Topic: Compiler Project

CSE 3212: Compiler Design Laboratory

Submitted To:

Dola Das

Lecturer,

Department of Computer Science and Engineering,

Khulna University of Engineering & Technology, Khulna.

Md. Ahsan Habib Nayan

Lecturer,

Department of Computer Science and Engineering,

Khulna University of Engineering & Technology, Khulna.

Submitted By:

Md. Tahmid Hasan Fuad

Roll: 1707114

Department of Computer Science and Engineering,

Khulna University of Engineering & Technology, Khulna.

Submission date: 15 June, 2021.

Introduction:

A compiler is a program that translates a source program written in a high-level programming language into machine code for some computer architecture. The generated machine code can be later executed many times against different data each time. In our lab, we have learnt how to design a simple compiler with flex and bison. At first, we learnt how to use flex. Then we learnt about bison. After combining both, we try to make a simple compiler. So, the final outcome of this lab is a simple compiler project with flex and bison.

Flex:

The patterns which are created with a text editor, Lex will read these patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on the pattern, and converts the strings into tokens. Tokens are numerical representations of strings, and simplify processing.

To implement these, we need a tool named "Flex". This tool generates a '.c' file from a '.l' file. In the '.l' file we wrote the regular expressions and corresponding program fragments. It generates a lex.yy.c file more specifically. The structure of flex file-

{Definitions}

%%

{Rules}

%%

{User Subroutine}

In my project, all the rules are defined in the .l file.

Bison:

Yacc will read the follow up grammar in the text file, which has been created. It also generates C code for syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze the tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical

structure of the tokens. For example, operator precedence and associativity are apparent in the syntax tree.

We need “Bison” to implement and run the whole process, which is a free version of “Yacc". It takes the language specification in the form of an LALR grammar and generates the parser. It is saved as a ‘.y’ file. After running, it creates ‘a.tab.h’ file and ‘a.tab.c’ file. The structure of bison file-

%{

Prologue (C declarations)

%}

Bison declarations

%%

Grammar rules

%%

Epilogue (Additional C codes)

In the code generation part, both the output of .y file and .l file is merged and does a depth-first walk of the syntax tree to generate code.

To compile the whole process, some lines of code of command promoted are followed. After executing those, an .exe file is created and with the .exe file we can see the output of the lines of coding. These commands-

```
bison -d code.y                 # create y.tab.h and y.tab.c

lex code.l                      # create lex.yy.c

gcc y.tab.c lex.yy.c -code.exe  # compile the output
```

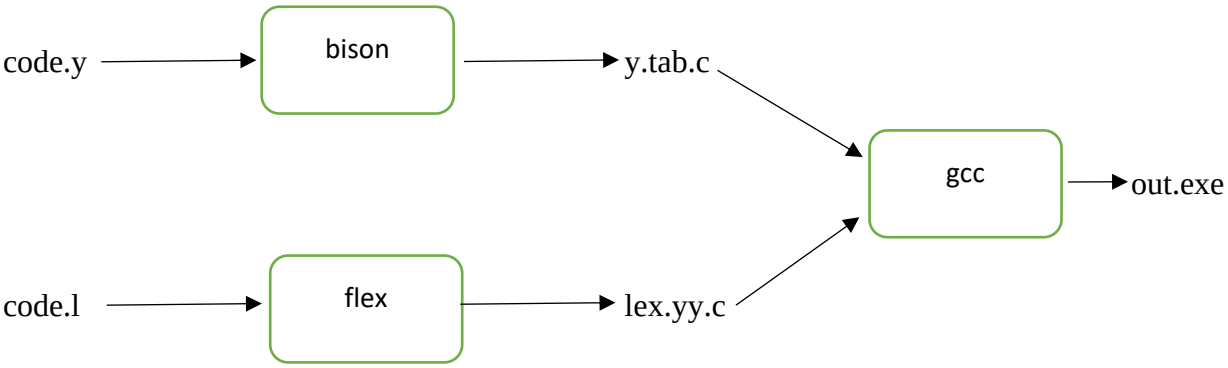

Figure 1: Building a compiler with flex/bison.

Procedure:

1. Write the code in flex and bison file.
2. Save flex file with .l extension.
3. Save bison file with .y extension.
4. Write the necessary commands in command promote and create output file with .exe extension.
5. Run the .exe file to see output.

Features:

1. Body declaration.
2. Variable declaration (Integer, Float and String).
3. Variables' values assignment.
4. Arithmetic operations. (+, -, *, /, %).
5. Logical operations. (==, !=, <=, >= etc).
6. For Loop.
7. While Loop.
8. If-else.
9. Switch-case.
10. Build in Trigonometric Functions. (SIN, COS, etc).
11. Print Function.
12. Single line comment.
13. Build in Max, Min Function.
14. Build in Prime checking Function.
15. Build in Factorial Function.

Tokens:

Sl. No.	Token	Input String	Definition
1	MAIN	main	Defines the start (mimic of C main)
2	START	{	Starts section.
3	END	}	Ends section.
4	T_INT	int	Integer variable declaration.
5	T_DOUBLE	double	Float variable declaration.
6	T_CHAR	string	Character variable declaration.
7	IF	if	If-else condition.
8	ELSE	else	If-else condition.
9	WHILE	while	While-loop token.
10	INC	inc	While-loop token.
11	SWITCH	switch	Switch-case token.
12	CASE	case	Switch-case token.
13	DEFAULT	default	Switch-case token.
14	BREAK	break	Switch-case token.
15	FROM	from	For-loop token.
16	TO	to	For-loop token.
17	INCREMENT	++	Decrement Operation
18	DECREMENT	--	Increment operation
19	EQUAL	==	Equal
20	N_EQUAL	!=	Not Equal
21	G_EQUAL	>=	Greater or Equal
22	L_EQUAL	<=	Less or Equal
23	PRINT	Print	Print Function.
24	SIN	SIN	Sine Function.
25	COS	COS	Cosine Function.
26	TAN	TAN	Tangent Function.
27	LOG	LOG	e base Log.
28	LOG10	LOG10	10 based Log.
29	MAX	MAX	Max determining Function.
30	MIN	MIN	Min determining Function.
31	PRIME	PRIME	Prime checking Function.
32	FACT	FACT	Factorial Function.

