FLEX / Bison Calculator Implementation with Calculus Functionality

(revised November 2018)

Jonah Kubath, Ioannis Nearchou, Western Michigan University

**Abstract**—Calculators and their functionality can be broken down into a set of expressions to which an operation is applied. FLEX stands for the fast-lexical analyzer generator. It is used to generate a program that can recognize lexical patterns from a given input. Bison is a parser generator that takes in a grammar and outputs a parser table. FLEX is used to read input from the user and break the input into defined tokens. Bison accepts these tokens and applies the generated rules to the sequence of tokens. The calculator functions that were implemented include: addition, subtraction, multiplication, division, modulo, power, sine, cosine, and tangent. The calculator can also recognize and handle basic calculus function of derivatives. Derivatives of polynomials and certain geometric functions are defined.

**Index Terms**— FLEX, Bison, Calculator, Calculus, Lexical, Analyzer, Parser, Tokens, Patterns

—————————— ◆ ——————————

# 1 Introduction

T

HIS document outlines the semester project for CS5800 done by Jonah Kubath and Ioannis Nearchou. The problem task is to take input from the user, in our case through the command line, and apply calculator operations on the given expression. The calculator is built to handle infix expressions of the form <expression> <operation> <expression>. Operation precedence is given to power, multiplication, and division.

The input is handled with a lexical analyzer. The lexical analyzer will have a set of defined tokens such as “+”, “-“, etc. The defined tokens are held in a lex file. The analyzer generator will read the lex file and output a lex.yy.c file. A parser generator will read defined grammars in the parse.y file and convert the grammars into the corresponding C code. Once the two files are generated with FLEX and Bison, the lex.yy.c and parse.tab.c can be compiled together to form one executable file. This executable can then be run to receive the user’s input and output the result.

# 2 Various Methods

## 2.1 Lexical Analyzer

The first step in implementing a calculator is receiving input from the user. The input is read in through the command line or terminal and passed to the lexical analyzer to be broken into tokens. The main idea for lexical analyzers was developed by Mike Lesk and Eric Schmidt in 1975 [3]. This idea from Lesk and Schmidt released one of the popular lexical analyzers called LEX. One of the competitors to LEX is FLEX. The benefits of FLEX when compared to LEX are improved run times and table compression. The run time for flex is approximately two times faster than LEX. The generated parser table is compressed to approximately 17 times smaller than that of LEX too [4]. An alternative option for lexical analyzers is Quex. Quex is a very modern lexical analyzer generator, but with most of the references to lexical analyzer generators pointing to LEX or FLEX, development with Quex would pose to be more difficult.

A lex file is what the lexical analyzer generators read to output the given C code to create the parser. Lex files are broken into three sections. The first section is for definitions. This section allows the user to define variables, macros, and include any needed header files. The second section is the main section. It contains all the definitions that the parser should recognize and what tokens should be returned when the patterns are found. The patterns are defined with string literals or regular expressions. An example of a pattern to recognize is “+”, which causes the PLUS token to be returned. This defines one of the operations that the calculator will handle, addition. The final section of the lex file is reserved for any addition C code that program would want defined for added features. The generators will take this C code and copy it into the generated lex.yy.c file.

## 2.2 Handling the tokens

Once the input from the user is parsed, it is passed to a function that will handle the tokens based on defined rules. A parser generator is used to take defined grammars and create a parse tree. The parse tree defines how to match the sequence of tokens to the data elements [7]. There are an abundant number of options for parser generators covering all of the main programming languages for output [8]. Since the project is implemented in C, Bison is a great choice with direct support for FLEX. Bison is a parser generator that converts context-free grammars into a deterministic LR parser [2]. “LR” stands for left-to-right parsing and a right most derivation in reverse order [9]. The grammar file that Bison uses is defined similarly to the lex file. The first section is reserved for definitions, included headers, and can contain user defined C code. In between the first and second sections, the expected tokens returned from the lexical analyzer are defined. %token is used to define tokens and %left or %right is used to give precedence to the tokens on the respective sides. Precedence is also given to the rules in a top-down fashion. Highest precedence is given to the first line of tokens and decreases as the lines of tokens are defined going down. The second section of the grammar file is the parsing ruleset. These rules layout the format for matching a sequence of tokens. They also define a segment of C code to go along with the rule. Once a rule is matched, the defined fragment of C code is executed. The final section of the grammar file is for additional C code that can be called when rules are matched. Functions can be implemented in this section and will be copied by Bison to the parse.tab.c output file. A main function must be implemented in this section calling the yyparse() function to read the input from the lexical analyzer.

