

# **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:**

**Shujoy Kundu**

Roll: ***1907004***

3rd Year 2nd Semester

Department of Computer Science and Engineering, KUET

**Submitted To:**

**Nazia Jahan Khan Chowdhury**

Assistant Professor, Department of Computer Science and Engineering, KUET

**Dipannita Biswas**

Lecturer , Department of Computer Science and Engineering, KUET

**Submission date:**  November 21, 2023

**Objectives:**

* To know how to create different and new semantic and synthetic rules for the compiler.
* To know about shift and reduce policy of a compiler.
* To know about top down and bottom up parser and how they work.
* To build a parser generator using bison.

**Introduction:**

A compiler is a computer program that translates computer code written in one programming language into another language. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower-level language to create an executable program.

**Flex and Bison:**

FLEX (Fast Lexical analyzer generator) is a tool for generating scanners. Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. The tokens generated in this phase is then fed to the parser.

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. It is used to perform semantic analysis in a compiler. Parsing involves finding the relationship between input tokens. Bison is upward compatible with YACC(Yet Another Compiler Compiler) : all properly-written YACC grammars ought to work with Bison with no change.

Flex and Bison are tools are used together in the development of compilers and interpreters. Flex generates lexical analyzers, breaking down source code into tokens, while Bison generates parsers, analyzing the syntactic structure of the code based on a specified grammar. The two tools communicate through a token stream, with Flex providing tokens to Bison. This collaborative approach helps convert source code into a structured form, such as a parse tree, facilitating further processing for tasks like compilation or interpretation.

**Compilers with Flex and Bison:**

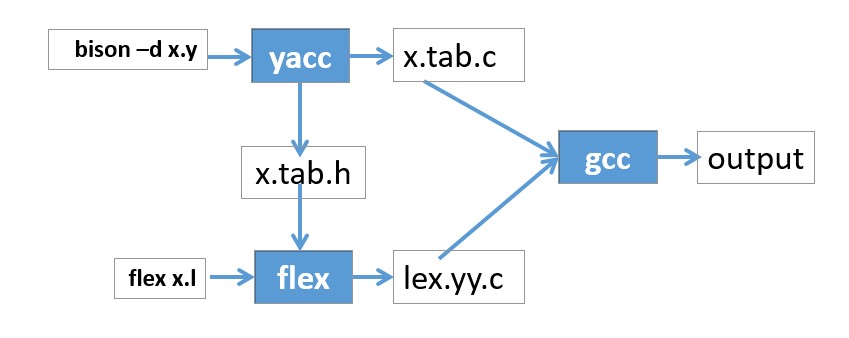


Figure-1.1: Workflow of bison and flex

**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 .

**Run the program in terminal**

1. bison -d -t 1907004.y
2. flex 1907004.l
3. gcc 1907004.tab.c lex.yy.c -o a
4. ./a

After running the program we will get a parser, and get the expected output in the output file.

**Project Description:**

**DATA TYPES:**

1. **Integer:**

Range: -2,147,483,648 to 2,147,483,647.

Tokens:

* + INT: Returned if regular expression for detecting integers match an expression.
  + INT: Returned if “var\_name” found for declaring an integer
  + Syntax:

int var\_name;

1. **Double:**

Range: 1.2E-38 to 3.4E+38 Tokens:

* + DOUBLE: Returned if regular expression for detecting floating point numbers match an expression
  + DOUBLE: Returned if “a” found for declaring a floating point number .
  + Syntax: double a;

1. **String:**

Range: a-z (small letters), A-Z (capital letters), 0-9 (digits) and symbols (: ” ”) Tokens:

* + STRING: Returned if regular expression for detecting strings match an expression § STRING: Returned if “name” found for declaring a string
  + Syntax: string name;

**VARIABLES:**

Range: a-z (small letters), A-Z (capital letters), 0-9 (digits). A variable name has to start with a small or a capital letter and can only be 31 characters long.

Type: A variable can contain either an Integer value or Double value or Character or a String.

Token:

* + ID: Returned if regular expression for detecting a variable name matched an expression.
  + Syntax: type var\_name;

**OPERATORS:**

Token: operator name in caps is returned for each operator.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operators** | **Data Type** | **Type** | **Description** | **Syntax** |
| + | Integer Double | Arithmetic | Adds two operands. | a + b |
| - | Integer  Double | Arithmetic | Subtracts second operand from the first. | a - b |
| \* | Integer Double | Arithmetic | Multiplies both operands. | a \* b |
| / | Integer Double | Arithmetic | Divides numerator by denumerator. | a / b |
| % | Integer | Arithmetic | Modulus  Operator and remainder of after an integer division. | a % b |
| <= | Integer Double | Relational | True if the value of left operand is greater than or equal to the value of right operand. | a <= b |
| >= | Integer Double | Relational | True if the value of left operand is less than or equal to the value of right operand. | a >=b |
| > | Integer Double | Relational | True if the value of left operand is greater than the value of right operand. | a>b |
| < | Integer Double | Relational | True if the value of left operand is less than the value of right operand. | a<b |
| == | Integer Double | Relational | True if the values of two operands are equal. | a==b |
| = | all | Assignment | Assigns value from the right to left | a=b |

**CONDITIONAL STATEMENTS;**

1. **IF-ELIF-ELSE :**

If(condition) {

Any number of operations

}

else if(condition) {

any number of operations

}

else {

Any number of operations.

}

1. **If-else:**

If(condition){

Some statements;

}

else {

Some statements;

}

1. If:

If (condition){

Some statements

}

**LOOPS:**

1. **FOR LOOP:**

Tokens:

* + FOR: Returned when “FOR” statement is found.

