Compiler Design Using Flex & Bison

ArenaBoard Compiler Design Lab

Md Sanaul Haque Shanto

November 15, 2017

Content

1 Introduction **2**
1.1 Compiler . . . 2
1.2 Flex . . . 2
1.3 Bison . . . 2

2 Procedure **3**
2.1 Figure . . . 3
2.1.1 Steps . . . 3
2.1.2 Features . . . 4
2.2 Used CFG . . . 5
2.3 Terminal Commands . . . 7

3 Result **7**
3.1 Sample Input . . . 7
3.2 Sample Output . . . 8

4 Discussion **9**

1 Introduction

1.1 Compiler

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 my lab, I have learn how to design a simple compiler with flex and bison. At first, I learn how to use flex. Then I learn about bison. After combining both, I try to make a simple compiler. So, the final outcome of this lab is a simple compiler with flex and bison.

1.2 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 I wrote the regular expressions and corresponding program fragments. It generates a (lex.yy.c file, 2017) 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 (CompilerPro.l, 2017).l file.

1.3 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.

I 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 (compilerpro.y file, 2017) '.y' file. After running, it creates (compilerpro.tab.h, 2017) 'a.tab.h' file and (compilerpro.tab.c, 2017) '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.

```
%{
Prologue (C declarations)
%}
Bison declarations
%%
Grammar rules
%%
Epilogue (Additional C codes)
```

2 Procedure

2.1 Figure

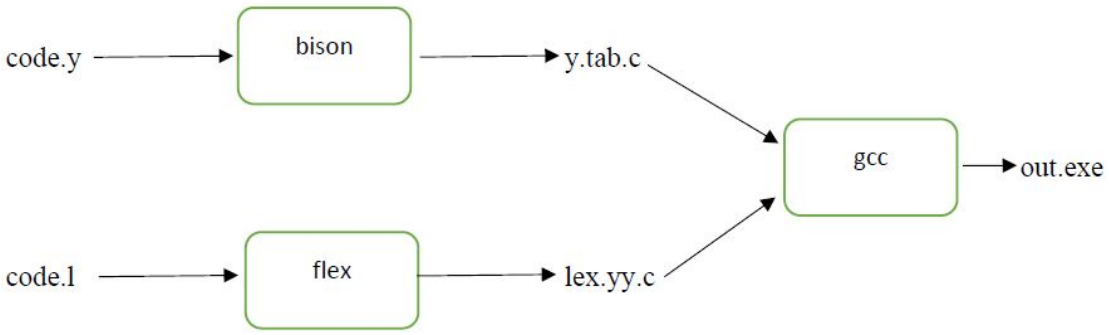

Figure 1: Building a compiler with flex/bison.

2.1.1 Steps

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.

2.1.2 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
01	MAIN	main	Defines the start (mimic of C main)
02	START	{	Starts section.
03	END	}	Ends section.
04	T_INT	int	Integer variable declaration.
05	T_DOUBLE	double	Float variable declaration.
06	T_CHAR	string	Character variable declaration.
07	IF	if	If-else condition.
08	ELSE	else	If-else condition.
09	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	SINE	SIN	Sine Function.
25	COS	COS	Cosine Function
26	TAN	TAN	Tangent Function.
27	LOG	LOG	e-base Log
28	LOG10	LOG10	10 base Log
29	MAX	MAX	Max determining Function.
30	MIN	MIN	Min determining Function.
31	PRIME	PRIME	Prime checking Function.
32	FACT	FACT	Factorial Function.

2.2 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};
```

2.3 Terminal Commands

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

3 Result

3.1 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);
}
```

3.2 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
```

4 Discussion

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

References

CompilerPro.l. (2017). Compilerpro.l file. Retrieved October 9, 2017, from https://github.com/Md-Sanaul-Haque-Shanto/compiler-design-for-ArenaBoard/blob/main/compilerpro.l

compilerpro.tab.c. (2017). Compilerpro.tab.c file. Retrieved October 9, 2017, from https://github.com/Md-Sanaul-Haque-Shanto/compiler-design-for-ArenaBoard/blob/main/compilerpro.tab.c

compilerpro.tab.h. (2017). Compilerpro.tab.h file. Retrieved October 9, 2017, from https://github.com/Md-Sanaul-Haque-Shanto/compiler-design-for-ArenaBoard/blob/main/compilerpro.tab.h

compilerpro.y file. (2017). Compilerpro.y file. Retrieved October 9, 2017, from https://github.com/Md-Sanaul-Haque-Shanto/compiler-design-for-ArenaBoard/blob/main/compilerpro.y

lex.yy.c file. (2017). Lex.yy.c file. Retrieved October 9, 2017, from https://github.com/Md-Sanaul-Haque-Shanto/compiler-design-for-ArenaBoard/blob/main/lex.yy.c