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

Project Report

A Simple Compiler Using Bison and Flex

Course No: **CSE 3212**
Course Title: **Compiler Design Laboratory**

Submitted By

Md Arefin Molla
Roll: **2007050**
Batch: **2k20**
Year: **3rd**, Semester: **2nd**

Submission Date: 14/01/2025

Table of Contents

Introduction

A compiler is a software tool responsible for translating source code written in a higher-level programming language into a lower-level language, such as assembly or machine code, to generate an executable program. It undergoes a series of six phases, each transforming the source program from one representation to another. These phases work sequentially, with each phase taking input from the previous one and producing output for the subsequent phase. Together, these phases facilitate the conversion of high-level language code into machine code. The phases of a compiler are:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator

In our project, we will implement the first three phases of a compiler using Flex and Bison.

Flex

FLEX, which stands for Fast Lexical Analyzer Generator, is a software tool developed by Vern Paxson in C around 1987. It serves the purpose of generating lexical analyzers, also known as scanners or lexers. FLEX is a modern alternative to the classic lex, originally created by Mike Lesk and Eric Schmidt in 1975, and it has become the standard lexical analyzer generator on many Unix systems.

In a FLEX program, we typically provide a list of regular expressions (regexps) along with instructions specifying what actions to take when the input matches any of these expressions. These actions are defined in the program. When a FLEX-generated scanner processes its input, it scans through the text, comparing it to the specified regexps and executing the corresponding actions for each match.

Basic Structure:

```
%{
        // Definitions
%}

%%

        Rules

%%

User code section
```

1. Definition Segment: The definition segment entails the declaration of variables, regular definitions, and manifest constants. This segment is identified by text enclosed in "%{ %}" brackets.
2. Rule Segment: The rule segment consists of a set of rules formatted as pattern-action pairs, where both the pattern and action appear on the same line within {} brackets. The entire segment is enclosed by "%% %%".
3. User Code Segment: In this segment, C statements and additional functions are included. These functions can be compiled separately and integrated into the lexical analyzer.

Bison:

Bison, a parser generator within the GNU Project, analyzes a specification of a context-free language, highlights parsing ambiguities, and constructs a parser (in C, C++, or Java). This parser reads sequences of tokens and determines whether they conform to the grammar's specified syntax. Importantly, the parsers generated by Bison are portable and don't rely on particular compilers. While Bison defaults to producing LALR(1) parsers, it also has the capability to generate canonical LR, IELR(1), and GLR parsers.

Basic Structure:

```
%{
        C Declaration
%}
        Bison Declaration
%%

         Grammar Rules

%%

Additional C code.
```

1. Declaration Segment in C: This segment encompasses the declaration of variables, constants, and additional C functions required within the context-free grammar (CFG) rules. The content is enclosed within "%{ %}" brackets.
2. Bison Definitions: The Bison declaration includes definitions specific to Bison, such as token declarations, type declarations, and other necessary elements.
3. Rules of Grammar: In this segment, one finds the context-free grammar (CFG) rules that define the structure of the language.
4. Supplementary C Code: This segment incorporates C statements and additional functions that complement the overall functionality of the system.

How Flex and Bison Works Together:

Flex and Bison collaborate by following a coordinated process. Bison reads the grammar descriptions from a .y file and produces a syntax analyzer (parser) in a file named .tab.c, incorporating the yyparse function. The .y file also includes token declarations, and when the -d option is utilized, Bison generates token definitions in a .tab.h file. On the other hand, Flex reads pattern descriptions from a .l file, includes the .tab.h file, and generates a lexical analyzer with the yylex function in a file called lex.yy.c. Subsequently, the lexer and parser are compiled and linked together to generate an executable file (.exe). In the main program, the yyparse function is invoked to initiate the compiler. Internally, yyparse automatically invokes yylex to retrieve each token, facilitating the parsing process.