Used CFG:

Context Free Grammar (CFG) is a set of recursive rules used to generate patterns of strings. Below, the CGF for my project is described.

```
program:        MAIN START begin END      {code}
                ;
begin:                                    {code}
                | begin statement
                ;
statement:                                {code}
                | expres                  {}
                | declaration             {code}
                | PRINT '(' VAR ')' ';'   {code}
                | FROM INT TO INT INC INT START expres END
                                          {code}
                | IF '(' expres ')' START expres ';' END  then
                                          {code}
                | IF expres START expres ';' END ELSE START expres ';' END
                                          {code}
                | WHILE VAR INCREMENT '<' INT START expres END
                                          {code}
                | WHILE VAR DECREMENT '>' INT START expres END
                                          {code}
                | SWITCH '(' expres ')' START B  END
                                          {}
                ;
```

```
B   :   C
    | C D
    ;

C   :   C '+' C
    | CASE INT ':' expres ';' BREAK ';' {}
    ;

D   :   DEFAULT ':' expres ';' BREAK ';' {}
    ;

declaration:
        T_INT INTID

    | T_DOUBLE DOUBLEID

    | T_CHAR CHARID
    ;

INTID  :
        INTID ',' INT_ID
    | INT_ID

INT_ID:     VAR '=' expres          {code}
    | VAR                           {code}
    ;

DOUBLEID:
        DOUBLEID ',' DOUBLE_ID
    | DOUBLE_ID

DOUBLE_ID:
        VAR '=' expres              {code}
    | VAR                           {code}
    ;
CHARID:
        CHARID ',' CHAR_ID
    | CHAR_ID
    ;
CHAR_ID:
        VAR '=' STRING
    | VAR                           {code}
    ;

    expres:     INT                 {}
    | DOUBLE                        {}
    | VAR                           {code}

    | expres '+' expres             {}
    | expres '-' expres             {}
```

```
| expres '*' expres                {}
| expres '/' expres                {}
| expres '^' expres                {}
| expres '%' expres                {code}

| expres '>' expres                {}
| expres '<' expres                {}
| expres EQUAL expres              {}
| expres N_EQUAL expres            {}
| expres L_EQUAL expres            {}
| expres G_EQUAL expres            {}

| SIN '(' expres ')'               {code}
| COS '(' expres ')'               {code}
| TAN '(' expres ')'               {code}
| LOG10 '(' expres ')'             {code}
| LOG '(' expres ')'               {code}
| MAX '(' expres ',' expres ')'    {code}
| MIN '(' expres ',' expres ')'    {code}

| PRIME '(' expres ')'             {code}
| FACT '(' expres ')'              {code}
;
```

Terminal Commands:

1. bison -d compilerpro.y
2. flex compilerpro.l
3. gcc lex.yy.c compilerpro.tab.c -o output
4. output

Sample Input:

```
main{
int ab=5;

print(ab);
int b=ab+4;
print(b);

if b>ab{
1+2;
}

else{
```

```
2+12;
}
string c = "abc";
print(c);

int a=7;
while a -- > 3{
2+4
}

#single;
from 2 to 6 inc 1{
1+3
}

SIN(30);
LOG10(100);
LOG(16);
COS(60);
TAN(45);

MAX(2,3);
PRIME(5);
PRIME(10);

FACT(3);
}
```

Sample Output:

```
program beagan.
ab is stored at index :0: with value :5
variable declared.
ab is an Integer.Value of ab is: 5
b is stored at index :1: with value :9
variable declared.
b is an Integer.Value of b is: 9
Value of expression in if block is 3.0000
c is stored at index :2: with value :"abc"
variable declared.
c is a String.Value of c is: "abc"
a is stored at index :3: with value :7
variable declared.
Inside while loop.
Value of expression:6.0000
Inside while loop.
Value of expression:6.0000
Inside while loop.
```

```
Value of expression:6.0000
Inside while loop.
Value of expression:6.0000

Single line comment.
expression in 2th : 4.0000
expression in 3th : 4.0000
expression in 4th : 4.0000
expression in 5th : 4.0000
expression in 6th : 4.0000
Value of Sin is 0.500001.
Value of Log10 is 2.000000.
Value of Log is 2.772589.
Value of Cos is 0.499998.
Value of Tan is 1.000004.
3 is greater.
5 is prime.
10 is not prime.

Factorial of 3 is 6.

program end
```

Discussion:

This compiler follows bottom-up parsing in the input code. In this task, if-else, for-loop and while-loop are showed. However, it is only pattern matching, built with flex and bison. As, flex-bison have not enough functionality to make original version. It is designed as C and python language. The code segment for this project can be found in the following github link.

Conclusion:

Compiler is an essential topic in Computer science. If we want to work with the core of programming languages, we need to learn compiler design. Following the steps, I tried to design a simple compiler project which mimics python and C programming language. This project executes perfectly without any error.