DOT;
[ \t\n]*            ;
.           return yytext[0];
%%
```

2) Syntax Analysis:

=> Syntax analysis is only responsible for verifying that the sequence

of tokens forms a valid sentence given the definition of the Programming Language grammar.

=> The design implementation supports

1) Multiple Variable declarations and initializations

2) Variables of type **char, int, float** and **double** .

3) Arithmetic and boolean expressions .

4) Constructs -

- **IF, IF...ELSE, IF..ELSE IF...ELSE,**

- **TERNARY Expression**

- **FOR()** , **WHILE()** , **DO...WHILE()** , **BOUND()** loop and

- **COUT** statement to print values.

5) Nested loops are also allowed.

6) One important thing is that this mini compiler accepts syntactically correct all programs according to the definition of the lex & yacc (flex & bison) file .

=> Yacc/Bison tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

The following grammer is used...

```
%{
        #include <stdio.h>      /* C declaration */
        #include <stdlib.h>
        #include <math.h>
        int yylex(void);
        int sym[26],store[26];
%}
```

%token ID NUM T_lt T_gt T_lteq T_gteq T_neq T_eqeq T_pl T_min T_mul T_div T_and T_or T_incr T_decr T_not T_eq WHILE INT CHAR FLOAT DOUBLE VOID H MAINTOK INCLUDE BREAK CONTINUE IF ELSE COUT STRING FOR ENDL DOT DO RETURN BOUND IN T_dot

```
%nonassoc IF
%nonassoc ELSE

%left T_lt T_gt
%left T_pl T_min
%left T_mul T_div

%%
S
        : START {printf("\n Successful Compilation.\n");exit(0);}
        ;

START
        : INCLUDE T_lt H T_gt MAIN
        | INCLUDE "\"" H "\"" MAIN
        ;
MAIN
        : VOID MAINTOK BODY
        | INT MAINTOK BODY
        ;

BODY
        : '{' C '}'
        ;
```

```
C
        : C statement ';'
        | C LOOPS
        | statement ';'
        | LOOPS
        ;

LOOPS
        : WHILE '(' COND ')' LOOPBODY
        {
                if($3=='true')
                {
                        printf("\n Inside WHILE loop.\n");
                }
                else
                {
                        printf("\n Exit from WHILE loop.\n");
                }
        }

        | FOR '(' ASSIGN_EXPR ';' COND ';' statement ')' LOOPBODY
        {
                if($5)
                {
```

```
                printf("\n Inside FOR loop.\n");
        }
        else
        {
                printf("\n Exit from FOR loop.\n");
        }
}

| IF '(' COND ')' LOOPBODY
{
        if($3)
        {
                printf("\n Inside IF block.\n");
        }
        else
        {
                printf("\n IF has no output.\n");
        }
}

| IF '(' COND ')' LOOPBODY ELSE LOOPBODY
{
        $$ = $3 ;
        if($3)
        {
```

```
                printf("\n Inside IF block.\n");
        }
        else
        {
                printf("\n Inside ELSE block.\n");
        }
}

| IF '(' COND ')' LOOPBODY ELSE IF '(' COND ')' LOOPBODY ELSE LOOPBODY
{
        if($3=='true')
        {
                printf("\n Inside IF block.\n");
        }
        else if($9=='true')
        {
                printf("\n Inside ELSE IF block.\n");
        }
        else
        {
                printf("\n Inside ELSE block.\n");
        }
}

| DO BODY WHILE '(' COND ')' LOOPBODY
```

```
{
        printf("\n Inside DO-WHILE loop.\n");

        $$ = $5;
        if($5)
        {
                printf("\n Continue to DO-WHILE loop.\n");
        }
        else
        {
                printf(" \n Exit from DO-WHILE block.\n"); exit(0);
        }
}
;

| BOUND LIT IN LIT T_dot LIT LOOPBODY
{
        $$ = $5;
        if($5)
        {
                printf("\n Inside Bound() loop.\n");
        }else{
                printf("\n Exit from Bound() loop.\n");
        }
```

