

# **KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY**

## Department of Computer Science and Technology

**Title:**  A simple compiler using flex and Bison.

**Course Title:** Compiler Design Laboratory

**Submitted by:**

**Suraiya Akter Mumu**

Roll: ***1707100***

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

**Submission date:**  June 15, 2021

**Objective:**

After doing this project, we will be able to know

* About Flex and Bison.
* About token and how to declare rules against token.
* How to declare CFG (context free grammar) for different grammar like if else pattern, loop and so on.
* About different patterns and how they work.
* How to create different and new semantic and synthetic rules for the compiler.
* About shift and reduce policy of a compiler.
* About top down and bottom up parser and how they work.

**Introduction:**

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

***Flex:*** Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). An input file describing the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c. The C compiler compiles the lex.yy.c file into an executable file called a.out. The output file a.out takes a stream of input characters and produces a stream of tokens.

/\* definitions \*/

....

%%

/\* rules \*/

....

%%

/\* auxiliary routines \*/

....

***Bison:*** GNU Bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. **Bison** command is a replacement for the **yacc**. It is a parser generator similar to yacc. Input files should follow the yacc convention of ending in .y format.

/\* definitions \*/

....

%%

/\* rules \*/

....

%%

/\* auxiliary routines \*/

....

**Instruction in cmd when we use flex and bison together:**

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

**Features of this mini-compiler:**

* Import section or header declaration section.
* User define function section.
* Main part or body of the program.
* Declaration of different id or variable and assignment operation.
* If else condition.
* Arithmetic and logical operation.
* Loop (for loop and while loop).
* Switch-case condition.
* Print and show different values.
* Single line and multiple line command.
* Built-in sine function.
* Built-in cosine function.
* Built-in tan function.
* Built-in ln function.
* Built-in log10 function.
* Built-in factorial function.
* Built-in oddeven function.

**Token:**

A **token** is a pair consisting of a **token** name and an optional attribute value. The **token** name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The **token** names are the input symbols that the parser processes.

**Tokens used in this mini-project:**

Bellow in the table it is shown those tokens that is used in this mini-project and their realtine meaning-

|  |  |  |  |
| --- | --- | --- | --- |
| **Serial no.** | **Token** | **Input string** | **Realtime meaning of Token** |
| 1. | NUMBER | "-"?[0-9]+ | Any integer number either positive or negative. |
| 2. | VARIABLE | [a-zA\_Z0-9]+ | Any string using upper case alphabets, lower case alphabets and number. |
| 3. | ENDED | ; | Indicate ends of a line. |
| 4. | INT | int | Declaretion of integer variable. |
| 5. | FLOAT | float | Declaretion of floation point variable. |
| 6. | CHARACTER | char | Declaretion of character variable. |
| 7. | IF | if | If condition start |
| 8. | EF | else | Else condition start |
| 9. | FOR | for | For loop line c programming |
| 10. | COLON | : | Colon sign like default |
| 11. | SWITCH | switch | Switch case like c programming |
| 12. | DEFAULT | default | Alternative option |
| 13. | VALUE | value | For printing any variable |
| 14. | ASSIGN | = | Assignment operator |
| 15. | INC | ++ | Used for increment any value by one |
| 16. | DEC | -- | Used for decrement any value by one |
| 17. | LT | < | Less than sign |
| 18. | GT | > | Greater than sign |
| 19. | EQ | == | Check equal or not |
| 20. | BSIEQ | >= | Greater than or equal sign |
| 21. | START | main | Start of a program |
| 22. | UP | up | How many increase we want to |
| 23. | DOWN | down | How many decrease we want to |
| 24. | CASE | case | Different case |
| 25. | WHILE | while | While loop like c program |
| 26. | FACT | fact | Calculating factorial of a number |
| 27. | DEF | def | Funtion defination |
| 28. | HEADER | something.h | Header function |
| 29. | IMPORT | #include | Needs for including header function |
| 30. | SIN | sin | Sine function |
| 31. | COS | cos | Cosine function |
| 32. | TAN | tan | Tangent function |
| 33. | LN | ln | ln function |
| 34. | LOG10 | log10 | Log function |
| 35. | ODDEVEN | OddEven | For calculation Oddeven function |
| 36. | ‘+’ | + | Addition operation |
| 38. | ‘-’ | - | Subtraction operation |
| 39. | ‘\*’ | \* | Multication operation |
| 40. | ‘/’ | / | Division operation |
| 41. | ‘%’ | % | Module operation |
| 42. | ‘(’ | ( | First bracket opening |
| 43. | ‘)’ | ) | First bracket closing |
| 44. | ‘{’ | { | second bracket opening |
| 45. | ‘}’ | } | second bracket closing |
| 46. | ‘^’ | ^ | Power operation |
| 47. | ‘[’ | [ | Third bracket opening |
| 48. | ‘]’ | ] | Third bracket closing |
| 49. | ‘,’ | , | Comma |

