Lab Report

Topic: Compiler Project

Course No: CSE 3211

Course Title: Compiler Design Laboratory

Submitted To

Dola Das

Lecturer

Md. Ahsan Habib Nayan

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Submitted By

Zahra Ferdous

Roll: 1707047

Department of Computer Science and Engineering

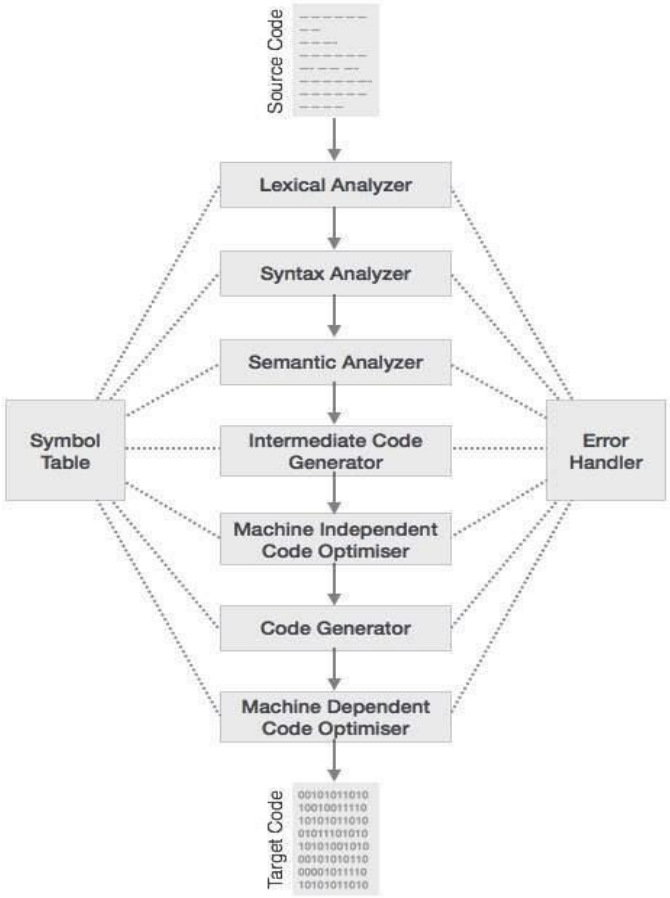
Khulna University of Engineering & Technology

Submission Date

15/06/21

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. It translates source code from a high-level programming language to a lower level language to create an executable program. A compiler is broken down in several parts.



The compiler that we are to create is only to be made with lexical analyzer and semantic analyzer and then compiled into C code. We will use Flex and Bison in the process.

Flex:

Flex is a computer program that generates lexical analyzers. It converts a sequence of characters or input steam into a sequence of tokens. Flex allows one to specify a lexical analyzer specifying regular expressions to describe patterns for tokens.

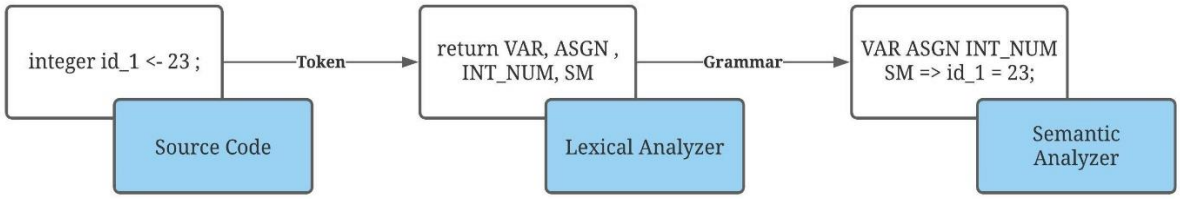
The input of the Flex tool is referred to as the Lex Language and the tool itself is the Lex Compiler. The Lex compiler transforms the input patterns into a transition diagram and generates code in a file called lex.yy.c, that simulates the transition diagram.

GNU 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 that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison by default generates LALR(1) parsers.

Description of the Compiler:

The compiler is divided into a lexical analyzer called syn.l and a parser sem2.y and by using the Flex and GNU Bison we create a file called lex.yy.c and sem2.tab.c that transforms the codes in C language and into an executable file called ex.exe.



syn.l tokenizes the input file and divides them in tokens that are predefined. The tokens used in the compiler are given below along with their input stream.