# 3 Implementations

## 3.1 Lex File

As stated in the Various Methods section 2.1, the lex file is used by the lexical analyzer generator to create the C code needed to parse the input from the user. The lexical analyzer generator that was chosen for the project is FLEX. The input from the user is sent as input to the analyzer which parses and tokenizes the characters. The parsing is based off of a deterministic finite automaton (DFA) [10]. The patterns that are recognized are in the definition sections of the lex file. An example of the patterns that are matched is a float value with one or more digits.

[0-9.]+

The patterns are in the form of regular expressions. “FLEX translates all of the regular expressions into an efficient internal form that lets it match the input against all the patterns simultaneously, so it’s just as fast for 100 patters as for one. “[11]

## 3.1.1 Definition Section

The first section of the lex file starts with %{ and ends with }%. In this section, the program can define data structures, define macros, or include header files. In the calculator implementation, numbers and strings must be accepted and parsed. The numbers are related to the basic calculator functions and the strings related to handling derivatives of polynomials. To handle both types of input the data\_s struct is defined. This struct contains a character pointer to hold a string and a double to hold any number. To allow proper communication between the lexical analyzer and Bison grammars, a value must be used to pass the data. The YYSTYPE macro is used to define the data type that FLEX and Bison should expect for this data. For our implementation, the YYSTYPE macro is defined to the data\_s struct to pass either strings or doubles to the Bison grammars. An important file to include in the definition section is the \*.tab.h file generated by Bison. For our implementation, this file is called “parse.tab.h”. This file contains definitions for the tokens that Bison expects. The final declaration in the definition section is extern YYSTYPE yylval. This is the variable that is given global scope and passes data between FLEX and Bison.

## 3.1.2 Regular Expressions

The second section of the lex file begins and ends with %%. This section is designated for the regular expressions against which the input is compared. In the scope of the project, this is an important and complex section that can be simplified. The general outline for the rules are <Regular Expression> { C code; return TOKEN; }. Most rules are string literals, defining operations such as “+”, “-“, or “cos”. The remaining rules are used for reading whitespace, reading numbers as doubles, reading polynomials, handling a derivative, and reading a newline character. Reading whitespace is simple. When a space, tab, or carriage return is read nothing is done. The numbers take the regular expression form [0-9.]+. This form allows numbers to contain one or more digits and the period character. For reading polynomials, the regular expression for numbers is expanded to accept letters, “^” for powers and “–“ for negative. The regular expression is defined as

[a-zA-Z0-9.^-]. This allows the lexical analyzer to match characters with the rules. When a rule is matched, C code or a return TOKEN can be defined. For the string literals that handle operations, such as “+” or “-“, a token is returned that describes the operation. When “+” is read, return PLUS. When “-“ is read, return MINUS. It becomes a more complicated when data must be copied or modified but using C code alleviates this complication. When a number is read, the string is saved by FLEX in the yytext variable. This variable is sent to the sscanf function defined in the stdio.h header file. This function saves the long float or double held in the yytext variable to a given double variable. In our case, we pass the data\_s struct’s pointer as the double. This algorithm is then repeated for handling polynomials and the sscanf function is replaced with the strcpy function. The yytext data is then saved to the data\_s struct’s char \* variable. This concludes the second section defining the regular expressions for the lexical analyzer.

The third section reserved for supporting C code is not needed for the calculator implementation.

## 3.2 Bison Grammar File

At this point in the program, part or all of the input string has been parsed and tokens have been sent to the Bison generated parse.tab.c file. The parse.tab.c file is generated by calling Bison on a grammar file. Bison grammar files are broken into the same sections as the Lex file. The first section is for definitions and included headers, the second section for grammar definitions, and the final section for assisting C code.

## 3.2.1 Definition Section

The first section is for definitions and included headers. In the calculator implementation, the YYSTYPE macro must be defined. In our case, it is defined as the data\_s struct. This struct contains a pointer to a character array, a number held as a double, and a character used for derivatives. Once the YYSTYPE is defined, needed C code headers are included and assisting C code functions are defined. A function that must be defined is yyerror(). This function is called when the input received from the lexical analyzer cannot be matched to a defined grammar.