```
                for(int i=$4; i<=$6; i++)
                {
                        printf(" Value: %d \n", i);
                }
        }
        ;

LOOPBODY
           : '{' C '}'
           | ';'
           | statement ';'
           ;

statement
        : ASSIGN_EXPR
        | ARITH_EXPR
        | TERNARY_EXPR
        | PRINT
        ;

COND  : LIT RELOP LIT                   /* (x >= y), (x<5), (10>5) */
                | LIT                           /* (x) */
                | LIT RELOP LIT bin_boolop LIT RELOP LIT      /* ((w>x) && (y<z)) */
                | un_boolop '(' LIT RELOP LIT ')'             /* !(x<=y) */
                | un_boolop LIT RELOP LIT     /*    (!x < y) */
```

```
                | LIT bin_boolop LIT              /*    (x && y) */
                | un_boolop '(' LIT ')'           /* (!(x)) */
                | un_boolop LIT                         /* (!x) */
                ;


ASSIGN_EXPR
                : TYPE ID1      {printf("\n Valid Declaration.\n");}
        | ID T_eq ARITH_EXPR          {printf("\n Valid Initialization.\n");}
        | ID T_eq TERNARY_EXPR
        {
                printf("\n Valid Ternary Initialization.\n"); $$ = $3 ;
        }
        | ASSIGN_EXPR ',' ID T_eq ARITH_EXPR        {printf("\n Valid Initialization.\n");}
        | ID T_eq STRING
        | ASSIGN_EXPR ',' ID T_eq STRING
        | RETURN LIT
                ;


ID1     : ID
        | ID1 ',' ID
                | ID T_eq ARITH_EXPR
        | ID T_eq TERNARY_EXPR
        | ID1 ',' ID T_eq ARITH_EXPR
        | ID T_eq STRING
        | ID1 ',' ID T_eq STRING
```

```
        ;

ARITH_EXPR
                : LIT                           {$$=$1; }       /* x, 5 */
        | LIT DOT ARITH_EXPR                    {$$=$1; }       /* 5.5 */
                | LIT T_pl ARITH_EXPR                           /* x + y */
        {
                $$=$1 + $3;
                printf("\n Add Value = %d \n", $$);
        }

                | LIT T_min ARITH_EXPR                          /* x - y */
        {
                $$=$1 - $3;
                printf("\n Sub Value = %d \n", $$);
        }

                | LIT T_mul ARITH_EXPR                          /* x * y */
        {
                $$=$1 * $3;
                printf("\n Mult Value = %d \n", $$);
        }

                | LIT T_div ARITH_EXPR                  {
```

```
            if($3)
            {
                $$=$1/$3;
                printf("\n Div Result = %f \n", $$);
            }else{
                $$=0;
                printf("\n Division by zero is not possible\n");
            }
        }

            | LIT bin_boolop ARITH_EXPR{$$=$1&&$2, $$=$1||$2; }      /* x && y, r||y */
            | LIT un_arop                   {$$=$1++, $$=$1-- ;}      /* x++, y-- */
            | un_arop ARITH_EXPR            {$$=++$1, $$=--$1 ;}      /* ++x, --y */
            | un_boolop ARITH_EXPR          {$$=!$1; }      /* !x */
            ;

TERNARY_EXPR
            : '(' COND ')' '?' statement ':' statement
        {
            $$ = $2 ;
            if($2)
            {
                printf("\n Ternary Value: %d\n", $5);
            }
            else
```

```
		{
			printf("\n Ternary Value: %d\n", $7);
		}
	}
		;

