Report on CSE - 3212

Compiler Design Laboratory

Project Manual of Simple Compiler Design using Bison and Flex

Submitted To:

Dola Das

Lecturer

Department of Computer Science & Engineering

Khulna University of Engineering & Technology

Md. Ahsan Habib Nayan

Lecturer

Department of Computer Science & Engineering

Khulna University of Engineering & Technology

Submitted By:

Alvi Ahmmed Nabil

Session: 2019 - 2020

Section: A

Roll: 1707009

Year: 3rd Term: 2nd

Department of Computer Science & Engineering

Submission Date: 15 June, 2021

Contents

Topics: **Page No**

Objectives

1. To know about the basics of a compiler and how things work behind a program execution.
2. To acknowledge the translation of a high level language into a low level language.
3. To learn about the top down parser and the bottom up parser.
4. To acquire knowledge about the Flex and Bison for implementation of a compiler using C language.
5. To create a our own new language and it’s semantic and syntactic rules.
6. To check what our compiler is capable of, in other word what it responds to different types of input.
7. To implement Regular Expression, Context Free Grammar in the compiler.

Introduction:

a compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program. There are many different types of compilers which produce output in different useful forms. A compiler that can run on a computer whose

CPU or operating system is different from the one on which the code it produces will run is called a cross-compiler. A bootstrap compiler is written in the language that it intends to compile. A program that translates from a low-level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a source-to-source compiler or transcompiler. A language rewriter is usually a program that translates the form of expressions without a change of language. The term compiler-compiler refers to tools used to create parsers that perform syntax analysis.

A compiler is likely to perform many or all of the following operations: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness

Flex:

Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. .It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Flex code can be derived into three particular section as follows:

{Definition}

%%

{Rules}

%%

{User Subroutine}

Bison:

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1), IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble.

Structure of Bison can be defined as:

```
%{
Prologue/c declaration (such as header inclution)
%}

Bison declarations
```

```
%%
Grammar rules
%%
Epilogue/ Optional C codes
```

Procedure

1.The code is divided into two part flex file (.l) and bison file (.y) .

2.Input expression check the lex (.y) file and if the expression satisfies the rule then it check the CFG into the bison file .

3.it’s a bottom up parser and the parser construct the parse tree .firstly ,matches the leaves node with the rules and if the CFG matches then it gradually goes to the root .

Terminal Comands/ Command Prompt Command to run the program:

1. Bison -d alvi.y
2. Flex alvi.y
3. gcc lex.yy.c alvi.tab.c -o object
4. object

Token

A token is the smallest element(character) of a computer language program that is meaningful to the compiler. The parser has to recognize these as tokens: identifiers, keywords, literals, operators, punctuators, and other separators .

Tokens that have been used in this compiler are:

Serial No.	Token	Corresponding Input String	Definition of the Token
1	MAIN	task	Codes starts running from here similar to the main() function in c/cpp
2	INT	PURNO	Refer the type of a variable as Integer
3	CHAR	OKKHOR	Refer the type of a variable as Character
4	FLOAT	DOSHOMIK	Refer the type of a variable as Float
5	SHURU	SHURU	Defines the start of a segment of a code
6	SHESH	SHESH	Defines the end of a segment of code
7	JOG	JOG	Acts for the addition operation

8	BIYOG	BIYOG	Acts for the subtraction operation
9	GUN	GUN	Acts for the multiplication operation
10	VAAG	VAAG	Acts for the divition operation
11	LOG	LOG	Acts for the e base Logarithm operation
12	LOG10	LOG10	Acts for the 10 base logarithm operation
13	SIN	SIN	Acts for the sin function
14	COS	COS	Acts for the cosine function
15	TAN	TAN	Acts for the tan function
16	EQUAL	SHOMAN	Checks whether two values are equal
17	NOTEQUAL	OSHOMAN	Checks whether two values are unequal
18	SHOW	SHOW	Output the result on the display
19	POW	POW	Acts for the power function
20	GT	BORO	Works to check if a number is greater than the other
21	GOE	BOROSHOMAN	Works to check if a number is greater than the other or equal
22	LT	CHOTO	Works to check if a number is less than the other

