KHULNA UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Compiler Design using flex and Bison

Project Report

Submitted by:

Mahin Rashid Chowdhury
1907021
Year : 3rd
Term : 2nd

Objectives

1. Implementation of the lexical analyzer using Flex
2. Identification and tokenization of various lexical elements, including keywords, identifiers, literals, and operators.
3. Implementation of grammar rules defined to capture the syntax of the programming language.
4. Describe the process of semantic analysis, checking for correctness in variable usage, type checking, and other semantic rules.
5. To use the gcc compiler and create a new programming language.

Introduction

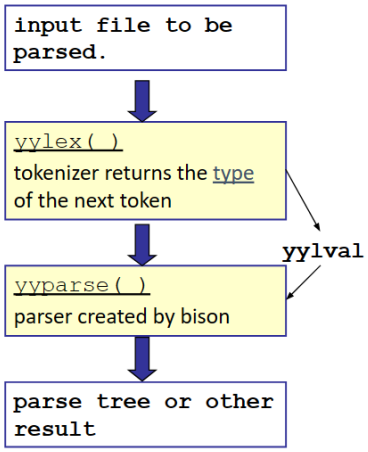
FLEX :

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. Scanners perform lexical analysis by dividing the input into meaningful units. For a C program the units are *variables*, *constants, keywords, operators, punctuation* etc. These units also called as tokens

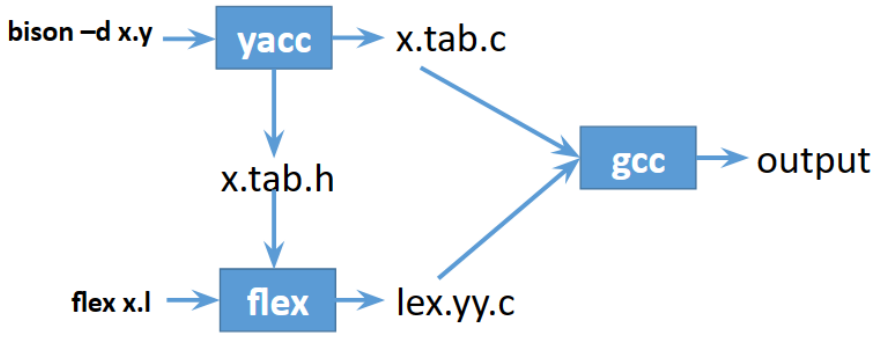
BISON :

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Bison is used to perform semantic analysis in a compiler. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Parsing involves finding the relationship between input tokens. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Interfaces with a scanner generated by Flex. Scanner is called as a subroutine when the parser needs the next token.

Flex and Bison are aging Unix utilities that help to write very fast parsers for almost arbitrary file formats. Flex and Bison will generate a parser that is virtually guaranteed to be faster than anything that could be written manually in a reasonable amount of time. Second, updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code. Third, Flex and Bison have mechanisms for error handling and recovery, finally Flex and Bison have been around for a long time, so they are far freer from bugs than newer code.



Compiler with Flex/lex and Yacc/Bison



Project Description

Header File :

import "stdio.h"
import "cse2k19.h"

Comment :

#This is a single line comment.

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
- Syntax: int a;

2. Float:

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

- FLOAT: Returned if regular expression for detecting floating point numbers match an expression
- Syntax: float f;

3. 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
- Syntax: char c;

Variable Declaration :

```
int a;
int b,c,d;
float f;
char str;
```

Variable Declaration with assignment :

```
char name = "Mahin",str;

float fl = 2.3456,num,pi;

int a = 3 , b = 7, c;
```

Error using a variable when It is not declared :

```
gh = 20; // Error : Variable not declared
```

Input function :

```
loadInt(a) // Will store an Int Variable
```

loadFloat(f) // Will store and float Variable
loadStr(s) // Will store String Variable

Output Print function :

fireVar(lab); // Will print the variable stored value
fireStr("Hello World"); // Will print the String Hello World
fireLn(); // will print a new line.

