
# Lab Report

## CSE 3212: COMPILER DESIGN LABORATORY

## **Topic**: Design an Abstract Programming Language

**Submitted To:**


| | |
|---|---|
| Dola Das<br>Lecturer<br>Department of Computer Science and Engineering<br>Khulna University of Engineering and Technology, Khulna | Md. Ahsan Habib Nayan<br>Lecturer<br>Department of Computer Science and Engineering<br>Khulna University of Engineering and Technology, Khulna |


**Submitted By:**


Md. Nazrul Islam
Roll: 1707086

Year: 3rd; Semester: 2nd, Section: B
Department of Computer
Science and Engineering
Khulna University of Engineering
& Technology, Khulna


**Submission Date: 15 June 2021**

# Table of Contents

## COMPILER DESIGN:

A language translator is a program which translates programs written in a source language into an equivalent program in an object language. The source language is usually a high-level programming language and the object language is usually the machine language of an actual computer. From the pragmatic point of view, the translator defines the semantics of the programming language, it transforms operations specified by the syntax into operations of the computational model to some real or virtual machine.

A compiler is a translator whose source language is a high-level language and whose object language is close to the machine language of an actual computer. The typical compiler consists of several phases each of which passes its output to the next phase:

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

• The lexical phase (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine. Lex and Flex are tools for generating scanners: programs which recognize lexical patterns in text. Flex is a faster version of Lex. In this chapter Lex/Flex refers to either of the tools.

• The parser groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as a push-down automata. Yacc and Bison are tools for generating parsers: programs which recognize the grammatical structure of programs. Bison is a faster version of Yacc.

• The semantic analysis phase analyzes the parse tree for context-sensitive information often called the static semantics. The output of the semantic analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program. This phase is often combined with the parser. During the parse, information concerning variables and other objects is stored in a symbol table. The information is utilized to perform the context-sensitive checking.

• The optimizer applies semantics preserving transformations to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code.

• The code generator transforms the simplified annotated parse tree into object code using rules which denote the semantics of the source language. The code generator may be integrated with the parser.

• The peep-hole optimizer examines the object code, a few instructions at a time, and attempts to do machine dependent code improvements.

## FLEX:

**FLEX (Fast LEXical analyzer generator)** is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.
The flex input file consists of three sections, separated by a line containing only '%%'.

**definitions option, declaration**
**%%**
**rules**
**%%**
**user code**

## Program Structure of FLEX:

In the input file, there are 3 sections:

**1. Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{ %}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c

```
%{
  // Definitions, option, declaration
%}
```

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

```
%%
pattern action
%%
```

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

## BISON:

Bison is a general-purpose *parser generator* that converts a grammar description (Bison Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

Bison takes the parser specification from a file. Following the convention of yacc the file name should end with .y . The output file name by default uses the prefix of the input file and is named as .tab.c .

**Bison File Format:**
A bison input file (bison grammar file) has the following structure with special punctuation symbols %%, %{ and %}.

```
%{
Declarations
%}
Definitions
%%
Productions
%%
 User subroutines
```

## Program Structure of BISON:

**1. Definition Section:** This section covers macro definitions as well as declarations of functions and variables utilized in the grammar rules' activities. The declarations from a header file are obtained using the #include command. We can eliminate the "% {… %}" delimiters that enclose this section if there are no C declarations.

**2. Grammar Section:** The grammar rules section contains one or more Bison grammar rules. A formal grammar is a mathematical construct. To define the language for Bison, one must write a file expressing the grammar in Bison syntax. I

**3. User Code Section:** This section contains C statements and additional functions.

Bison is designed for use with C code and generates a parser written in C. The parser is configured for use in conjunction with a flex generated scanner and relies on standard shared features (token types, yylval, etc.) and calls the function yylex as a scanner coroutine. A grammar is provided specification file, which is traditionally named using a .y extension. Bison is invoked on the .y file and it creates the y.tab.h and y.tab.c files containing a thousand or so

lines of intense C code that implements an efficient LALR(1) parser for the given grammar, including the code for the actions that is specified. The file provides an external function yyparse.y that will attempt to successfully parse a valid sentence.