Tokens:

Input Streams	Tokens	Meanings
#start#	STRT	Start Symbol
#end#	END	End Symbol

integer	INT	Data Type : Int
real	DOUBLE	Data Type : double
text	TEXT	Data Type : char*
print	PRINT	Print Statements
arr	ARR	Array Identifier
if	IF	if
or if	ELIF	else if
else	ELSE	else
switch	SWITCH	switch
case	CASE	case
def	DEFAULT	default
break	BREAK	break
while	WHILE	While Do Loop Identifier
do	DO	While Do Loop Identifier
from	FROM	From To Loop Starting Iterator
to	TO	From To Loop Ending Iterator
skip	SKIP	From To Loop Iterator Skip Value
<add>	ADD	(+) Plus Operator
<sub>	SUB	(-) Minus Operator

<mul>	MUL	(*) Multiplication Operator
	DIV	(/) Division Operator
<mod>	MOD	(%) Modulus Operator
<power>	POWER	(^) Power Operator
<abs>	ABS	Absolute Value Function
<fact>	FACT	Factorial Function
<inc>	INC	(++) Increment Operator
<dec>	DEC	(--) Decrement Operator
<grt>	GRT	(>) Greater Than Operator
<les>	LES	(<) Less Than Operator
<equ>	EQU	(==) Equal To Operator
<not_equ>	NOT_EQU	(!=) Not Equal To Operator
<grt_equ>	GRT_EQU	(>=) Greater Than or Equal To Operator
<les_equ>	LES_EQU	(<=) Less Than or Equal To Operator
<or>	OR	(||) Binary Or Operator
<and>	AND	(&&) Binary And Operator
<not>	NOT	(!) Unary Not Operator

(	LP	Left Parenthesis
)	RP	Right Parenthesis
{	LB	Left Curly Bracket
}	RB	Right Curly Bracket
[	LI	Left Square Bracket
]	RI	Right Square Bracket
,	CM	Comma
;	SM	Semicolon
:	CLN	Colon
<-	ASGN	(=) Assignment Operator
[\n\t]*		White Spaces
/nl/	NL	New Line
[a-zA-Z]([a-zA-Z0-9_]){0,14}	VAR	Variable Names
\'[^'*"]*\'	STRING	String Values
[-]?[0-9]+	INT_NUM	Integer Values
[-]?[0-9]+[.][0-9]+	REA_NUM	Double Values
^**[^\n\r]+	SLC	Single Line Comment
[*][^*]+[*]	MLC	Multi Line Comment

The lexical analyzer will match these patterns and send the appropriate tokens and values to the compiler.

sem2.y will use these tokens and the grammar rules given there to run the given input code.

Grammar Rules

1. **Main Program**: #start# …..*statements*..... #end#
2. **Data Declarations**: there are 5 declarations. 3 for the datatypes and 2 for the arrays.
   integer a ;
   real b_1 , b_2 ;
   text Antora <- 'my name' , roll <- '1707047' ;
   arr integer stu[5] ;
   arr real gpa[60] ;
3. **Print:** This function can print a string, the value of an expression and the value of a variable.
   print ('This is a string') ;
   print (roll) ;
   print (3.4 2 <mod> 3) ;
4. **If:** The condition is an expression and if the expression is true then it will execute a statement that print either a string, the value of an expression and the value of a variable.
   if (3 <grt> 2) : { print ('yes') ; } ;
   if (3 <sub> 2) : { print (3 <sub> 2) ; } ;
   if (3) : { print (roll) ; } ;
5. **If Else:** If the condition expression in the if block is true then the print statement inside the if block will execute, else the print statement inside the else block will execute.
   if (1 <or> 0) : { print ('If Block') ; }
   else : { print ('Else Block') ; } ;
6. **If ... Or If … Else**: If follows with one or more Or Ifs with their own conditions. If one condition is met the print statement in that block is executed and no other condition is checked. If all conditions remain false, the statement in the else is executed.
   In this compiler there can be at most 10 Or Ifs but it can be extended.
   if (5 <grt> 5) : { print(1) ; }
   or if (5 <grt> 2) : { print(2) ; }
   or if (5 <grt> 3) : { print(3) ; }
   or if (5 <grt> 4) : { print(4) ; }

else { print(5) ; } ;

Here , the 2nd block will execute. Even though the next 2 condition is also true they will not execute.

7. **From ... To … Loop**: This loop will execute from an integer until it is iterated to another integer. And a print statement will execute that many number of times.

   from 3 to 6 : { print (‘lala’) ; } ;

   This statement will print ‘lala’ 4 times.

8. **From … To … Skip … Loop**: This loop will execute just as the above loop and it can skip 1 or more iterations.

   from 0 to 7 skip 2 : { print (5 <add> 3) ; } ;

   This line will print 8 3 times.