PRINT  : COUT T_lt T_lt STRING
		| COUT T_lt T_lt STRING T_lt T_lt ENDL
	| COUT T_lt T_lt LIT                                    /* cout << x ; */
	{
		$$ = $4;
		if($4)
		{
			printf("\n Value: %d\n", $4);
		}
	}
	| COUT T_lt T_lt LIT T_lt T_lt ENDL                     /* cout << x << endl; */
	{
		$$ = $4;
		if($4)
		{
			printf("\n Value: %d\n", $4);
		}
	}
	| COUT T_lt T_lt LIT T_lt T_lt LIT                      /* cout << x << y; */
```

```
        | COUT T_lt T_lt LIT T_lt T_lt LIT T_lt T_lt ENDL        /* cout << x << y << endl; */
                ;
LIT     : ID
                | NUM
                ;
TYPE    : INT
                | CHAR
                | FLOAT
                | DOUBLE
                ;
RELOP   : T_lt
                | T_gt
                | T_lteq
                | T_gteq
                | T_neq
                | T_eqeq
                ;
bin_boolop
                : T_and
                | T_or
                ;
un_arop         : T_incr
                | T_decr
                ;
un_boolop
```

```
                    : T_not
                    ;
%%

int yywrap()
{
        return 1;
}

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

int main()
{
        printf("\n Your Program:\n");
        freopen("pro.txt","r",stdin);
        yyparse();
        return 0;
}
```

Input Program File for test:

```
#include <program.h>
void main()
{
	int x, y, z, add, sub, mult;
	float div;

	x = 15;
	cout << x << endl;

	y = 20;
	cout << y << endl;

	z = 25;
	cout << y << endl;

	add = 15 + 20 ;
	cout << add << endl;

	sub = 20 - 15 ;
	cout << sub << endl;

	mult = 15 * 25 ;
	cout << mult << endl;
```

```
div = 10.5 / 0 ;
cout << div << endl;

int a = 11, b = 22, c = 33;

cout << c << endl;

int ans;

ans = ( 5 < 10 ) ? 10 : 5 ;

cout << ans << endl;

char name = "Jibon";


if(10 < 15)
{
        cout << 15 << endl;
}


int mark = 70;

if(mark>=80)
```

```
{
        cout << 80 << endl;
}
else if(mark <= 79)
{
        cout << 75 << endl;
}
else
{
        cout << 65 << endl;
}

int loop=1;
do
{
        int m,n,o;
        char p;

        m = 37, n = 38;
        cout << n << endl;
}
while(loop==1);


while(mark<=80)
```

```
	{
		int newMark = mark + 10;


		if(newMark >= 80){
			cout << 80 << endl;
		}
		else{
			newMark++;
		}
	}


	for(int x=1; x<=7; x++)
	{
		cout << x << endl;
	}


	bound val in 5...10
	{
		cout << val << endl;
	}
}
```

Output File for the Input program:

Your Program:

Valid Declaration.

Valid Declaration.

Valid Initialization.

Value: 15

Valid Initialization.

Value: 20

Valid Initialization.

Value: 25

Add Value = 35

Valid Initialization.

Value: 20

Sub Value = 5

Valid Initialization.

Value: 15

Mult Value = 375

Valid Initialization.

Value: 25

Division by zero is not possible

Valid Initialization.

Valid Declaration.

Value: 33

Valid Declaration.

Ternary Value: 10

Valid Ternary Initialization.

Value: 5

Valid Declaration.

Value: 15

Inside IF block.

Valid Declaration.

Value: 80

Value: 75

Value: 65

Inside ELSE block.

Valid Declaration.

Valid Declaration.

Valid Declaration.

Valid Initialization.

Valid Initialization.

Value: 38

Inside DO-WHILE loop.

Continue to DO-WHILE loop.

Add Value = 90

Valid Declaration.

Value: 80

Inside IF block.

Exit from WHILE loop.

Valid Declaration.

Value: 7

Inside FOR loop.

Value: 10

Inside Bound() loop.

Value: 5

Value: 6

Value: 7

Value: 8

Value: 9

Value: 10

Successful Compilation.