## INTRODUCTION:

This project being a mini compiler for a unique abstract programming language, focuses on generating tokens and parsers. We have to create the tokens using flex and for creating the grammar rules we have to use the bison. We also have to create an input file in the context of the abstract language for which the tokens and parsers have been generated.
In nutshell in this project

- Making a design of an abstract programming language
- Creating a token for the abstract language using a flex
- Creating a parser for the abstract language using a bison
- Creating an input file in the context of the language and generate an output.

In this paper there is a demonstration of how to create a token and parser for an abstract programming language using flex and bison.

## TOKENS USED IN THE LANGUAGE:

| Input Strems | Token | Meaning/Equivalence to C |
|---|---|---|
| "import "[a-zA-Z]([a-zA-Z])* | import | Including header files |
| [&][&].* | comment [&][&].* | Single line Comment |
| [ ]*[&][*][&][a-zA-Z0-9!@#*(){}_+-,.:\|?><\n\t ]*[&][*][&] | multiple_line_comment | Multiline Comment |
| “[” | LB | Left Parenthesis, “(“ |
| “]” | RB | Right Parenthesis, “)” |
| “<” | LAB | Left Curly Brace, “{“ |
| “>” | RAB | Right Curly Brace, “}” |
| “,” | CM | Comma |
| “;” | SM | Semicolon |
| “integer” | INT | Datatype: int |
| “float” | FLOAT | Datatype: float |
| “double” | DOUBLE | Datatype: double |
| “character” | CHAR | Datatype: char |
| [0-9]+ | NUM | Natural Number |
| “++” | PLUS | “+” (Plus Operator) |
| “--” | MINUS | “-” (Minus Operator) |
| “**” | MULT | “*” (Multiply Operator) |
| “//” | DIV | “/” (Division Operator) |
| “!” | FACT | Factorial |

| | | |
|---|---|---|
| “#++” … “#++” | inc | “++” (Increment Operator) |
| “#--” … “#--" | dec | “—” (Decrement Operator) |
| “^” | not | “!” (Not Operator) |
| “->” | ASSIGN | “=” (Assignment Operator) |
| “>>” | GT | “>” (Greater than) |
| “<<” | LT | “<” (Less than) |
| “>=” | GE | “>=” (Greater or Equal) |
| “<=” | LE | “<=” (Less or Equal) |
| “==” | EQUAL | “==” (Equal to) |
| “!=” | NOTEQUAL | “!=” (Not Equal) |
| “if” | IF | if (Conditional Statement) |
| “else” | ELSE | else (Conditional Statement) |
| “elif” | ELIF | else if (Conditional Statement) |
| “control” | SWITCH | “switch” in Switch Statement |
| “event” | CASE | “case” in Switch Statement |
| “Default” | DEFAULT | “default” in Switch Statement |
| “loop” | FOR | Quite similar to “for” iterator |
| “While” | “WHILE” | While Loop iterator |
| “function main” | VOIDMAIN | Main Function |
| "function "[a-zA-Z]([a-zA-Z])* | FUNCTION | User Defined Function |
| [a-zA-Z]([a-zA-Z0-9])* | VAR | Identifier or Variable |

## ARCHITECTURE OF LANGUAGE:

In the abstract language the following contents have been implemented:

- Arithmetic expressions
- Boolean expressions
- Logical and Relational expressions
- Assignment
- Conditional Statements
    - IF – ELSE
    - Switch – Case
- Loop statements
    - Simple loop
    - While loop
- Function definition and declare
- Importing necessary header files

## FEATURES:

Compiler is going to support following features:

- **Keywords**: integer, float, double, character, if, else, elif, control, event, Default, While, loop, function main, function, 'import '

