Lab Report

CSE 3212: Compiler Design Laboratory

Submitted to:

Dola Das
Lecturer
Department of Computer Science & Engineering
Khulna University of Engineering & Technology

Md. Ahsan Habib Nayan
Lecturer
Department of Computer Science & Engineering
Khulna University of Engineering & Technology

Submitted by:

Farhan Sadaf
Roll: 1707066
Year: 3rd, Term: 2nd
Department of Computer Science & Engineering
Khulna University of Engineering & Technology

Submission date: 14 June, 2021

Introduction:

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers") or patterns. Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex.
Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidt and described in 1975, is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX standard.
Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. A Flex lexical analyzer usually has time complexity **O(n)** in the length of the input. That is, it performs a constant number of operations for each input symbol.

Bison is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1) or GLR parser for that grammar. Yacc/Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar. The input file for the Yacc/Bison utility is a Yacc/Bison grammar file. The Yacc/Bison grammar input file conventionally has a name ending in .y.

Fig 1 illustrates the sequence in which Flex and Bison work together to compile a user code. The **pattern** in this diagram is a **lex** file (e.g. project.l) that is created by the programmer. Lex will read the patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings from **source code** and based on the patterns written in lex file (project.l), it converts strings to tokens. **Tokens** are numerical representations of strings and simplify processing.
The **grammar** in fig 1 is the bison file (e.g. program.y) created by the programmer. Yacc will read this grammar file 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**. The syntax tree imposes a hierarchical structure of the tokens.
Then on the **code generation** step, a depth-first walk of the syntax tree is done in order to **generate final code**.

