PROJECT REPORT

ON

COMPILER DESIGN

COURSE NAME: COMPILER DESIGN LABORATORY

COURSE NO: CSE 3212

DATE OF SUBMISSION: 21 November, 2023

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

SUBMITTED BY

Mst. Shafiatun Nur Shimu

Roll : 1907001

Year : 3rd

Semester : 2nd

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY, KHULNA

Objectives:

1. To design and implement a new programming language using Flex and Bison.
2. Design and implement a lexical analyzer to tokenize the input source code.
3. Create a parser to check the syntax of the input program.
4. Ensure accurate lexing and parsing including error handling for unexpected inputs.
5. Detect conflicts within the grammar rules and resolve them.

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 scanner generated by Flex. Scanner called as a subroutine when 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 write 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 far freer from bugs than newer code.

Compiler with Flex and Bison

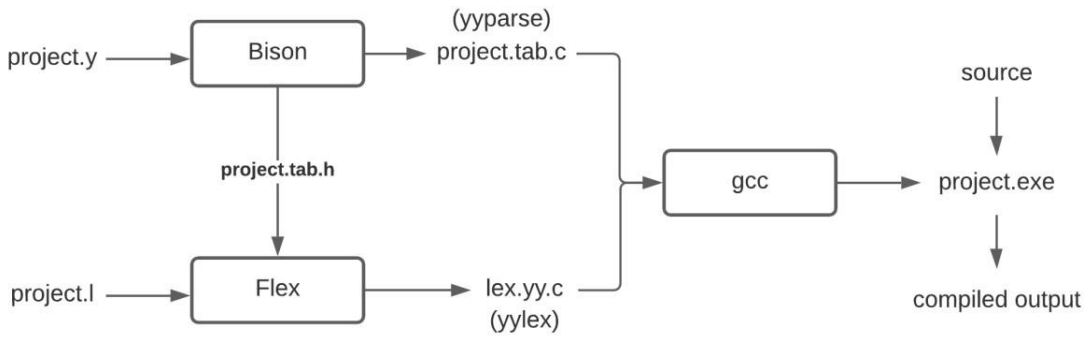

Fig: A diagram of how a compiler build with flex and bison works

Commands to create compiler:

Here *1907001.y* is the bison file and *1907001.l* is the lex file. *bison –d 1907001.y flex 1907001.l*

gcc –o app 1907001.tab.c lex.yy.c

After running these commands on command prompt, an executable file named *app.exe* will be created.

Then *./app* to run. All of these commands were written in a Makefile. To we don't need to write these commands repeatedly.

Project Description:

Topic	**Description**
Header	Syntax: @insert["{id}".set] Here id matches for [_A-Za-z][_A-Za-z0-9]* Example: @insert[stdio.set] @insert[stdlib.set]

Base Function	Similar as main() in C Syntax: base_function : ==> Function body <==
User defined Function	Similar as user defined functions in C. Must be declared before the base function Syntax: func : return type function_name (datatype para1, datatype para2,..) ==> <== Example: func : int ABC (int xxx, dbl a) ==> <==
Function call	Syntax: call function_name (parameters) # Example: call function1 (xxx, a) #

Topic	Description
Data types	**int(Integer):** Default data type. Similar to C int data type. **dbl(double):** Similar to C double data type **char(character):** Similar to C char data type **string(String)**: Similar to C character array.
Variables declaration	**Syntax:** var : datatype variable_name # **Example:** var : int a #
Variables initialization	Variable can be initialized while declaring. Syntax: var : datatype variable_name = value #

	example: var : int a = 20 #
Multiple variable declaration and initialization	Multiple variable can be declared and also can be initialized. Syntax : var : datatype variable_name1,var2 = value ,var3# *Example: var : int v1 = 10,v2=20,v3 #*
Value Assignment	Value can be assigned to any declared variable. Syntax: variable_name = expression # *Example: var : int v1 = 10,v2=20,v3 #* *V3 = v1+v2 #* *V3=30 #* *V3=v1 #* *etc*

Basic features:

Topic	Description
Comment	Single line comment: starts and ends with "!!" *Example: !! This is a single line comment !!* Single line comment: starts with "!*" ends with "*!" *Example: !* This is a multiple* *line comment *!*
Print statement	Syntax: show(variable_name) # show("any string") # show("any string.."@variable_name) # *int a= 10#* *show("value of a = "@a) #*
Scan statement	Syntax: scan(variable_name) # *Int a#* *scan (a) #*

SYMBOLS:

Symbol	TOKEN
#	ENDLINE
==>	START
<==	END
=	ASSIGN
(	LFP
)	RFP
{	LSP
}	RSP
[	LTP
]	RTP
,	COMMA
:	SIGN
@	AT
&	AND
|	OR
^	NOT
++	INCREAMENT
--	DECREMENT
<	LT
<=	LE
>=	GE
>	GT
given	DEFAULT
throw	RETURN
skip	CONTINUE
halt	BREAK
Blank Space	No action taken
New Line(\n)	No action taken
Tab(\t)	No action taken