Syntax:

for ( expr1; expr2 ; expr3 ) {

Any number of statements

}

Suppose i is a loop control variable.

**expr1**: initial value of loop control variable ( i = 0 )

**expr2**: upper bound of the loop control variable ( i<10 )

**expr3**: the value by which loop control variables will

i = i + 1

1. **WHILE LOOP:**

Tokens:

* + WHILE: Returned if “WHILE” statement is found.

Syntax:

While( condition ) {

Any number of statements

}

**Input & Output:**

For input:

READ: Returned if “input” statement is found.

Syntax: read >> a;

For Output:

PRINT: Returned if “output” statement is found.

Syntax: print(a);

**Array:**

Syntax: int arr[10];

**Functions:**

FUNC: returned if “function” statement is found.

Syntax:

func function\_name (type name, type name…) : type {

any code;

}

**Dependency**

IMPORT: returned if “dependency” is found.

Syntax: import numpy;

**Comment:**

1. Single line comment:

Syntax: # this is a single line comment

1. Multi line comment:

Syntax: /multi/ this

Is a multiple line

Comment.

/multi/

**SYMBOLS:**

|  |  |
| --- | --- |
| **Symbol** | **TOKEN** |
| ( | LPAR |
| ) | RPAR |
| { | LCUR |
| } | RCUR |
| , | COMM |
| : | COL |
| Blank Space | No action taken |
| New Line(\n) | No action taken |
| Tab(\t) | No action taken |

**My Parser CFG:**

input:

|input line

;

line:

library

|declare

|assign SEMIC

|condition SEMIC

|if

|for

|while

|OutPut

|cin

|terminate

;

for:

FOR LP f\_first f\_second f\_third RP LCUR input RCUR {

cout << "\nParsing for loop\n";

}

;

while:

WHILE LP condition RP LCUR input RCUR {

cout << "\nParsing while loop\n";

}

;

f\_first:

declare

|assign SEMIC

;

f\_second:

condition SEMIC

;

f\_third:

assign

;

if:

IF LP condition RP LCUR input RCUR elseif {

cout << "\nparsing if block\n";

}

;

elseif:

|else

|ELSE IF LP condition RP LCUR input RCUR elseif {

cout << "\nparsing else if block\n";

}

;

else:

ELSE LCUR input RCUR {

cout << "\nparsing else block\n";

}

;

condition:

expr

|comparison

|logical

;

comparison:

expr EE expr

|expr NE expr

|expr GT expr

|expr LT expr

|expr LE expr

|expr GE expr

;

declare:

int\_declare

|double\_declare

|string\_declare

|boolean\_declare

|char\_declare

|func\_declare

|array\_declare

;

assign:

|arr\_assign

|int\_assign

|string\_assign

|char\_assign

|int\_assign COMMA assign

|string\_assign COMMA assign

|char\_assign COMMA assign

;

type:

INT {strcpy($$.sval,"int");}

|DOUBLE {strcpy($$.sval,"double");}

|STRING {strcpy($$.sval,"string");}

|CHAR {strcpy($$.sval,"char");}

|BOOL {strcpy($$.sval,"bool");}

|VOID {strcpy($$.sval,"void");}

;

expr:

term {

// only the int and double value

$$.dval = $1.dval;

$$.ival = $1.ival;

}

|MINUS expr %prec UMINUS {

$$.dval = $2.dval;

$$.ival = $2.ival;

}

|expr PLUS term {

$$.dval = $1.dval + $3.dval;

$$.ival = $1.ival + $3.ival;

}

|expr MINUS term {

$$.dval = $1.dval - $3.dval;

$$.ival = $1.ival - $1.ival;

}

;

term:

factor {

$$.dval = $1.dval;

$$.ival = $1.ival;

}

|term MUL factor {

$$.dval = $1.dval \* $3.dval;

$$.ival = $1.ival \* $1.ival;

}

|term DIV factor {

$$.dval = $1.dval / $3.dval;

$$.ival = $1.ival / $1.ival;

}

|term REM factor {

$$.ival = $1.ival % $3.ival;

$$.dval = $$.ival;

}

;

factor:

number {

$$.dval = $1.dval;

$$.ival = $1.ival;

}

data:

|INTD|DOUBLED|BOOLD|STRINGD|CHARD

|IDEN

|IDEN COMMA data

|INTD COMMA data

|DOUBLED COMMA data

|BOOLD COMMA data

|STRINGD COMMA data

|CHARD COMMA data

;

number:

INTD {

$$.dval = $1.dval;

$$.ival = $1.ival;

}

|DOUBLED {

$$.dval = $1.dval;

$$.ival = $1.dval;

}

|BOOLD {

$$.dval = $1.bval;

$$.ival = $1.bval;

}

;

**Features of this compiler**

1. import header files/ dependencies

2. Comments

3. Single/Multiple Character Variable declaration

4. IF ELSE-If Block

5. Variable assignment

6. Loop

7. Print function

9. Input function

10. Arrays

11. Function declaration and calling.

**Discussion:**

The project focuses on building a compiler using Flex and Bison. Flex tokenizes the source code, while Bison constructs an AST. The compiler verifies semantic rules, generates executable code, and incorporates robust error handling. Flex and Bison's modular approach streamlines the development process and enhances maintainability.

**Conclusion:**

The project showcases the practical application of Flex and Bison in compiler construction. By achieving the outlined objectives, the compiler transforms source code into executable programs, offering a comprehensive exploration of compiler design principles. This endeavor enhances understanding of programming languages and their translation into functional code.

**References:**

1. <https://whatis.techtarget.com/definition/compiler/>
2. <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>
3. <https://www.gnu.org/software/bison/manual/bison.html>
4. Principles of Compiler Design By Alfred V.Aho & J.D Ullman