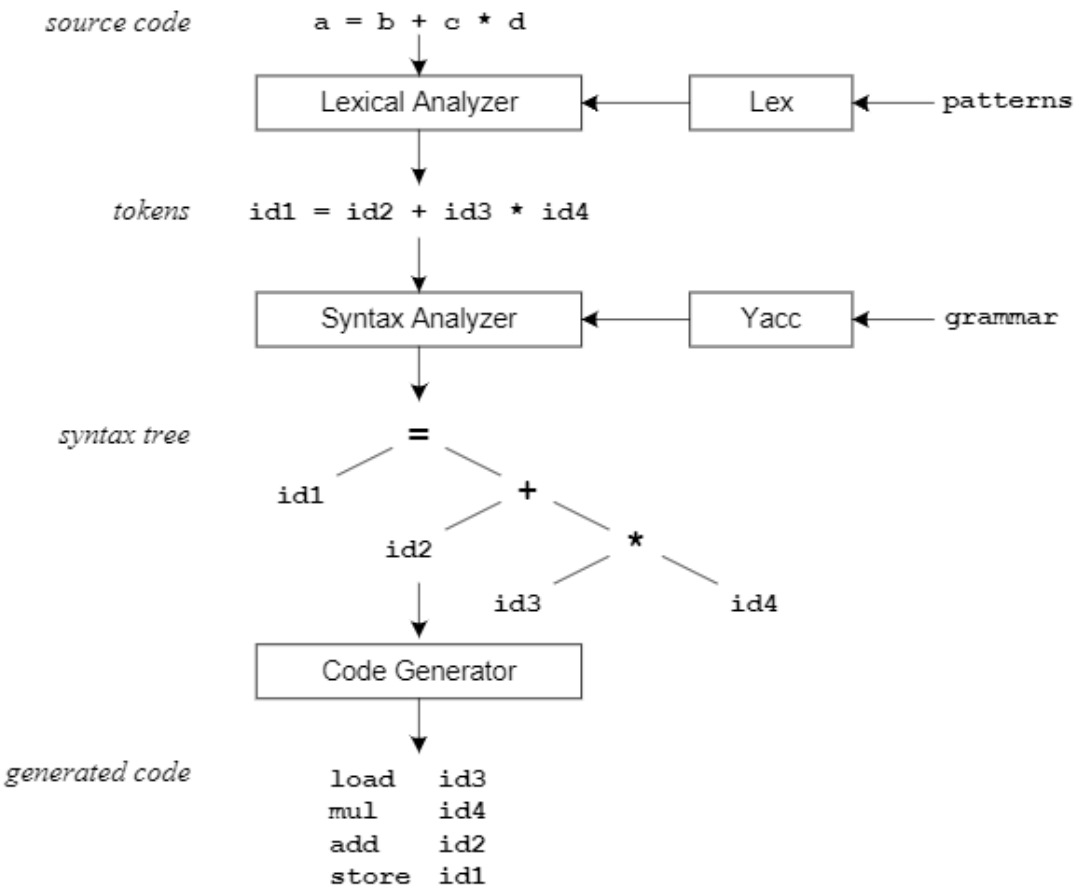

Fig 1: Compilation sequence.

Commands to create compiler:

Here *project.y* is the bison file and *project.l* is the lex file.

bison –d project.y
flex project.l
gcc project.tab.c lex.yy.c –o project

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

Fig 2 illustrates the file naming conventions and the whole process that commands written above accomplish. Yacc/Bison generates a parser that include the function **yyparse** in file **project.tab.c**. Token declarations are included in **project.y** file. Lex/Flex reads the pattern descriptions given in **project.l** which includes **project.tab.h** header, and generates a lexical analyzer, that includes the function **yylex** in the file **lex.yy.c**.

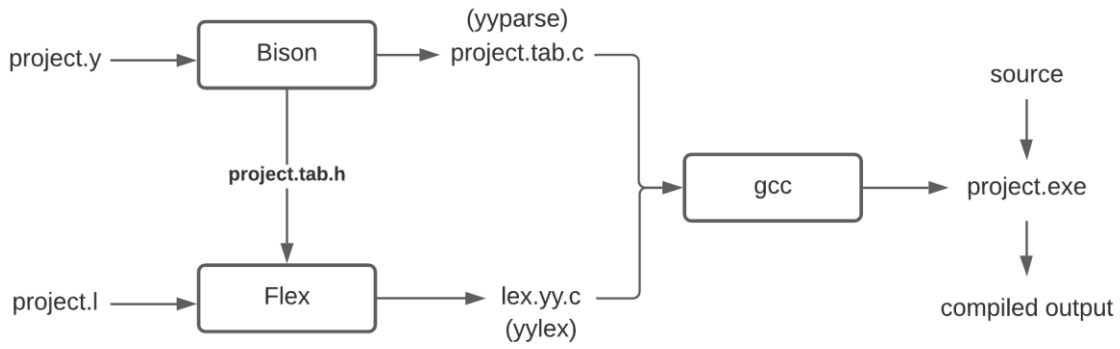

Fig 2: Building *project.exe* compiler with Flex/Bison.

Finally, the lexer and parser are compiled and linked together to create compiler executable **project.exe**. The function **yyparse** is called to run the compiler from **main**. Function **yyparse** automatically calls **yylex** to obtain each token.

Run a program using *project.exe* compiler:

A program can be executed in 2 modes:

1. **Interactive mode** command is *project*.
2. To **run a script** which is in text format,
   command is *project <file-name>.txt*.
   For example, *project sample-programs/8_prime_numbers.txt*.

Documentation:

Topic	Description
Data types	**Numbers:** Default data type. Similar to C double data type. **String:** Similar to C character array.
Variables initialization	All variables are initialized globally. **variablename := expression;** *i := 0;* *firstVariable := -21.5;*

