Ioannis Nearchou

Jonah Kubath

CS 5800 Theory Foundations

11/20/2018

**CS 5800 Project Preliminary Design: YACC/BISON Calculator Implementation**

This paper outlines the design of a project to implement a yacc/bison description of a calculator.

**Project Approach**

As discussed in the previous project proposal, a calculator implementation using FLEX is designed. To implement the calculator, a grammar file will be written by this project group to handle expressions input by the user into the console. When an expression is being parsed by FLEX, the rules and transitions for parsed parts of the expression will be engaged, culminating in the return of a number to the console that is the calculation of the expression input by the user. Our possible approaches include writing a grammar that works for infix user expressions, for arithmetic and geometric mathematical expressions. The arithmetic and geometric expressions include standard multiplication, division, subtraction, addition, modulo, power, sine, cosine, tangent, etc. The calculator will be expanded by handling the calculus operation derivative. The derivative of polynomials, sine, cosine, tangent, cosecant, secant, and cotangent are to be implemented.

**Current Research Efforts**

Resources have been found for explaining parser grammar rules, including a class handout for a simple calculator and through <https://www.ibm.com/developerworks/aix/tutorials/au-lexyacc/index.html>, for FLEX in particular. These provide enough examples in what to consider for handling order of precedence with expressions, to govern what grammar we may write for our implementation.

**Design Flow**

The design of the project is as follows:

1. Download FLEX and associated tools.
2. Create calculator grammar and input parser files.
3. Call flex and bison to convert lexical analyzer and grammar files
4. Test project by the three approaches to expression format, with generated input.

**Implementation**

The first section of the project is to implement the lexical analyzer. This section is designed to take the input from the user and break the strings into the largest defined tokens. As stated in the Project Approach, the project will implement basic arithmetic and geometric expressions.

Rules to handle these expressions:

%%

“+” return PLUS;

“- “ return MINUS;

“\*” return MULTIPLY;

“cos” return COS;

%%

These rules are continued for all the arithmetic and geometric expressions expected from the user. To handle derivatives, the lexical analyzer must also read strings. These strings include coefficients, variables of the polynomial, exponents, addition, subtraction, multiplication, and division. The rules created to handle strings are as follows:

%%

"d"[a-z] return DERIV;

[a-zA-Z0-9.^+\-\*\/]+ return STRING;

%%

The first rule will trigger the token of a derivative with the proceeding character telling the calculator with respect to which variable the derivative is taken. The second rule will read the sections designed for the polynomials. C code is written to copy the data, such as numbers, characters, and operators, to a data structure containing the number as a double or the string as a pointer to a character array.

struct data\_s {

char \* s;

double num;

} YYSTYPE;

The second section of the project contains the grammar file. This file is used to interpret the tokens that the lexical analyzer will pass to it. The same data structure is defined that contains a character pointer and a number as a double. All the return tokens from the lexical analyzer must be defined in the grammar file.

%token EQUALS NUMBER POWER MODULO EOLN

%token SIN COS TAN CSC SEC COT

Now that the grammar file will know what to expect as input, the rules on how to handle the input can be defined. The first command is a generic rule that will continue to parse all the input until empty.

%%

commands:

/\* Empty \*/

| commands EOLN

| commands exp EOLN

| commands error EOLN;

%%

The following rules are created with a filter down effect. The grammar file will attempt to match the tokens with best fitting rule that is defined.

%%

exp: add\_sub

;

add\_sub: mult\_div

| add\_sub PLUS mult\_div { $$.num = $1.num + $3.num; }

| add\_sub MINUS mult\_div { $$.num = $1.num - $3.num; }

mult\_div: power

| mult\_div MULTIPLY power { $$.num = $1.num \* $3.num; }

| mult\_div DIVIDE power { $$.num = $1.num / $3.num; }

| mult\_div MODULO power { $$.num = calcMod($1.num, $3.num); }

%%

These rules are continued for all the functionality that can be passed from the lexical analyzer. As in the lexical analyzer, the grammar file allows for C code to be written to handle the operations. Some of the basic arithmetic, geometric, and functions are already defined and can used by including <math.h>. The calculus functions must be manually implemented. The grammar file can handle derivatives of polynomials and the basic trigonometric functions. The derivative of a polynomial is done by breaking the given string into pieces by space separated values. The individual values are then split into four sections. The leading coefficient, variables left of the variable being derived, the derived variable and its exponent, variables right of the variable being derived. The derivative of a polynomial is defined by multiply the coefficient by the exponent and then subtracting one from the exponent. This is done for all the sections of the polynomial and then concatenated into one string at the end.

The derivative of the trigonometric functions becomes easier when the derivative of polynomials has previously been defined. The derivative of the function is defined by taking the derivative of the function multiplied by the derivative of the polynomial inside the function. Sine will turn to cosine and cosine will turn to negative sine when derived. The inner polynomial is then passed to the previous function and multiplied to the resulting function derivative.

**Testing**

Testing the functionality of a calculator is tested by passed different variations of expressions and verifying the output as correct.

Basic arithmetic and Geometric expressions

1 + 1 = 2

2 \* 9 = 18

2 \* 8 \* 5 = 80

2 + (5 \* 2) = 12

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

Testing the functionality of derivatives can be done by passing polynomials and trigonometric functions and verifying the output as correct.

Derivative expressions

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