Project Report

CSE 3212: Compiler Design Laboratory

A Simple Compiler Using Bison and Flex

Submitted by:

Md. Mehrab Hossain Opi

Roll: 1707002

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering & Technology (KUET)

Table of Contents

Introduction

In computing, a compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower-level language (e.g., assembly language, object code, or machine code) to create an executable program.

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.
There are 6 phases in a compiler. Each of these phases helps in converting the high-level language to 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.

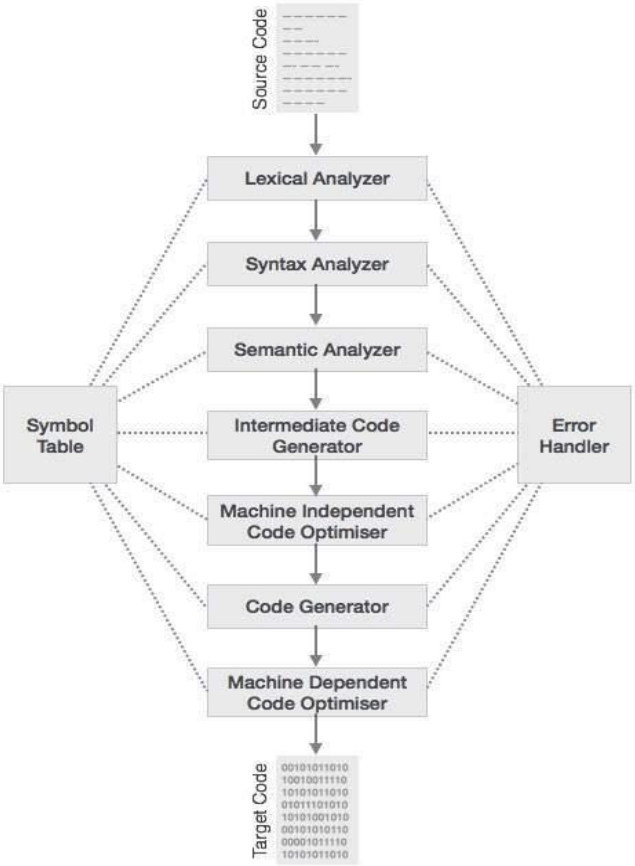

Figure 1: Phases of Compiler

Flex

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is the modern replacements for the classic lex. Lex, originally written by Mike Lesk and Eric Schmidt and described in 1975, is the standard lexical analyzer generator on many Unix systems.

A flex program basically consists of a list of regular expressions(regexps) with instructions about what to do when the input matches any of them, known as actions. A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.

Basic Structure:

```
%{
        // Definitions
%}

%%

        Rules

%%

User code section
```

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, the text is enclosed in “%{ %}” brackets.

2. Rules Section: The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in “%% %%”.

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load them with the lexical analyzer.

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 (either in C, C++, or Java) that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. The generated parsers are portable: they do not require any specific compilers. Bison by default generates LALR(1) parsers but it can also generate canonical LR, IELR(1), and GLR parsers.

Basic Structure:

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

         Grammar Rules

%%

Additional C code.
```

1. C Declaration Section: This section contains the declaration of variables, constants, and other c functions needed in CFG rules. The text is enclosed in “%{ %}” brackets.
2. Bison Declaration: Bison declaration contains the bison definitions like token declarations, type declarations, and others.
3. Grammar Rules: This section contains the CFG rules.
4. Additional C codes: This section contains C statements and additional functions.

How Flex and Bison Works Together:

Bison reads the grammar descriptions in .y file and generates a syntax analyzer (parser), that includes function yyparse, in file .tab.c. Included in file .y are token declarations. The –d option causes bison to generate definitions for tokens and place them in file .tab.h. Flex reads the pattern descriptions in .l, includes file .tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.
Finally, the lexer and parser are compiled and linked together to create executable .exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

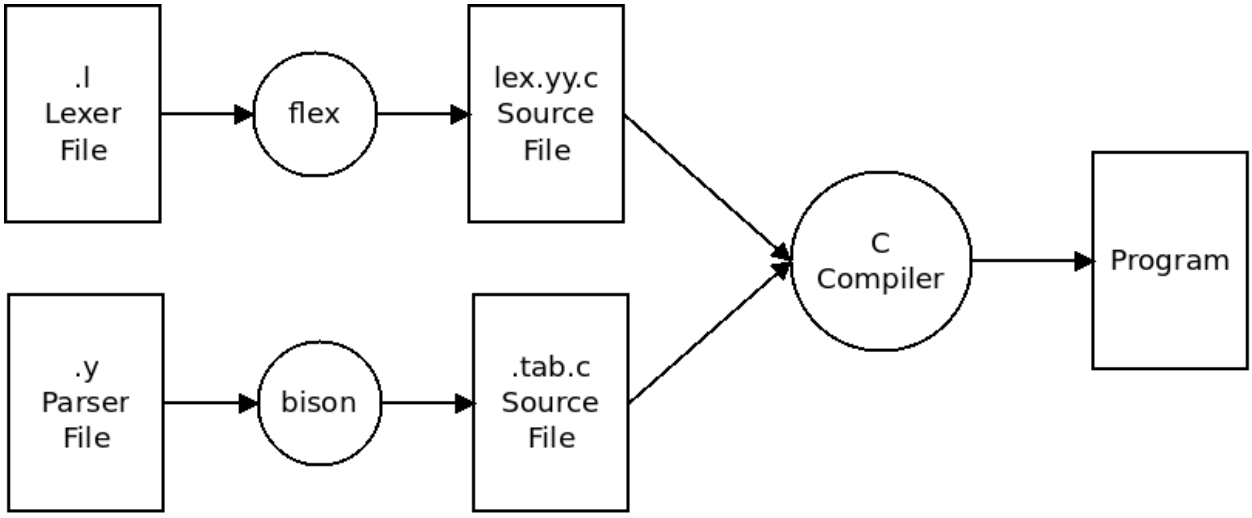

Figure 2: Building a Compiler with Flex and Bison.

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:

Table 1: Tokens of the compiler

#	Token	Input String	Description
1	ROOT	root	Indicates the start of the main program.
2	START	:	Indicates the start of any block.
3	END	end	Indicates the end of a block.
4	INT_TYPE	int	Declares variable of integer type.
5	REAL_TYPE	real	Declares variable of real type.
6	STRING_TYPE	string	Declares variable of string type.

7	VARIABLE	[A-Za-z][A-Za-z 0-9]+	Declaration of a variable. A variable will start with a letter and can contain letters and digits.
8	ARRAY_VAR	[$][$][A-Za-z][A-Za-z0-9]+	Declaration of an array.
9	COMMENT	[#][^#]*[#]	Declaration of a comment. A comment starts and ends with #.
10	FROM	from	Denotes start value of a loop.
11	TO	to	Denotes end value of a loop.
12	BY	by	Denotes increment/decrement of value in a loop.
13	DO	repeat	Indicates the block to repeat till a certain condition is met.
14	WHILE	while	Denotes a looping block that repeats as long as the condition is fulfilled.
15	IF	if	Indicates a conditional block.
16	ELIF	elif	Tries to match this conditional block if previous conditions were not matched. Similar to else if in C/C++.
17	ELSE	else	If no IF/ELIF is matched this block is executed.Similar to else in C/C++.
18	CHOICE	choices for	The CHOICE statement allows us to execute one code block among many alternatives.
19	OPTION	option	Indicates an alternative for the choice block.
20	DEFAULT	none	Default block to execute if no alternative matches the condition in the choice statement.
21	SEE	see	Print any variables.
22	READ	read	Take input from the user.
23	MODULE	module	Declaration of a module(function in C/C++)
24	CALL	call	Call a module.
25	PUSH	push	Push element in an array.
26	POP	pop	Remove the last element from an array.
27	SORT	sort	Sort an array in ascending order.
28	EOL	;	Indicates the end of a line.
29	INTEGER	[0-9]+	Indicates an integer number.

30	REAL	[0-9]*[.][0-9]+([eE][+-]?[0-9]+)?	Indicates a float number.
31	STRING	\"(\\.|[^"\\])*\"	Indicates a string literal.
32	PPLUS	++	Increments an integer variable by 1.
33	MMINUS	--	Decrements an integer variable by 1.
34	LEQL	<=	Relational Operators for less than or equal to check.
35	GEQL	>=	Relational Operators for greater than or equal to check.
36	EQL	==	Relational Operators for equality check.
37	NEQL	!=	Relational Operators for not equality check.
38	XOR	XOR	Logical XOR operation
39	AND	AND	Logical AND operation
40	OR	OR	Logical OR operation.
41	NOT	~	Logical NOT operation.
42	SIN	SIN	Sine Function.
43	COS	COS	Cosine Function
44	TAN	TAN	Tangent Function
45	SQRT	SQRT	Square root Function.
46	LOG	log	Log function.
47	LOG2	log2	Log function in 2-base.
48	LN	ln	Natural Logarithm.
49	FACTORIAL	!	Factorial of a number.
50	ARROW	->	Helper operator for see(printing variable)
51	RARROW	<-	Helper operator for read(taking input)

Context-Free Grammars(CFG):

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but it cannot describe all possible languages. Context-free grammars can generate context-free languages. They do this by taking a set of variables that are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

Context-Free Grammars used in Compiler:

```
program:   ROOT START statements END
             ;
statements:
         |   statements statement
   ;
statement:        EOL
                | COMMENT
                | declaration EOL
                | assigns EOL
                | show EOL
                | read EOL
                | expr EOL
                | if_blocks
                | choice_block
                | loop_block
                | module_declare
                | module_call
                | array_operations
   ;
declaration:
         INT_TYPE int_variables
         | REAL_TYPE real_variables
         | STRING_TYPE string_variables
   ;

int_variables:
         int_variables ',' int_var
         | int_var
   ;
int_var:
         VARIABLE '=' expr
         | VARIABLE
         |   ARRAY_VAR
         |   ARRAY_VAR '=' '{' ints '}'
```

```
    ;
ints:
          ints ',' expr
        | expr
    ;
real_variables:
          real_variables ',' real_var
          | real_var
    ;
real_var:
          VARIABLE '=' expr
          | VARIABLE
          | ARRAY_VAR
          | ARRAY_VAR '=' '{' reals '}'
       ;
reals:
          reals ',' expr
        | expr
       ;
string_variables:
          string_variables ',' string_var
          | string_var
    ;
string_var:
          VARIABLE '=' STRING
        | VARIABLE
        | ARRAY_VAR
        | ARRAY_VAR '=' '{' strings '}'
       ;
strings:
          strings ',' STRING
        | STRING
      ;
show:
          SEE ARROW print_vars
    ;
print_vars:
          print_vars ',' VARIABLE
        | print_vars ',' ARRAY_VAR
        | print_vars ',' ARRAY_VAR '[' expr ']'
        | VARIABLE
        | ARRAY_VAR
        | ARRAY_VAR'[' expr ']'
      ;
read:
      READ RARROW read_vars
    ;
read_vars:
          read_vars ',' VARIABLE
        | read_vars ',' ARRAY_VAR '[' INTEGER ']'
```

```
          | VARIABLE
          | ARRAY_VAR '[' INTEGER ']'
   ;
assigns:
           assigns ',' assign
          | assign
   ;
assign:

          VARIABLE '=' expr
          | ARRAY_VAR '[' INTEGER ']' '=' expr
              ;
if_blocks:
          IF if_block else_statement
   ;
if_block:
          expr START statement END
               ;
else_statement:
          | elif_statement   single_else
          | single_else
   ;
single_else: ELSE START statement END
             ;
elif_statement:
          elif_statement  single_elif
          | single_elif
   ;
single_elif:
          ELIF expr START statement END
                ;


choice_block:
          CHOICE choice_variable START options END
            ;
choice_variable:
          VARIABLE
             ;
options:
          optionlist default
          | default
   ;
default:
          DEFAULT START expr END
              ;
optionlist:
          optionlist option
          | option
   ;
```

```
option:
        OPTION expr START expr END
      ;

loop_block:
        FROM expr TO expr BY expr START expr END
                | FOREACH ARRAY_VAR AS VARIABLE START expr END
                | WHILE while_conditions START expr END
                | DO START expr END WHILE while_conditions EOL
            ;
while_conditions:
            VARIABLE PPLUS '<' expr
          | VARIABLE PPLUS LEQL expr
          | VARIABLE PPLUS NEQL expr
          | VARIABLE MMINUS '>' expr
          | VARIABLE MMINUS GEQL expr
          | VARIABLE MMINUS NEQL expr
              ;

module_declare:
          MODULE module_name '(' module_variable ')' START statements END
   ;
module_name:
        VARIABLE
              ;
module_variable:
        module_variable ',' single_var
        | single_var
   ;
single_var:  INT_TYPE VARIABLE
        | REAL_TYPE VARIABLE
        | STRING_TYPE VARIABLE
      ;
module_call: CALL user_module_name '(' user_parameters ')'
       ;
user_module_name:
        VARIABLE
       ;
user_parameters: user_parameters ',' single_param
        | single_param
   ;
single_param:  VARIABLE
   ;
array_operations:
          ARRAY_VAR '.' PUSH '(' expr ')'
          | ARRAY_VAR '.' POP '('')'
          | ARRAY_VAR '.' SORT
   ;
expr:
        INTEGER
```

```
        | REAL
        | VARIABLE
        | '+' expr
        | '-' expr
        | PPLUS expr
        | MMINUS expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '^' expr
        | expr '%' expr
        | expr '<' expr
        | expr '>' expr
        | expr LEQL expr
        | expr GEQL expr
        | expr EQL expr
        | expr NEQL expr
        | expr AND expr
        | expr OR expr
        | expr XOR expr
        | NOT expr
        | '(' expr ')'
        | SIN '(' expr ')'
        | COS '(' expr ')'
        | TAN '(' expr ')'
        | LOG '(' expr ')'
        | LOG2 '(' expr ')'
        | LN '(' expr ')'
        | SQRT '(' expr ')'
        | VARIABLE PPLUS
        | VARIABLE MMINUS
        | expr FACTORIAL

          ;
```

Main Features:

- **Beginning of Program**: The program starts with ‘root’.Every statement of the program is inside the root block. A block starts with ‘:’ and ends with ‘end’.

- **Variable Declaration**: There can be three types of data:
    1. Integer: A number without any fractional component.
    2. Real: 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:

```
int abc,xy,p=12;
real var1 = 4.5;
string str = “ablce”;
```

- **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. An array variable name starts with $$ and follows the variable declaration rule. To initialize an array we need to put its value enclosed in curly braces separated by a comma.
  Numeric array indexes can be updated too.
  Code Snippet:

```
int $$array = {1,2,3,4,5};
$$array[2] = 45;
string $$str = {“abc”,”def”,”ghi”};
```

  Array Operations: We can perform 3 types of operations on an array.
  1. PUSH(): We can insert an element at the end of an array.
  2. POP(): We can pop the last element from the array.
  3. SORT: We can sort the array in ascending order.

  Code Snippet:

```
int $$array = {1,2,3,4,5};
$$array.push(3);
$$array.pop();
$$array.sort;
```

- **Taking Input**: We can also take input given by the user from the console using ‘read’.
  Code Snippet:

```
int a,b,c;
read<-a,b,c;
```

- **Printing Output**: We can also print output in the console using ‘see’ console.
  Code Snippet:

```
int a,b,c;
read<-a,b,c;
```

- **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. From-To Loop: Basic syntax for this loop is:

```
from X to Y by Z:
        #code blocks#
end
```

The code blocks are executed until X reaches Y .The value of X is changed by Z each time the code runs.

Code Snippet:

```
from 5 to 10 by 1:
    1+2+3;
end
```

  2. While Loop: Basic syntax for while loop:

```
while  expression:
        #code-blocks#
end
```

The code blocks are executed as long as the expression is true. The expression is checked before executing the codes each time.

Code Snippet:

```
int a = 5;
while a-->0:
        2*4/3;
end
```

3. 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:

```
repeat:
        #code-blocks#
end
while expression;
```

Code Snippet:

```
repeat:
        12*12
end
while a++<15 ;
```

- **Conditional Statements**: Conditional Statements are used to make decisions based on the conditions. There are four types of conditional statements in this language:
    1. if statement: 'If' statement is always used with a condition. The condition is evaluated first before executing any statement inside the body of If. The syntax for 'if' statement is as follows:

```
if  condition:
        #codes#
end
```

The codes inside if block are executed if the condition is true.

Code Snippet:

```
if a>10:
    X = 4;
end
```

2. if-else statement: The if-else is statement is an extended version of If. The general form of if-else is as follows:

```
if condition:
        #codes
end
else:
        #codes
end
```

The codes inside else block will be executed when the condition in if block is false.

Code Snippet:

```
int p = 25;
if(p>20):
        1*2*3;
end
else:
        2*3*4;
end
```

3. If-elif-else statement: Nested elif is used when multipath decisions are required. The general syntax of how elif ladders are constructed is as follows:

```
if:
        #codes
end
elif:
        #codes
end
elif:
        #codes
end
else:
        #codes
end
```

This chain generally looks like a ladder hence it is also called an elif ladder. The test expressions are evaluated from top to bottom. Whenever a true expression is found, statements associated with it are executed. When all the expressions become false, then the default else statement is executed.

Code Snippet:

```
if p>20 AND p<25:
        1;
end
elif p>30 :
        2;
end
elif p>20 OR p<10:
        3;
end
else:
        4;
end
```

4. Choice-option statement: The choice statement allows us to execute one code block 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 and write.
   Basic syntax:

```
choices for X:
        option A:
                #codes
        end
        option B:
                #codes
        end

        option C:
                #codes
        end

        option D:
                #codes
        end

        none:
```

```
            #codes
        end
```

The expression is evaluated once and compared with the values of each option label. If there is a match, the corresponding statements of the matching label are executed. If there is no match, the default statements are executed.

Code Snippet:

```
choices for p:
        option 5:
                1;
        end
        option 10:
                2;
        end
        option 25:
                3;
        end
        none:
                4;
        end
end
```

- 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. To declare a module we need to add ‘module’ before the declaration.
  The basic syntax for module declaration is:

```
module moduleName (data-type var1,data-type var2,....):
        #codes
end
```

Code Snippet:

```
module myFunction(int a,int b):
                #codes#
        end
```

If we want to call the modules we will have to use the ‘call’ keyword.
Code Snippet:

```
call myFunction(a,p);
```

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

Conclusion

The compiler is built using flex bison without using any data structures. As a result, our compiler won't be able to make any jump throughout the whole process. Hence the actual functionality of if-else or loops can't be achieved. To achieve such functionality we will need to use some kind of data structure like Abstract Syntax Tree(AST). Nevertheless, the compiler works perfectly 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-BrehamPie/A-Simple-Compiler-Using-Flex-And-Bison: Compiler Project-3212 (github.com)