Compiler Description

Tokens:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming languages, keywords, constants, identifiers, strings, numbers, operators and punctuation symbols can be considered as tokens.
Tokens used in this compiler are described in the following table:

#	Token	Input String	Description
1	FUNC	func	Indicates the start of declaring a function.
2	PRINT	print	Used for printing.
3	SWITCH	switch	Indicates the start of a switch block.
4	CASE	case	Indicates the cases of a switch block.
5	DEFAULT	def_ault	Indicates the default case of a switch.
6	ASIN	asin	Inverse sine function.
7	ACOS	acos	Inverse cosine function.
8	ATAN	atan	Inverse tangent function.
9	LOG10	Log10	Logarithm function with base 10.
10	LOG	log	Logarithm function.
11	SIN	sin	Sine function.
12	COS	cos	Cosine function.
13	TAN	tan	Tangent function.
14	GCD	gcd	Performs the greatest common divisor (GCD) operation.
15	LCM	lcm	Performs the least common multiple (LCM) operation.
16	POW	pow	Performs the power operation pow(a, b).
17	IMPORT	import	Used for importing header files.

18	VAR	var	Declares a variable.
19	CLASS	class	Declares a class.
20	INIT	init	Initializes a class.
21	INT	[0-9]+	Indicates an integer number.
22	DOUBLE	[0-9]+(\.[0-9]+)?	Indicates a double-precision floating-point number.
23	STRING	\"(\\.|[^"\\])*\"	Indicates a string literal.
24	ELSEIF	elseif	Specifies a conditional block after an if.
25	ELSE	else	Executes when previous if or elseif conditions do not match.
26	IF	if	Indicates a conditional block.
27	FOR	for	Declares a for loop.
28	WHILE	while	Declares a while loop.
29	CONST	const	Indicates a constant variable.
30	CONTINUE	continue	Indicates a continue statement in a loop.
31	RETURN	return	Return statement.
32	VOID	void	Specifies a return type of void.
33	CLASS	class	Declares a class.
34	DO	do	Indicates the do part of a do-while loop.
35	RIGHT_ASSIGN	>>=	Right shift assignment operator.
36	LEFT_ASSIGN	<<=	Left shift assignment operator.
37	ADD_ASSIGN	+=	Addition and assignment operator.
38	SUB_ASSIGN	-=	Subtraction and assignment operator.
39	MUL_ASSIGN	*=	Multiplication and assignment operator.

<table>
<tr><td>40</td><td>DIV_ASSIGN</td><td>/=</td><td>Division and assignment operator.</td></tr>
<tr><td>41</td><td>MOD_ASSIGN</td><td>%=</td><td>Modulus and assignment operator.</td></tr>
<tr><td>42</td><td>AND_ASSIGN</td><td>&=</td><td>Bitwise AND and assignment operator.</td></tr>
<tr><td>43</td><td>XOR_ASSIGN</td><td>^=</td><td>Bitwise XOR and assignment operator.</td></tr>
<tr><td>44</td><td>OR_ASSIGN</td><td>|=</td><td>Bitwise OR and assignment operator.</td></tr>
<tr><td>45</td><td>RIGHT_OP</td><td>>></td><td>Right shift operator.</td></tr>
<tr><td>46</td><td>LEFT_OP</td><td><<</td><td>Left shift operator.</td></tr>
<tr><td>47</td><td>INC_OP</td><td>++</td><td>Increment operator.</td></tr>
<tr><td>48</td><td>DEC_OP</td><td>--</td><td>Decrement operator.</td></tr>
<tr><td>49</td><td>AND_OP</td><td>&&</td><td>Logical AND operator.</td></tr>
<tr><td>50</td><td>OR_OP</td><td>||</td><td>Logical OR operator.</td></tr>
<tr><td>51</td><td>LE_OP</td><td><=</td><td>Less than or equal to operator.</td></tr>
<tr><td>52</td><td>GE_OP</td><td>>=</td><td>Greater than or equal to operator.</td></tr>
<tr><td>53</td><td>EQ_OP</td><td>==</td><td>Equality operator.</td></tr>
<tr><td>54</td><td>NE_OP</td><td>!=</td><td>Not equal to operator.</td></tr>
<tr><td>55</td><td>;</td><td>;</td><td>Indicates the end of a line.</td></tr>
<tr><td>56</td><td>-></td><td>-></td><td>Arrow operator (used as a helper operator).</td></tr>
<tr><td>57</td><td>(</td><td>(</td><td>Open parenthesis.</td></tr>
<tr><td>58</td><td>)</td><td>)</td><td>Close parenthesis.</td></tr>
<tr><td>59</td><td>=</td><td>=</td><td>Assignment operator.</td></tr>
</table>