Comment	Single line comment starts with # character. *# This is a comment*
Print statement	Prints expression or string without newline at the end. **print expression;** *print 10+20;* *print firstVariable;* *print string;* *print "Hello world\n";*
Scan statement	Only works in interactive mode. **scan variable;** *scan firstVariable;*
Operators	Precedence: 1. (expression) *print (10+20)/3; Output: 10* 2. ^: Exponent *print 8.3^2; Output: 68.89* 3. not: Logical NOT *print not 0; Output: 1* 4. * : Multiplication *print 2*4; Output: 8* / : Division *print 4/2; Output: 2* % : Modulus *print 11%3; Output: 2* 5. + : Plus

	print 4+2; Output: 6 - : Minus *print 4-2; Output: 2* 6. >= : Grater than or equal *print 11 >= 4.2; Output: 1* <= : Less than or equal *print 11 <= 11; Output: 1* = : Equal to *print 4 = 4; Output: 1* != : Not equal to *print 4 != 4; Output: 0* > : Grater than *print 4 > 2; Output: 1* < : Less than *print 4 < 2; Output: 0* 7. and : Logical AND *print 1 and 0; Output: 0* or : Logical OR *print 1 or 0; Output: 1*
If Else	• **if expression then statement** *if not 0 then print "Hello\n";* • **if expression then statement else statement** *if 0 then print "Zero\n";* *else print "Not zero";*

	• **if expression then statement** **else if expression then statement** **else statement** *number1 := -10;* *number2 := -7;* *if number1 = number2 then print "number1 = number2";* *else if number1 > number2 then print "number1 > number2";* *else print "number1 < number2";*
While loop	**while expression then statement** *i := 0;* *while i < 10 then {* *print i;* *print "\n";* *i := i + 1;* *}*
For loop	**for variable : (start, end, step) then statement** Where, **variable**: A pre-initialized variable. **start**: An expression defining where to start the loop. **end**: An expression defining when to end the loop. **step**: An expression defining the value variable will be incremented with. Example 1: *i := 0;* *for i : (0, 5, 1) then print i;* *Output: 0 1 2 3 4* Example 2: *i := 0;*

	for i : (5, 0, -1) then print i; *Output: 5 4 3 2 1* Pseudocode for example 1: *initialize i ← 0* *set start as reference to i* *set start ← 0, end ← 5, step ← 1* *while (start < end) do* *execute statements* *start ← start + step* *end while*
Exit program	Keyword **exit** terminates the program. *exit;*
Random number	**random(lower, upper)** Returns a randomly generated number within lower to upper. *print random(1.21, 3.15);*
Necessary maths	• **abs(x)** Returns absolute value of given number. *print abs(-21.65); Output: 21.65* • **sqrt(x)** Returns square root √x of given number. *print sqrt(81); Output: 9* • **floor(x)** Returns the nearest integer less than given argument. *print floor(31.91); Output: 31* • **ceil(x)** Returns the nearest integer grater than given argument. *print ceil(31.91); Output: 32*

Logarithm	• **log(x)** Returns natural logarithm of a number x with base e. *print log(10); Output: 2.302585* • **log(x, base)** Returns natural logarithm of a number x with base given as parameter. *print log(100, 2); Output: 6.643856* • **exp(x)** Returns e (2.71828) raised to the power of the given argument. *print exp(10); Output: 22026.465795*
Trigonometry	• **PI** The keyword PI π returns the ratio of a circle's circumference to its diameter. *print PI; Output: 3.141593* • **sin(x)** Returns the sine of an argument (angle in radian). *print sin(PI/2); Output: 1* • **asin(x)** It takes a single argument (-1 ≤ x ≤ -1), and returns the arc sine (inverse of sin) in radian. *print asin(0.5); Output: 0.523599* • **cos(x)** Returns the cosine of an argument (angle in radian). *print cos(PI/2); Output: 0*

	• **acos(x)** It takes a single argument (-1 ≤ x ≤ -1), and returns the arc cosine (inverse of cosine) in radian. *print acos(0.5); Output: 1.047198* • **tan(x)** Returns the tangent of an argument (angle in radian). *print tan(PI/4); Output: 1* • **atan(x)** It takes a single argument, and returns the arc tangent (inverse of tangent) in radian. *print atan(0.5); Output: 0.463648*

GitHub link of this project is provided here.

References:

1. LEX &YACC TUTORIAL by Tom Niemann
2. https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)
3. https://book.huihoo.com/compiler-construction-using-flex-and-bison/YaccBison.html