- **Data Types:** integer(working), float, double, character

- **Special keywords:** function main, function, 'import '

| Punctuators | Purpose | Equivalence to C |
|---|---|---|
| function main | Used as main function | main |
| 'import ' | To include necessary headers | #include |
| function | To declare and call a function | Typedef function_name |

## ❖ Structure of the language:

```
❖ && import necessary headers
❖ import library
❖
❖ && user defined function
❖ function name [ integer a ; ]  && function accepts one integer
  parameter
❖ <
❖ 6 ++ 8 ;
❖ >
❖
❖ && main function
❖ function main []
❖ <
❖ integer  aa ,  bb , cc ;
❖
❖ aa -> 8 ;
❖ bb -> 4 ;
❖
❖ && user defined function calling
```

```
❖ function name [ integer param ;] ; && passing integer parameter p
  aram
❖
❖ >
```

❖ **Punctuators:**

- ➢ Brackets (‘[ ]’)
- ➢ Semicolon (;)
- ➢ Angular Brackets (‘<>’)
- ➢ Equal Sign ( -> )
- ➢ Comma operator and punctuator (,)

| Punctuators | Purpose | Equivalence to C |
|---|---|---|
| Brackets **(‘[ ]’)** | Used as Parenthesis | ( ) |
| Semicolon **(;)** | To define end of a statement | ; |
| Angular Brackets **(‘<>’)** | Used to separate block | { } |
| Comma operator and punctuator **(,)** | Separator | , |
| Equal Sign **( -> )** | To assign value to a variable | = |

❖ **Variable:** Variable must have to start with a letter and followed by any number of letters of digits. Eg:

### In case of the Identifiers Rules:

- Identifiers are case sensitive
- Variable must have to start with a letter and followed by any number of letters of digits
- Keywords are not allowed to be used as identifier
- No special character can be used in identifier

```
❖              integer a ; && valid
❖              integer aa, b9, dddf8 ; && valid
❖              integer 9ge ; && not valid
❖              integer _ekf, &dnfke ; && not valid
```

## ❖ Operators:

### ➢ Unary Operators:

- ***Increment*** (#++ variable/integer #++)

```
❖          #++ 8 #++ ;  #++ var #++ ;
```

- ***Decrement*** (#-- variable/integer #--)

```
❖          #-- 8 #-- ; #-- var #-- ;
```

- ***Factorial*** (!)

```
❖              9 ! ; var ! ;
```

### ➢ Binary Operators:

- ***Summation*** (++)

```
❖          var ++ 2 ;    6 ++ 2;  9 ++ var;
```

- ***Subtraction*** (--)

```
❖          6 -- 2 ; var -- 2 ;       9 -- var;
```

- ***Multiplication*** (**)

```
❖          6 ** 2 ; var ** 2 ;    4 ** var;
```

- ***Division*** (//)

```
❖          6 // 2 ; var // 2 ;    3 // var ;
```

### ➢ Logical and Relational Operators:

- ***Greater than*** (>>): returns a Boolean value

```
❖          var >> 2 ; 6>> 2;   9 >> var;
```

- ***Less than*** (<<): returns a Boolean value

```
❖          var << 2 ; 6 << 2;  9 << var;
```

- ***Greater or Equal*** (>=): returns a Boolean value

```
❖          var >= 2 ; 6 >= 2;  9 >= var;
```

- ***Less or Equal*** (<=): returns a Boolean value

```
❖              var <= 2 ;  6 <= 2;   9 <= var;
```

- ***Equal (==)***: returns a Boolean value

```
❖              var == 2 ;  6 == 2;   9 == var;
```

- ***Not Equal (! =):*** returns a Boolean value

```
❖              var !=2 ;   6 != 2;   9 != var;
```