Context-Free Grammars(CFG):

Context-free grammars (CFGs) are employed to articulate context-free languages, serving as a collection of recursive rules designed to generate string patterns. While a context-free grammar encompasses the capacity to describe all regular languages and more, it falls short of capturing the entirety of possible languages.
The generation of context-free languages by context-free grammars involves utilizing a set of variables defined recursively in terms of one another through a series of production rules. The term "context-free" is aptly applied because any production rule within the grammar can be applied independently of the surrounding context. The application of a rule is not contingent upon the presence or absence of other symbols around the symbol to which the rule is being applied.

Context-Free Grammars used in Compiler:

```
starthere   :
                | function starthere
                | declaration starthere
                | classgrammer starthere
                | importgrammer starthere
                ;
classgrammer    : CLASS NAME '{' statement '}' ';'
                ;

function    :   FUNC NAME '(' fparameter ')' PTR_OP TYPE '{' statement '}'

TYPE        :   INT
                | DOUBLE
                | STRING
                | VOID
                | NAME
                ;
fparameter :
        | NAME ':' TYPE fsparameter
        ;
fsparameter :
        | ',' NAME ':' TYPE fsparameter
        | ',' NAME ':' TYPE
        ;
statement   :
        | ifgrammer statement
        | declaration statement
        | forgrammer statement
        | asgngrammer statement
        | whilegrammer statement
        | mathexpr statement
        | dowhilegrameer statement
        | returnstmt statement
        | printgrammer statement
        | switchgrammer statement
        ;
```

```
switchgrammer   :       SWITCH '(' expression ')' ':' '{' casegrammer '}'
                ;

casegrammer     :       CASE expression ':' statement casegrammer
                | def_ault ':' statement
                        ;

mathexpr    :   expression ';'
                ;


ifgrammer   :   IF '(' expression ')' '{' statement '}' elsifgrmr
                ;
expression  :   NUM
                | STR
                | VARACCESS
                | SIN
                | ASIN
                | COS
                | ACOS
                | TAN
                | ATAN
                | LOG
                | LOG10
                | POW
                | LCM
                | GCD
                | expression '+' expression
                | expression '-' expression
                | expression '*' expression
                | expression '/' expression
                | expression LE_OP expression
                | expression GE_OP expression
                | expression '<' expression
                | expression '>' expression
                | expression EQ_OP expression
                | expression NE_OP expression
                | '(' expression ')'
                | | expression AND_OP expression
                | expression OR_OP expression
                | expression "&" expression
                | '!' expression
                | '~' expression
                | expression '^' expression
                | expression '|' expression
                ;
returnstmt  :   RETURN mathexpr
                | return ';'
                ;


declaration :   VAR varriables ';'
```

```
varriables :     varriable ',' varriables
                 | varriable
                 ;
varriable  :     NAME ':' TYPE
                 | NAME '=' expression
                 | NAME ':' arraydim '*' '(' expression ')'
                 | NAME ':' arraydim

arraydim    :    '[' arrayx ']'
                 ;

arrayx     :     TYPE
                 | '[' arrayx ']'
                 ;
elsifgrmr  :
                 | ELSEIF '(' expression ')' '{' statement '}' elsifgrmr
                 | ELSE '{' statement '}'
                 ;

asgngrammer :          ASGNVAR ',' asgngrammer
                 | ASGNVAR ';'
                 ;

forgrammer  :  FOR '(' forassign ';' expression ';' forassign ')' '{' statement '}'
                 ;

forassign   :    ASGNVAR ',' forassign
                 | ASGNVAR
                 ;

ASGNVAR     :          VARACCESS '=' expression
                 | VARACCESS INC_OP
                 | VARACCESS DEC_OP
                 | VARACCESS ADD_ASSIGN expression
                 | VARACCESS SUB_ASSIGN expression
                 | VARACCESS MUL_ASSIGN expression
                 | VARACCESS DIV_ASSIGN expression
                 | VARACCESS MOD_ASSIGN expression
                 | VARACCESS AND_ASSIGN expression
                 | VARACCESS XOR_ASSIGN expression
                 | VARACCESS OR_ASSIGN expression
whilegrammer    :      WHILE '(' expression ')' '{' statement '}'

dowhilegrameer  :      DO '{' statement '}' WHILE '(' expression ')' ';'

printgrammer    :      PRINT '(' manyexprgm ')' ';'
manyexprgm      :      expression ',' manyexprgm
                       | expression
                       ;
importgrammer   :      IMPORT manyname ';'
                       ;
```