OPERATORS:

Suppose A and B are two integers:

Operators	Data Type	Type	Description	Syntax
jog	Integer	Arithmetic	Adds two operands.	A jog B
biyog	Integer	Arithmetic	Subtracts second operand from the first.	A biyog B
gun	Integer	Arithmetic	Multiplies both operands.	A gun B
vag	Integer	Arithmetic	Divides numerator by denumerator.	A vag B
mod	Integer	Arithmetic	Modulus Operator and remainder of after an integer division.	A mod B
++	Integer	Arithmetic	Increment operator increases the integer value by one.	A++
--	Integer	Arithmetic	Decrement operator decreases the integer value by one.	A--
<=	Integer	Relational	True if the value of left operand is greater than or equal to the value of right operand.	A<=B
>=	Integer	Relational	True if the value of left operand is less than or equal to the value of right operand.	A>=B
>	Integer	Relational	True if the value of left operand is greater than the value of right operand.	A>B

<	Integer	Relational	True if the value of left operand is less than the value of right operand.	A<B
==	Integer	Relational	True if the values of two operands are equal.	A==B
!=	Integer	Relational	True if the values of two operands are not equal	A!=B
&	Integer	logical	True if the bitwise AND of two operands are equal	A & B
|	Integer	logical	True if the bitwise OR of two operands are equal	A|B
^	Integer	logical	True if the value is zero or false	^A

CONDITIONAL STATEMENTS:

Whether-else: this can be applied on integer only.

Topic	**Description**
Whether	Similar as If condition in C. Syntax: *whether [condition]* *{* *Statements #* *}*
Whether - else	Similar as if-else condition in C Syntax: *whether [10==100]* *{* *statements* *}* *else*

	{ *Statements* *}*
Whether -else whether ladder structure	Similar as else-if ladder structure in C. Syntax: whether [condition1] *{* *a-- #* *}* *else whether [condition]* *{* *a-- #* *}* *else whether [condition]* *{* *a-- #* *}* *else* *{* *a=a gun 2 #* *}*
Nested whether-else	Similar as nested if-else condition in C. Syntax: *whether [10 > 100]* *{* *a = 5 #* *whether [10 > 100]* *{* *a-- #* *}* *else whether [10 > 50]* *{* *a-- #* *}* *else whether [10 > 10]* *{* *a-- #*

	} *else* *{* *a=a gun 2 #* *}* *}* *else* *{* *a= a jog 2 #* *}*

2. SWITCH:

Similar as switch-case condition in C. Here cases can be any expression of integer. If no case matches then default case.

Topic	**Description**
Toggle-value	Syntax: *toggle[expression]{* *value 1 :* *{* *var : int line #* *halt #* *}* *value 10 :* *{* *halt #* *}* *value 1000:* *{* *halt #* *}* *given :* *{* *}* *}*

LOOPS:

Topic	**Description**
loop	This is similar as for loop in C. nested looping allowed. Syntax: *loop [value assign : condition check : increment/decrement/other expression]* *{* *xxx=100 #* *whether [xxx == 100]* *{* *//whether condition* *}* *loop [k = 1 : k<5 : k++]* *{* *Internal loop* *}* *}*
while	Similar as while loop in C. Syntax: *while[condition]* *{* *a++ #* *whether[a<2]* *{* *a++ #* *}* *}*

ARRAYS:

Arrays can only be of integer type.

Topic	**Description**
Declare array	Syntax: var : daratatype array_name[size] # Example: var : int arr1 [3] #
Assignment	Only declared array can be assigned values. Syntax: array_name = [elements…] Example: arr1=[1,2,3]

BUILT IN FUNCTIONS:

Function	**syntax**	**token**	**description**
Sine function	sin(expression)	SIN	The sine value of the angle in degrees.
Cosine function	cos(expression)	COS	The cosine value of the angle in degrees.
Tangent function	tan(expression)	TAN	The tangent value of the angle in degrees.
Logarithm	log(expression)	LOG	The 10 based logarithm value of the input.

ln function	ln(expression)	LON	The e based logarithm value of the input.
Square root function	sqrt(expression)	SQRT	Returns square root value of expression.
Power function	pow(expression, expression)	POW	Returns expr1 raised to the power expr2.
Factorial	factorial(expression)	FACTORIAL	Returns factorial value of expression.