In between the first and second section, the tokens are derived in order of their precedence. An example line is:

%token LEFT\_B RIGHT\_B

In this line, the tokens for the left and right parentheses are defined. The list of tokens will be used when parse.tab.h is generated.

## 3.2.2 Grammar Definitions

The second section is reserved for defining grammars. Bison accepts context-free grammars and converts them into a deterministic LR parser [2]. Bison is compatible with Yacc (Yet Another Compiler-Compiler). This allows any proper Yacc grammars to also work with Bison. A basic yacc grammar is used to define a sequence of tokens. The outline for grammars is:

<symbol>: <token, literal, or symbol> {C code to execute};

The <symbol> is just a name for the grammar definition. Tokens must be previously defined and are sent from the lexical analyzer. A literal can also be placed inside the grammars. Bison rules are nearly Backus-Naur Form. Some of the punctuation is simplified to assist in typing the rules [11]. When rules are created, Bison also creates a value to go along with it. The value of the target symbol, the symbol that is on the left of the “:”, has a name of $$. The values to the right of the semi-colon have values of $1, $2, etc. counting up to the number of symbols. When the grammars are defined, a link is created between the grammars by allowing the current grammar to attempt fitting the next defined grammar. Here is an example:

add: mult\_div

| NUMBER PLUS NUMBER;

In the example above, the add symbol or rule is defined. This symbol will first attempt to match the given tokens with the mult\_div symbol. If this fails, it will then attempt to match the second rule of NUMBER PLUS NUMBER. The C code to accompany the NUMBER PLUS NUMBER grammar would likely be { $$ = $1 + $3 } as it adds together the values held in the first and third symbols saving the result to the current symbol. This general outline for grammars is followed:

Addition / subtraction

Multiplication / division

Power

Functions – cos, sin, tan

Derivatives

Polynomial addition / subtraction

Saving a number or string

Reading parentheses

This outline allows Bison to continue searching for the grammar that matches the given tokens.

## 3.2.3 Assisting C Code

The third section allows the program to implement additional functionality through C code. These functions can then be called when grammars are matched. The first function implemented was an improved modulo function. Because modulo is computed over integers, but internally stored as doubles, if the user enters doubles, the data must be converted to integers. This conversion can cause a loss in data that the user was not expecting. Adding this additional function allows the user to be notified that the values were cast to integers.

The second, main, function was implemented to handle derivatives of polynomials. The input to this function accepts a character string of the polynomial along with the character of which the derivative is taken in respect. The polynomial is then broken into the individual sections of variables and numbers. Each individual section is then analyzed. The value is first checked to see if a negative sign is accompanied. If it is, then a negative flag is set. After the check for negative sign is finished, the value is broken into five sections.

1. Coefficients
2. Variables and their powers
3. The variable being derived
4. The variable’s power
5. Variables after the derived variable

After the sections are found, the derivation becomes straight forward. The variable being derived multiplies the coefficient by the current exponent. This exponent is then subtracted by one. After this is done, the string is recombined handling any output changes such as an exponent to zero will not print the variable. Once the five sections are combined to one value, the negative flag is used along with the current value to combine the final result back to the returned string.

The next function to be implemented is the derivative of functions such as sin, cos, and tan. This derivation is defined by deriving the function and multiply it by the derivation of the polynomial inside the function. The derivation of the function is set and defined by calculus. The derivation of the inner polynomial is simple as the previously defined function can be used to determine the derivative.

# 4 Testing

Testing is designed to fully cover the various states that the application can be put in through user input. In the case of a calculator, the easy test cases include testing the basic functions. Does adding two numbers return the correct result? Does multiplying two numbers and adding the result to another number return the correct result? These situations are not extremely difficult to iterate for testing although validating the results is important. These tests can be created with the following user input:

1 + 1 = 2

2 \* 9 = 18

2 \* 8 \* 5 = 80

2 + (5 \* 2) = 12

(1 + 2) \* (4 + 5) = 27

The derivative of polynomials and functions can be testing through the given user input:

dx(1) = 0

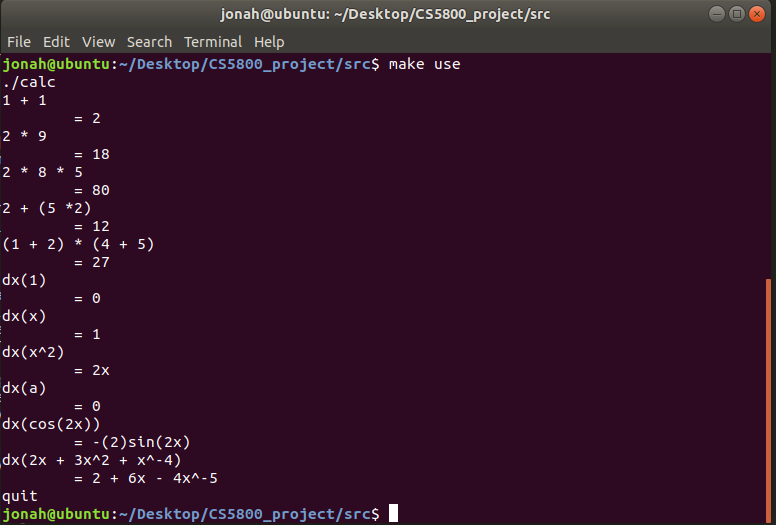
dx(x) = 1

dx(x^2) = 2x

dx(a) = 0

dx(cos(2x)) = -(2)(sin(2x))

dx(2x + 3x^2 + x^-4) = 2 + 6x – 4x^-5



# 5 Results

The purpose for this project was to implement a set rules that could be followed for various user inputs. The functionality would be based upon an arithmetic / geometric calculator with the basic calculus function of derivatives. A lex file was created to parse the input from the user and break it into the tokens needed by the grammar parser. The grammar file was created with context free grammars to implement calculator functions along with C code to handle derivatives of polynomials and trigonometry functions. The testing has validated that the project was successful in implementing these functions.

# 6 User Guide

## 6.1 Compilation

A makefile is used to automate the building the of the lexical analyzer, grammar generation, and executable compilation.

~$ make

This command will call flex to generate the lexical analyzer C code. The actual command is

~$ flex calc.l

Make will also call Bison to generate the parser C code. The actual command is

~$ bison -d -t parse.y

Once these two commands are called, multiple files are created. Lex.yy.c, parse.tab.c, and parse.tab.h are generated. These files can then further be compiled into an executable for the current operating system. The command uses an open source C compiler, GCC.

~$ gcc -o calc lex.yy.c parse.tab.c -lfl -lm

The generated executable can be run through the command line by calling

~$ ./calc

## 6.2 Input

The user can enter standard in-fix expressions. The defined expressions can include the operations: addition, subtraction, multiplication, division, modulo, power, cos, sin, tan, csc, sec, cot, and the calculus function of derivative for polynomials and the previously stated trigonometric functions.

Examples of input:

1 + 1

(1 + 1)

2 \* (4 + 1)

cos(2)

4.5 + 6.7

dx(x)

dx(2ax^2)

dx(cos(2x))

**References**

1. westes/flex. (n.d.). Retrieved December 1, 2018, from https://github.com/westes/flex
2. Bison - GNU Project - Free Software Foundation. (n.d.). Retrieved December 1, 2018, from https://www.gnu.org/software/bison/
3. Wikipedia contributors. (2018a, November 25). Lex (software) - Wikipedia. Retrieved December 1, 2018, from <https://en.wikipedia.org/wiki/Lex_(software)>
4. Srivastava, A. (1991). LEX VS. FLEX. Retrieved from http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=350072
5. Introduction — Quex Lexical Analyzer Generator 0.64.8 documentation. (n.d.). Retrieved December 1, 2018, from http://quex.sourceforge.net/doc/html/intro/intro.html
6. Wikipedia contributors. (2018b, November 29). yacc compatible parser generator. Retrieved December 1, 2018, from https://en.wikipedia.org/wiki/GNU\_Bison
7. Bison Tutorial. (n.d.). Retrieved December 1, 2018, from http://alumni.cs.ucr.edu/%7Elgao/teaching/bison.html
8. Wikipedia contributors. (2018c, November 29). Wikimedia list article. Retrieved December 1, 2018, from https://en.wikipedia.org/wiki/Comparison\_of\_parser\_generators
9. LR parser - javatpoint. (n.d.). Retrieved December 1, 2018, from https://www.javatpoint.com/lr-parser
10. Wikipedia contributors. (n.d.). Flex (lexical analyzer generator) - Wikipedia. Retrieved December 1, 2018, from https://en.wikipedia.org/wiki/Flex\_(lexical\_analyzer\_generator)
11. John Levine, J. L. (n.d.). flex & bison. Retrieved December 1, 2018, from https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html