9. **From ... To … Loop with Array Operations**: This for loop can assign a value of an expression in an integer or real number array and can print the values of an array.

   from 0 to 4 : { stu[] <- 678 ; } ;

   This will assign 678 to the 0th to 4th index of stu array.

   from 0 to 4 : { print (stu[]) ; } ;

   This statement print the values of the 0th to 4th index of stu array.

10. **While … Do … Loop:** This while loop takes an integer variable as condition and with each iteration print a string or expression and decrement the value of that variable by 1 until it reaches 0.

    a <- 4 ;

    while a do : {

        print (‘yes’);

        <dec> a ;

        };

11. **Switch Case:** Switch takes an integer expression and the cases each have an integer option. If the option is matched with the switch condition then the print statement is executed. There can be one or more cases. And even if one case is executed the others can still be executed if their options also match the switch condition.

    There can be a default case as well. If all cases are false the default case is executed.

    switch 8 :

```
{
 case 2: {print(2); }
 case 8: {print(0); }
 case 8: {print(8); }
 def : {print(10);
};
```

This statement will print 0 and 8.

12. **Switch Case With Break:** it works just as the above one but if a statement is executed with a break statement within the block the next cases will not be executed.

```
switch 8 :

{

case 2: {print(2); break ;}

 case 8: {print(0); break; }

 case 8: {print(8); break; }

};
```

This statement will only print 0 as the break statement will execute and no farther case will be checked.

13. **Assignments:** After a variable is declared any time a value can be assigned in the variable, values can be overwritten as well. Values can be assigned to array variable in selected indices.

```
a <- 6<add>4 ;
b_1 <- a <div> 7 ;
roll <- '687' ;
stu[2] <- 5 ;
gpa[23] <- 3.30 ;
```

14. **Expressions:** There are 3 assignment expressions.

```
b_2   <-     45;
```

b_2 <- -0.76;

b_2 <- a;

b_2 <- gpa[23] ;

There are 2 function expressions.

b_2 <- <abs>-65.89089;

b_2 <- <fact>5;

There are 6 arithmetic expressions.

b_2 <- 0.2<add>4;

b_2 <- 6<sub>23.8;

b_2 <- 4<mul>55;

b_2 <- 8 3;

b_2 <- 7<mod>2;

b_2 <- 3<power>3;

There are 2 increment and decrement expressions. A postfix and a prefix one.

integer a1 <- 3 , a2 <- 3 , b1, b2 ,c1, c2;

b1 <- <inc> a1 ;

b2 <- a1 <inc> ;

a1 = 4 , b1 = 4 , a2 = 4 , b2 = 3

c1 <- <dec> a1 ;

c2 <- a2<dec> ;

a1 = 3 , c1 = 3 , a2 = 3 , c2 = 4

There is a unary not expression. It transforms a nonzero value to zero and vice-versa.

<not> a1;

a1 = 0

There are 6 relational expressions.

a1 <grt> b2;

a1 <grt_equ> b2;

a1 <les> b2;

a1 <les_equ> b2;

a1 <equ> b2;

a1 <not_equ> b2;

They return 1 or 0.

There are 2 binary expressions.

a1 <or> b1;

a1 <and> b1;

They also return 0 or 1. And follow the rules of logical OR and logical AND.

There is a expression with parenthesis.

(a1);

All expression is transformed to a real data type and there is a precedence.

( ) > <and> > <or> > <not_equ> > <equ> > <grt_equ> > <les_equ> > <grt> > <les> > <not> > <dec> > <inc> > <power> > <mod> > > <mul> > <sub> > <add> > <fact> > <abs>

Assign operator <- is right associative. All other operators are left associative.

15. **Single Line Comment**: All characters after ** is considered a comment and ignored by the compiler until a newline appears. It prints the line as it is.

    ** This is a Single Line Comment

16. **Multiple Line Comment**: All characters between /* */ is considered a comment and ignored by the compiler. It prints the line as it is.

    /* This is a

    Multi-

    Line

    Comment

    */

17**. Variables**: The variables of this compiler can be consisting of alphabets or digits and _ (underscore), has to be initiated by upper or lower case alphabet and can be utmost 15 characters long.

18. **Integer Number**: Both positive and negative integers are allowed.

19. **Real Number**: Only positive and negative numbers with or without decimal points are allowed.

20. **String**: Any character enclosed in ‘’ are considered a string. ‘ , “ , * , \ are not allowed as string characters.

21. **New Line**: Prints a new line in output file.

/nl/

Conclusion:

By doing this project we learn about the works of lexical analyzer, parser, syntax and semantics of a computer language. We get through knowledge about the grammar rules that are needed to make a language.