Operators :

Operators	Data Type	Type	Description	Syntax
+	Integer Float	Arithmetic	Adds two operands.	+(a,b)
-	Integer Float	Arithmetic	Subtracts second operand from the first.	-(a,b)
*	Integer Float	Arithmetic	Multiplies both operands.	*(a,b)
/	Integer Float	Arithmetic	Divides numerator by denumerator.	/(a,b)
%	Integer	Arithmetic	Modulus Operator and remainder of after an integer division.	%(a,b)
++	Integer	Arithmetic	Increment operator increases the integer value by one.	+(a,+)
--	Integer	Arithmetic	Decrement operator decreases the integer value by one.	-(a,-)
<=	Integer Float	Relational	True if the value of left operand is greater than or equal to the value of right operand.	<=(a,b)

>=	Integer Float	Relational	True if the value of left operand is less than or equal to the value of right operand.	>=(a,b)
>	Integer Float	Relational	True if the value of left operand is greater than the value of right operand.	>(a,b)
<	Integer Float	Relational	True if the value of left operand is less than the value of right operand.	<(a,b)
==	Integer Float	Relational	True if the values of two operands are equal.	==(a,b)
!=	Integer Float	Relational	True if the values of two operands are not equal	!=(a,b)

Mathematical Expressions with assignment :

```
int res = a + b;
res = a * b;
a = a/b;
res = a - b
```

Some Buit-in library functions :

```
gcd(a,b) // Return gcd of two values
max(n,m) // Return max of two values
min(n,m) // Return min of two values
is_prime(x) // Return if a number prime or not
```

For loop Statement :

- incrementing version :

```
for (1 to 5 inc 2){

  }
```

Output :

===>For Loop Incement Working Successfully
1
3
5
===>Loop executed 3 times

- Decrementing Version :

  for (m to n dec 2){

  }

  Output :

  ===>For Loop Decrement Working Successfully
  10
  8
  6
  4
  ===>Loop executed 4 times

Conditional Statement :

```
if ( p != q )
     {

     }
else if( p > q ){

}

else{

}
```

IF

```
if[condition] {
 any number of statements
}
```

IF-ELSE

```
if[condition] {
 any number of statements
}
else{
any number of statements
}
```

IF-ELSE IF-ELSE

```
if[condition]{
 any number of statements
}
else if[condition] {
any number of statements
}
else if[condition] {
any number of statements
}

else{
any number of statements
}
```

NESTED IF-ELSE IF-ELSE

```
if[condition] {
      if[condition] {
            any number of statements
       }
       else if {
       any number of statements
       }
}
else if[condition] {
       if[condition] {
          any number of statements
       }
       else if[condition] {
            any number of statements
       }
       else if {
             any number of statements

} }
else{
       if[condition] {
             any number of statements
       }
}
```

Output :

IF condition found
If condition is false.

Else If condition found
Else If condition is false.

ELSE condition found

Switch-Case Statement :

```
Switch(7)
        {
                1:
                        {

                        }
```

```
        7:
            {

            }

        default:
            {

            }
}
```

Output :

Switch Case found.
Executed 7 block case!

Function Call :

Function-name: Same as variable name. Token is ID which is returned if any expression matches the regular expression to identify a variable name.

- Without Parameters :

  ```
  func calc(){

  }
  ```

- With Parameters :

  ```
  func dp(x,y){

  }
  ```

SYMBOLS:

Symbol	TOKEN
(	*yytext returned
)	*yytext returned
{	*yytext returned

}	*yytext returned
,	*yytext returned
:	COL returned
Blank Space	No action taken
New Line(\n)	No action taken
Tab(\t)	No action taken

Project Run :

```
bison -d mahin.y
flex mahin.l
gcc mahin.tab.c lex.yy.c -o app
./app
```

Discussion

In this laboratory we have learned some of the basic techniques of compilers such as lexical analyzer , parser. Using them we could create a programming language of our own syntax using the gcc compiler. The lexical analysis phase was effective in tokenizing the source code, accurately recognizing and categorizing various lexical elements, such as keywords, identifiers, literals, and operators. Semantic analysis played a crucial role in ensuring the correctness of the source code, encompassing tasks such as variable usage and type checking. Furthermore, the code generation phase translated the intermediate representation into executable code, with implemented optimizations contributing to the efficiency of the compiled programs. On bison we could parse syntax in terms of our own rules and logics. At last, using the compiler we can output a output.txt file from input.txt according to the created programming language.

Conclusion

In conclusion, the successful completion of this compiler design project marks a significant achievement in the understanding and application of theoretical concepts in the real-world development of language processing tools. The project accomplished its objectives, producing a functional compiler capable of translating source code into executable programs. The iterative development process involved careful consideration of lexical and syntactic elements, implementation of semantic checks, and the generation of optimized code. Overall, this project has not only deepened our understanding of compiler construction but has also provided a valuable tool for developers in translating high-level language constructs into executable machine code.