23	LOE	CHOTOSHOMAN	Works to check if a number is less than the other or equal
24	LOOP	LOOP	Acts as a for loop
25	OFFSET	OFFSET	Determines how much increments or decrements should happen in the for loop
26	WHILE	WHILE	Acts as a while loop
27	SWITCH	STEP	Works as the switch case in c/cpp
28	CASE	SCENE	Determines the case/scene for the switch
29	DEFAULT	ROOT	Works if none of the scene matches
30	IF	JODI	Check if a condition is true, then executes
31	ELSE	NOILE	Executes this only if the JODI condition comes out false
32	ELIF	OTHOBA	In between condition of JODI and NOILE

CFG

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

THE CFGs that are used in this compiler:

program: MAIN SHURU statement SHESH

;

statement:

|declaration statement

| assignment statement

| ifCondition statement

| for_code statement

| switch_code statement

| print_code statement

| powFunct statement

| sinFunct statement

```
        | cosFunct statement

        | tanFunct statement

        | logFunct statement

        | log10Funct statement

        |expression

        ;

print_code: SHOW '(' ID ')'';'

        ;

powFunct: POW '(' NUM ',' NUM ')'';'

        ;

sinFunct: SIN '(' NUM ')' ';'

        ;

cosFunct: COS '(' NUM ')' ';'

        ;

tanFunct: TAN '(' NUM ')' ';'

        ;

log10Funct: LOG10 '(' NUM ')' ';

        ;

logFunct: LOG '(' NUM ')' ';'
```

```
        ;
switch_code: SWITCH '(' ID ')' '{' case_code '}'
        ;
case_code: casenum_code default_code
        ;

casenum_code: CASE NUM '{' statement '}' casenum_code
        |
        ;
default_code: DEFAULT '{' statement '}'
        ;

for_code: LOOP ID THEKE NUM OFFSET NUM '{' statement '}'
        | LOOP ID THEKE ID OFFSET NUM '{' statement '}'
        |LOOP NUM THEKE ID OFFSET NUM '{' statement '}'
        |LOOP NUM THEKE NUM OFFSET NUM '{' statement '}'
        ;
while_code: WHILE'(' expression ')''{' statement '}'
        ;
```

```
ifCondition:  IF '(' expression ')''{'statement  '}' else_if_Condition else_Condition
;
else_if_Condition: ELIF '(' expression ')''{' statement  '}' else_if_Condition
|
          ;
else_Condition: ELSE '{' statement  '}’
|
          ;
;



declaration: TYPE ID1 ';'
          ;
TYPE:  INT
          | FLOAT
          | CHAR
          ;
ID1: ID1 ',' ID
          | ID
```

```
        ;
assignment: ID '=' expression ';'
        ;
expression: NUM
        | expression JOG expression
        | expression BIYOG expression
        | expression GUN expression
        | expression VAAG expression
        | expression '^' expression
        | expression LT expression
        | expression GT expression
        | expression GOE expression
        | expression LOE expression
        | '(' expression ')'
        | t
t: '(' expression ')'
        | ID
        | NUM
        ;
```

Features of this compiler

1.Can include associative header file

2. Main function

3.Comments(Single line and Multiline)

4.Variable declaration with multiple character

5. IF ELSE_IF ELSE Block

6.Variable assignment

7. For loop

8. While loop

9. Print function

10. Switch Case

11.Mathematical Expression

- Addition, Subtraction, Multiplication, Division,
- Power, Log () Operation, Log10() operation, Sin () operation,
- Tan () operation, Cos () operation.

Discussion:

This compiler is a very simple of form. It can do the basic i/o and arithmetic operation. Despite its simplicity it has lots of potential to achieve with the powerful header files it can include from c/cpp. This is a bottom up parser that means it parse the input file from bottom to top. Thought some of the core conditional functionality can't be obtained through this due to limitations of bison and flex's simplicity. Some of the function always needs double data type such as SIN,COS,LOG. Despite all the limitation it works on mark within the defined context free grammar.

Conclusion:

The objective was to learn how to implement a compiler using flex and bison which are very primitive tools to work with. But the lessons had been learnt. As the compiler works great with the CFG it had been provided. It was never meant be completely working compiler for every scenario. As a result it has some flaws which is expected. If the limitations of bison and flex wasn't a issue it could have been a great tool to work with.

Codes at:

https://github.com/AlviNabil/mini-Compiler-Project