➢ **Assignment Operators:**

- ***Assignment*** ( -> )

```
❖              var -> 2 ;
```

❖ **Comments:**

➢ **Single Line Comment**: (&& single line comment)

```
❖              && single line comment
```

➢ **Multiline Comment**: (&*& multiline comment&*&)

```
❖              &*& this is
❖              multiline comment
❖              &*&
```

❖ **Conditional Statements**

➢ **IF – ELSE:**

- ***Simple if***

```
❖     if [ 5 ++ 3 -- 8 ]
❖         <
❖             12 ++ 8 ;
❖         >
❖
❖     if [ 5 ++ 8 ]
❖         <
```

```
            4 ! ;
        >
    if [ 5 == 5 ]
        <
            5 ! ;
        >
```

- ***Simple if else block***

```
    if [ 7 <= 7 ]
        <
            2 ++ 7 ;
        >
    else
        <
            6 ++ 8 ;
        >
```

- ***If – elif – else block***

```
    if [ 7 << 5 ]
        <
            2 ++ 7 ++ 6 ** 2 -- 18 ;
        >
    elif [ 2 >> 1 ]
        <
            3 ++ 4 ** 2 -- 2 ;
        >
    else
        <
            6 ++ 8 ;
        >
```

- ***If – (nested if – else) – else block***

```
    if [ 420 >> 69 ]
        <
            if [ 420 << 69 ]
            <
                2 -- 18 // 3 ;
            >
            else
            <
                3 ** 3 -- 1 ;
            >
```

```
            2 -- 18 // 3 ;
        >
    else
        <
            6 ++ 3 -- 1 ;
        >
```

- ***if – else – (nested if – else) block***

```
    if [ 2 >= 7 ]
        <
            2 ++ 7 ;
        >
    else
        <
            if [ 13 != 69 ]
            <
                7 ++ 7 -- 2 ;
            >
            else
            <
                3 -- 1 ;
            >
            6 ++ 8 // 4 ++ 2 ** 3 -- 1 ;
        >
```

### ➢ Switch – Case:

- ***control – event***

```
    control [ 2 ]
        <
            event 1 : 4 ++ 2 ;
            event 2 : 3 ++ 2 ;
            Default : 5 ! ;
        >
```

## ❖ Loop:

### ➢ While loop:

```
    While [ 6 >> 5 ]
```

```
<
    4 ++ 2 ;
    aa -- 3 ;
>
```

- **Normal iterative loop using loop keyword:**

```
loop [ 2 , 6, 2 ]
    <
        bb -> bb ++ 2 ;
    >
```

# DESIGN STAGES AND IMPLEMENTATION:

## Phase 1: Creating token using Flex program from Lex file:

- FLEX tool was used to create a scanner for the abstract language
- The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
- The scanner also scans for the comments (single-line and multiline comments) and writes the source file without comments onto an output file which is used in the further stages.
- All tokens included are of the form <token-name>.
- A global variable ‘yylavl’ is used to record the value of each lexeme scanned. ‘yytext’ is the lex variable that stores the matched string.
- Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- The rules are regular expressions which have corresponding actions that execute on a match with the source input.

## Phase 2: Creating parser and symbol table to keep track of variables and operation using Yacc file:

● Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your programming Language grammar.

● The design implementation supports

1. Variable declarations and initializations
2. Variables of type
3. Arithmetic, boolean, logical and relational expressions
4. Postfix and prefix expressions
5. Constructs - if-else, switch-case, while loop and simple loop
6. function definition and function call

● Yacc tool was used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

● A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurrences, the category of the token (constant, variable, keyword, operator), the value that it holds the datatype.

```
typedef struct entry {
    char *str;
    int n;
} storage;
```

● $1 is used to refer to the first token in the given production like $n refers to the nth token and $$ is used to refer to the resultant of the given production.

● Expressions are evaluated and the values of the used variables are updated accordingly.

