Lab Report on Project

CSE 3212 : COMPILER DESIGN LABORATORY

Submitted To:

Dola Das Lecturer Department of Computer Science and Engineering Khulna University of Engineering and Technology, Khulna	Md. Ahsan Habib Nayan Lecturer Department of Computer Science and Engineering Khulna University of Engineering and Technology, Khulna

Submitted By:

Md. Samin Yeasar
Roll: 1507099

Year: 3rd Term: 2nd

Session: 2019-2020

Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

Submission Date: June 8, 2021

Flex and Bison

Flex and Bison are tools for building programs that handle structured input. They were originally tools for building compilers, but they have proven to be useful in many other areas. In this project, Flex and Bison were used to build a compiler.

Flex

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. In this lab, we have used Flex (Fast lexical analyzer generator), a free and open-source software, as an alternative to lex.

Flex is a tool for generating *scanners*: programs which recognized lexical patterns in text. flex 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'*, which defines a routine *'yylex()'*. This file is compiled and linked with the *'-lfl'* 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.

Bison

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. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. It interfaces with scanner generated by Flex. Scanner called as a subroutine when parser needs the next token.

Where flex recognizes regular expressions, bison recognizes entire grammars. Flex divides the input stream into pieces (tokens), and then bison takes these pieces and groups them together logically.

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF.

Project Idea

In this *CSE 3212: Compiler Design Laboratory* course, we were assigned to design our own unique programming language and to build a compiler using Flex and Bison for compiling the programs written in that language.

To perform this task and to complete the project, I followed these following steps sequentially:

- ✓ **Source Code:** A *programming language* with different features and rules created by me was designed.
- ✓ **Lexical Analyzer:** A *lex program* was written which reads patterns from the source code and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on the patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.
- ✓ **Syntax Analyzer:** *Grammar* rules, data structures and programs were written to a "*.y"* file. This builds symbol table which contain information such as data type and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index. Yacc reads the grammar and generate C code for a syntax analyzer or *parser*. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree.

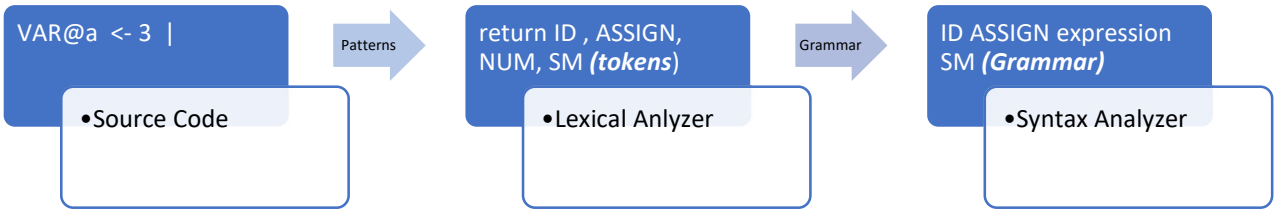

Figure 1: Working steps of the project

Manual:

1. **Datatypes:** Datatypes available are integer (declared by intgr), double (declared by dbl) and character (declared by ch). Double and character datatypes are still working in progress. Example: intgr VAR@a

2. **Predefined Functions:**

Name	Example	Purpose
`outVar`	`outVar #s VAR@aB e# |`	To print variable's value
`outStr`	`outStr #s "hello" e# |`	To print a string
`outNL`	`outNL #s e#|`	To print a new line

3. **Special Keywords:**

Keyword	Equivalents in C	Purpose
`#s e#`	( )	Parenthesis
`*s e*`	{ }	Curly braces
`compiler_function`	main()	Starting the program
`,,`	,	Separator
`|`	;	Endling of a statement

4. **Variable:** Variables start with `VAR@` and then follow any number of alphabet or number or special characters. Example: `VAR@yeasar99`

5. **Operators:**

Operator	Operation	Example
<-	Assign	`VAR@a <- 9 |`
jog	Addition	`VAR@a <- VAR@b jog VAR@c`
bad	Subtraction	`VAR@a <- VAR@b bad VAR@c`
gun	Multiply	`VAR@a <- VAR@b gun VAR@c`
vag	Division	`VAR@a <- VAR@b vag VAR@c`
vagShesh	Remainder	`VAR@a <- VAR@b vagShesh VAR@c`
ghat	Power	`VAR@a <- VAR@b ghat VAR@c`

6. **Relational Operators:**

Operator	Operation	Example
boro_theke	Greater than	`2 boro_theke 1`
choto_theke	Less than	`1 choto_theke 2`
boro_shoman	Greater or equal	`2 boro_shoman 1`
choto_shoman	Less than or equal	`1 choto_shoman 3`
shoman	Equal	`2 shoman 2`
oshoman	Not equal	`2 oshoman 1`

7. **Conditional Operators:** "jodi", "othoba" and "noile" are used to build conditional (If-else in C programming language) statements. Example:

```
jodi #s 5 choto_theke 4 e#
       *s
              ??statement here
       e*
othoba #s 5 choto_theke 4 e#
       *s
              ??statement here
       e*
noile
       *s
              ??statement here
       e*
```

8. **Loop:** In this project, there are two types of loop- forward and backward loop. Forward loop is executed by incrementing the loop control variable's value. And the backward one is executed by decreasing the value of the loop control variable. Example:

```
??forward loop

fwdloop #s VAR@st porjonto> VAR@ed barbe> 1 e#

*s

       VAR@b <- 7|

e*


?? backward loop

bckloop #s VAR@bckstart porjonto> VAR@bckend kombe> 1 e#
```

```
*s
      VAR@b <- 7|
e*
```

9. **Switch case:** Here, “mele_kina” is used as an alternative to the “switch-case” statements. Example:

```
??switch case implementation
mele_kina #s 7 e#
*s
      1>>
            *s
            e*
      7>>
            *s
                  outStr #s "switch" e# |
            e*
      sheshmesh>>  *s

            e*
e*
```

10. **Function creation:** Functions can be created like this:

```
intgr newFunction@Max #s intgr VAR@a,, intgr VAR@b e#
*s
      outStr #s "hello function" e# |
      outNL #s e# |
e*
```

11. **Comment:** “??” is used to comment a single line. Example: `??This is a comment.`

References

1. *LEX & YACC TUTORIAL* by Tom Niemann
2. *Flex, version 2.5* by Vern Paxson