Precedence:

Operations	Associativity
pow gun, vag, mod jog, biyog ==, !=, <,<=, >, >= &, |, ^ Sin(), cos(), tan()	**left**

Sample Input and Output:

Input	Output
@insert[stdio.set] @insert[stdlib.set]	return value User defined function ===> functhkjk Variable declared

func : int functhkjk (int xxx, dbl a)	
==>	Value assigned
throw xxx #	
<==	Value assigned
base_function :	
==>	Variable declared
var : int a,b #	
b=20 #	character Value assigned
a=10 #	
var : char c #	Variable declared
c='C' #	
var : string str1 #	string Value assigned
str1="ABC" #	
	Addition: 10 + 20 = 30
var : int d=a jog b #	variable initialized
show(d)#	
	Print: 30
var : int arr1 [3] #	
arr1= [1,2] #	Array declared
!!operations!!	Value assigned to array
var :int sq = pow(2, 3) #	this is single line comment
sq = log(1000) #	Power: 2 ^ 3 = 8
sq = ln(a++) #	
sq = sqrt(5) #	variable initialized
sq = factorial(5) #	
sq = sin(30) #	log(1000) = 3.000000
sq=a2 & a3 #	Value assigned
sq=a2 | a3 #	
sq= ^a2 #	ln(11) = 2.397895
a++ #	Value assigned
a-- #	
a = a jog (4 biyog 15) vag 3 mod 2 #	Square Root of 5 = 2.2361

!! function call !! call functhkjk (b, a) #	Value assigned

!! condition!!
whether [10 > 100]
{
a = 5 #
whether [10 > 100]
{
a-- #
}
else whether [10 > 50]
{
a-- #
}
else whether [10 > 10]
{
a-- #
}
else
{
a=a gun 2 #
}
}
else
{
a= a jog 2 #
}
	Factorial of 5 = 120
	Value assigned
	sin(30) = 0.50
	Value assigned
	logical AND: 0 & 0 = False
	Value assigned
	logical OR: 0 | 0 = False
	Value assigned
	logical NOT: ^0 = True
	Value assigned
	Value of expression = 11
	Value of expression = 9
	Subtraction: 4 - 15 = -11
	Division: -11 / 3 = -3

!!switch!! toggle[100] { value 1 :	Modulus: -3 mod 2 = -1 Addition: 10 + -1 = 9 Value assigned

{	
var : int line #	this is single line comment
halt #	calling function
}	
value 10 :	this is single line comment
{	Greater Than: 10 > 100?
halt #	No, 10 Not Greater
}	
value 1000:	Value assigned
{	
halt #	Greater Than: 10 > 100?
}	No, 10 Not Greater
given :	
{	Value of expression = 4
}	Greater Than: 10 > 50?
	No, 10 Not Greater
}	Value of expression = 4
!!for loop !!	Greater Than: 10 > 10?
loop [j = 30 : j > 10 : j=j biyog 7]	No, 10 Not Greater
{	
xxx=100 #	Value of expression = 4
whether [xxx == 100]	
{	Multiplication: 5 x 2 = 10
xxx=600 #	Value assigned
}	
loop [k = 1 : k<5 : k++]	else Condition true.
{	
	Addition: 10 + 2 = 12
}	Value assigned
}	else Condition true.

!!while loop!!
while[a<6]
{
  a++ #
  whether[a<2]
  {
    a++ #
  }
 	
}
 	
<==
	this is single line comment
	Variable declared
	
	matched for default case
	
	--- Switch Case ---
	
	this is single line comment
	Variable not declared
	
	Equal: 0 == 100?
	No, Not Equal
	
	Variable not declared
	
	whether Condition not true.
	
	Expression value = 1
	Expression value = 2
	Expression value = 3
	Expression value = 4
	
	--- For Loop Ends ---
	
	Expression value = 30
	Expression value = 23
	Expression value = 16
	
	--- For Loop Ends ---
	
	this is single line comment
	Less Than: 12 < 6?
	No, 12 Not Less

	Value of expression = 13 Less Than: 12 < 2? No, 12 Not Less Value of expression = 13 whether Condition not true. --- While Loop Starts --- --- While Loop Ends --- main function dependency added dependency added

Discussion:

While doing this project, various challenges and complexities were faced. Lots of errors and corner cases arised which were sincerely noticed and overcome. Designing the grammar rules and ensuring that the language was unambiguous was a critical aspect. We had to refine and modify the language's grammar multiple times to ensure its correctness and coherence.

Implementing the lexical analysis and parsing phases using Flex and Bison demanded a thorough understanding of tokenization, grammar specification, and AST generation. Debugging and error handling were essential in identifying and resolving issues within the language structure.

Conclusion:

After doing this project, it can be said that each step of implementing a compiler is very important and also sensitive. Every step related to another and sequential.This experiment was

an enriching experience in understanding the intricacies of language design and implementation using Flex and Bison. Building a programming language from scratch provided valuable insights into the underlying principles of lexical analysis and parsing. Despite the challenges faced, the successful creation of a basic language serves as a testament to the power and utility of Flex and Bison in language development.

References:

1. **https://stackoverflow.com/questions/tagged/flexbox?tab=Newest**
2. **https://stackoverflow.com/questions/tagged/bison**
3. **https://www.youtube.com/watch?v=U0hkkGy6_ig&t=1360s**
4. **https://www.youtube.com/watch?v=fFRxWtRibC8**