Table 1. Realtime meaning of tokens that are used in project

**Structure of the project:**

CFGs used in this project:

program:

import func START '{' lines '}'

| /\* NULL \*/

;

import: IMPORT LT HEADER GT { …………… }

| /\*empty\*/

;

func: DEF VARIABLE '(' param ')' '{' lines '}'

param :

param ',' VARIABLE

{

………

}

| VARIABLE

{

………..

}

;

lines : /\* empty \*/

|NUMBER

| lines codes

| declare

;

declare :

type id ENDED { ………….. }

;

type :

INT

| FLOAT

| STRING

;

id :

id ',' VARIABLE

{

………………

}

| VARIABLE

{

……………..

}

;

codes:

codes after

| after

;

after :

ENDED

| declare

| expression ENDED

{

………………

}

| VALUE '(' VARIABLE ')' ENDED

{

……………….

}

| VARIABLE '=' expression ENDED

{

…..……………

}

| IF '[' expression ']' '{' codes '}' %prec IFX

{

………………

}

| IF '[' expression ']' '{' codes '}' EF '{' codes '}' %prec IFX

{

……………….

}

| FOR '(' VARIABLE '=' NUMBER ',' VARIABLE KOMEQ NUMBER ',' VARIABLE UP NUMBER')' '{' codes'}'

{

………………..

}

| FOR '(' VARIABLE '=' NUMBER ',' VARIABLE BSIEQ NUMBER ',' VARIABLE DOWN NUMBER ')' '{' codes'}'

{

…………………

}

| WHILE '(' VARIABLE LT NUMBER ',' NUMBER ')' '{' codes '}'

{

………………..

}

| SWITCH '[' Exp\_case ']' '{' code '}'

;

Exp\_case:

expression

{

……………..

}

;

code: /\* empty \*/

| code expression COLON '{' codes'}'

{

……………..

}

| code DEFAULT COLON '{' codes '}'

{

…………

}

;

expression:

NUMBER

| VARIABLE

{

……………..

}

| expression '+' expression

{

……………..

}

|expression '-' expression

{

…………….

}

|expression '\*' expression

{

……………..

}

| expression '/' expression

{

…………..

}

| expression '^' expression

{

…………….

}

| expression '%' expression {……………. }

| '(' expression ')' {……..……..}

| expression LT expression { ……..…… }

| expression GT expression { ………..…. }

| expression KOMEQ expression { ………..….. }

| expression BSIEQ expression { .………….. }

| SIN expression {………………}

| COS expression {……………….}

| TAN expression {……………….}

| LOG10 expression {……………….}

| LN expression {……………....}

|ODDEVEN expression {……………….}

|FACT expression {……………….}

;

**Discussion:**

The input code is parsed using a bottom-up parser in this compiler. Because it is only built with flex and bison, this compiler is unable to provide original functionality such as if-else, loop, and switch case features. However, when creating code in this compiler-specific style, header declaration is not required but if we need we can use header file. The float variable always returns a value in the double data type, which is a compiler requirement. Any variable's string value is not stored by this compiler. With certain modifications, the code format supported by this compiler is similar to that of the C language. This compiler is error-free while working with the stated CFG format.

**Conclusion:**

Every programming language has required the use of a compiler. Designing a new language without a solid understanding of how a compiler works may be a challenging endeavor. Several issues were encountered during the design phase of this compiler, such as loop, if-else, switch case functions not working as they should owing to bison limitations, character and string variable values not being stored properly, and so on. In the end, some of these issues were resolved, and given the constraints, this compiler performs admirably.

**References:**

* <https://whatis.techtarget.com/definition/compiler>
* Principles of Compiler Design By Alfred V.Aho & J.D Ullman