## Phase 3: Run the Yacc file and Lex file together to verify an input program:

- First the Yacc file with a filename of x(x.y) has been compiled with bison which will create two file x.tab.h and x.tab.c(this two file has the naming convension is filename.tab.h and filename.tab.c). The definition of the token in the x.tab.h" file which is later used in the lex file. To compiler the Yacc file with bison:

```
>> bison -d x.y # create x.tab.h, x.tab.c
```

- Then the lex file wil filename x.l has been compiler with the flex program which will create lex.yy.c file. To compile lex file

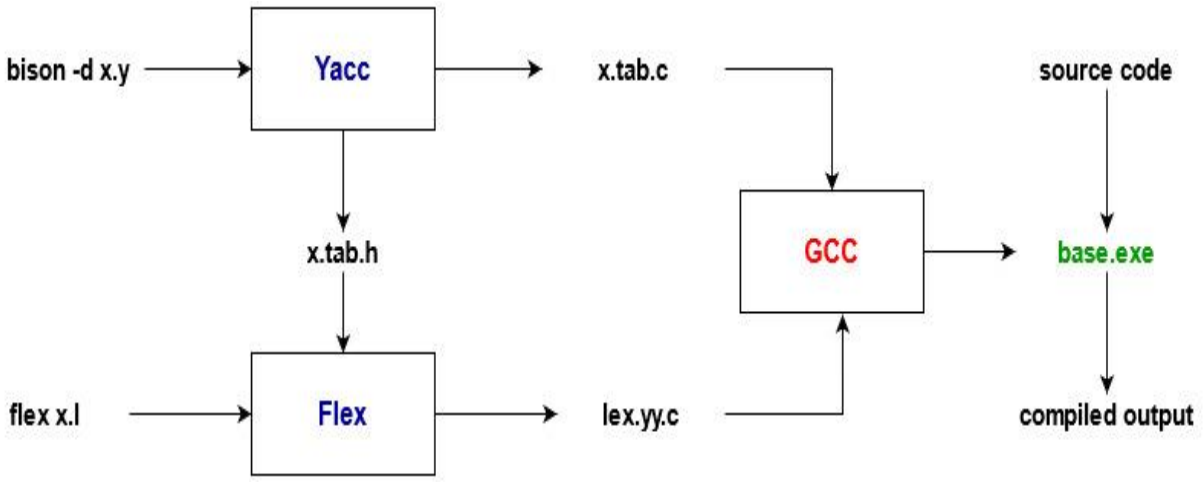
```
>> flex x.l   # create lex.yy.c
```

- Then after the using a GCC compiler both c files have been created in bison and flex file compilation step called lex.yy.c and x.tab.c to create a executable file.

```
>> gcc -o base lex.yy.c x.tab.c # Creates an executive
                                  # File named base
```

- Now using this executable file, the input source file can be compiled easily.

## The whole process in a nutshell:



## The Source Code

- Lex File:

  Mini-Compiler-Design-Project/1707086.l at master · nz-nAJn/Mini-Compiler-Design-Project (github.com)

- Bison File:

  Mini-Compiler-Design-Project/1707086.y at master · nz-nAJn/Mini-Compiler-Design-Project (github.com)

## Sample Input and Compiled Output

- ❖ Input:

  Mini-Compiler-Design-Project/input.txt at master · nz-nAJn/Mini-Compiler-Design-Project (github.com)

- ❖ Output:

  Mini-Compiler-Design-Project/output.txt at master · nz-nAJn/Mini-Compiler-Design-Project (github.com)

## REFERENCES:


1. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/120%20Introducing%20bison.pdf
2. https://cse.iitkgp.ac.in/~goutam/compiler/lect/lect8.pdf
3. http://alumni.cs.ucr.edu/~lgao/teaching/bison.html
4. http://web.cecs.pdx.edu/~sheard/course/Cs321/project/index.html
5. All the file link:

   nz-nAJn/Mini-Compiler-Design-Project (github.com)


**The End**