Main Features:

- **Beginning of Program**: The program can start from import or function declarations.

- **Variable Declaration**: There can be three types of data:
    1. Integer: A number without any fractional component.
    2. Double: A number with the fractional component.
    3. String: A sequence of characters terminated with a null character \0.

  To declare a variable at first we need to tell the data type. Then we can name our variables. A variable name can start with a letter followed by any number of letters and numbers. A variable can be also initialized during declaration.

  Code Snippet:

```
var xx:Int;
var a:Int,b=10,c:[Int],Dd:[[[Int]]]*(100);
var aa=10,cc=10.01;
```

- **Array Declaration:** An array is defined as the collection of similar types of data items stored at contiguous memory locations. To declare an array we first need to tell its data type.

```
var ar[100] - 1D array of size 100
var arr[[100]] - 2D array of size 100
```

- **Printing Output**: We can also print output.

  Code Snippet:

```
print(&a)
```

- **Conditional Statements**: Conditional Statements are used to make decisions based on the conditions. There are four types of conditional statements in this language:

```
if( condition ){

//code
.
.
.
      if( condition )
      {
      // code
       }
else if( condition ){
// code

}
.
.
.
} else if( condition ) {

//code
  if( condition )
 {
//code
 }
}
.
.
.
} else {

//code
}
```

- **Looping Statements:** Sometimes we need to execute a block of codes several times.A loop statement allows us to execute a statement or group of statements multiple times. There are three types of looping statements in the described language:

1. For Loop: Basic syntax for this loop is:

```
for( assignment ; expression ; assignment) {
      //code
      for( assignment; expression ; assignment)
      {
         //code
      }
}
```

2. Do-while Loop: This loop is almost similar to while loop. But instead of checking the condition before executing codes,the condition here is checked after executing codes. Basic syntax

```
DO {

        //code

 }while
```

3. While Loop: Basic syntax for while loop:

```
while ( expression ){
              //code
}
```

Switch statement: The switch statement allows us to execute one codeblock among many alternatives. The same thing can be done with the if...elif ladder. However, the syntax of the switch statement is much easier to read andwrite.

Basic syntax:

```
switch( condition ):
  {
     case x:
             // code
     case y:
             // code
             .
             .
             .
             .
             .
             .
     case default:
             //code
  }
```

- **User-Defined Modules**: A module is a block of code that performs a specific task.We can define modules according to our needs. These modules are known as user-defined functions.

  The basic syntax for module declaration is:

  ```
  func func_name( func_parameter1_name: TYPE , func_parameter2_name: TYPE , ...) ->
  return_type {
       // statements
  }
  ```

- **Numeric Operations**: As a general language different numeric operation can beperformed such as- trigonometric function(sine,cosine,tangent),logarithmic functions(log,log2,ln) etc.

Conclusion

The compiler, created solely with flex and bison and lacking data structures, cannot facilitate jumps, thus limiting the implementation of if-else statements or loops. To enable such functionalities, integrating a data structure like an Abstract Syntax Tree (AST) becomes necessary. However, the compiler performs well for linear execution.

References:

1. flex & bison - John R. Levin
2. LEX & YACC TUTORIAL - Tom Nieman
3. Compiler - Wikipedia
4. Flex (Fast Lexical Analyzer Generator) - GeeksforGeeks
5. GNU Bison - Wikipedia
6. Context Free Grammars | Brilliant Math & Science Wiki
7. Source Code - Arefin-Nahid/Compiler-Project