Benjamin Steenkamer
CPEG 621-010
3/13/19

Lab 1 Report: A Calculator Compiler Front End

Abstract:

The goal of this lab is to create a simple calculator that supports basic operators and user defined variables using lex (flex) and yacc (bison). A secondary goal of the calculator is to have it convert the infix input equation into prefix form. The motivation to create this calculator is to learn how to use flex and bison, which are two main tools used in the creation of compiler front ends. By creating the calculator using these tools, I gained insight into how a compiler divides a program into tokens and then finds meaning in those tokens with grammar rules. Because of this project, I now know how to specify token recognition rules in flex, define my own grammar and supporting logic in bison, and understand how flex and bison interact with each other. In short, I now understand how to create a lexer and parser, which are key components in the creation of a compiler front end.

Overall, two calculator programs were successfully created using flex and bison. The first is an infix calculator which supports single-line calculations using user defined variables and basic operators. The second program generates a prefix version of the input infix equations. This version uses a stack and concatenation method to convert the postfix string inherently produced by the recursive bison grammar rules and converts it to prefix notation. Both programs operate to the lab requirements using my own flex token rules, bison grammar rules, and C helper functions.

Detailed Strategy:

There are 5 main program files that create the calculator functions for this lab: `infix.l`, `infix.y`, `Makefile`, `prefix.l`, and `prefix.y`. The "infix" files implement part one of the lab, which is an infix expression calculator with support for multiple operators (+, -, *, /, !, **), parenthesis, and user defined variables. The prefix files build on top of the infix calculator to output the prefix notation of the inputted infix equation, along with also calculating the normal output and supporting all the features of the regular infix calculator. The lex files (.l) for both parts are effectively identical, as they implement the same scanning rules. The yacc files (.y) are where the main differences occur.

The strategy for implementing the lex rules was to simply only allow the tokens needed for the proper calculator operation and rejecting everything else as invalid tokens. These lex rules tokenize the input equations and pass them on to yacc for more processing. Going through the rules, the first one is `[a-z][A-Za-z0-9]*`, which recognizes a character sequence as a variable name if it starts with a lower case letter and is followed by zero or more alphanumeric characters, upper or lower case. The variable is saved to yylval to pass to the yacc rules, and the VARIABLE token is returned. The next rule is `[0-9]+`, which matches one or more numerals from 0 to 9 in a row. Lex tokenizes this as an INTEGER and the integer value is placed in yylval. The "`**`" rule recognizes two consecutive asterisks as the exponential (power) operator, and it returns a POWER token type. The `[-+()=*/!]` identifies any one of the accepted math operators, including bitwise not (!) and parenthesis. These operators are passed as character literals

for the token type. `"\n"` and `[ \t]+` deal with accepting a newline and passing it to the yacc rules and ignoring one or more white-spaces by not sending any token for them. Finally, `.` states that any other character encountered while scanning will result in an "invalid character" message being printed. In the calculator's processing flow, the yacc rules continuously call the lex file and use these rules to tokenize the next part of the math equation input.

From here, the yacc rules file was created to define the grammar and process the tokens, which it gets from calls to the lex file. First focusing on `infix.y`, the definitions of the INTEGER, POWER, and VARIABLE tokens are stated, a union is used to allow yylval in the lex rules to hold different data types (int or string), and the token types are declared. These definitions are needed by the lex rules, so they know what the custom token types are and how to pass values to the yacc rules. The grammar associativity and precedence for the operators are also given. This is done to make the grammar rules unambiguous when yacc/bison is deciding to shift or reduce an expression. The associativity and precedence of the operators was taken from the PEMDAS rules and C's normal definition of operator associativity and precedence.

In the next section of the yacc file, the grammar rules are defined for the various tokens. The first rule states the general flow of the program as starting with a `statement` followed by a new line. Next, a `statement` is defined as an `expr`ession or `VARIABLE` assignment. When an expression is fully resolved via recursion, its value is printed out. When a variable assignment is detected, a variable name is allocated in a global struct array, set to the assigned value by a helper function, and then its value is printed out. Below these are the expression grammar rules.

When a `VARIABLE` token is encountered, the rule resolves it by fetching its value from storage, or it creates a new variable if it doesn't already exist. The next four rules are the grammar for +, -, *, and / operations, which must follow the syntax of an operator token surrounded by two expressions. These grammar rules simply perform the math operation represented by the token on the two expressions. The bitwise not operator is a unary operator, so it must be followed by an expression to the right. Since the power token was defined as the literal "**", an equation with a space between the asterisks will not be recognized as the power operator. Finally, the parentheses rule is defined such that an expression must be surrounded by matching parentheses. They are then stripped away, leaving only the expression. From this rule, if there is an unmatched parenthesis in the input equation, a syntax error is thrown. In general, if an operator token is encounter but does not exactly match its grammar rule, a syntax error will be thrown.

The major design decisions and tradeoff for this calculator occurred in the creation of variables. When a new variable is declared or used for the first time, it is added to an array of structs that contain the variable name and value. When the variable used, the name is searched in the array to find the entry and return the value. This simple struct array was chosen because it was a simple means of keeping track of the defined variables. The down side of this implementation is that there is a max variable name length and max number of declared variables, since the struct array and strings are a fixed size. If either of these values are exceeded, a warning message will be printed for the specific case and no new variable will be created. This limit is artificial for the purposes of this lab, and they are made large enough to be avoided without much effort. Given enough time, a system could be created to dynamically allocate arbitrary amounts of variables and arbitrary name lengths.

The second part of this lab included many more design decisions. First, the infix `.l` and `.y` files were copied and turned into `prefix.l` and `prefix.y`. The lex rules for `prefix.l` are identical to `infix.l`. `prefix.y` builds off of the `infix.y` by adding more helper functions to create a prefix version of the input equation and then output the equation answer like normal. For the general method of this prefix creator, I discovered that when the grammar rules resolved recursively, they created a postfix version of the infix equation. I realized that if I store the expressions as they are resolved, I could create a buffer of the postfix equation. I can then reverse the postfix equation after the statement is fully resolved to create the prefix version.

`prefix.y` does this by first adding to the `infix.y` grammar rules and using a new buffer that resolved expressions will be pushed to in postfix order. The first change was to the `expr` rule, which now calls the function to print the prefix form and then prints the equation result as usual. The variable assignment rule pushes the variable name and "=" sign to the postfix buffer and then calls the print prefix function. For the grammar rules under the `expr` type, the integer, variable name, or operator symbol are now pushed into the postfix buffer along with doing the normal resolving code. The parenthesis grammar rule doesn't use the push call as they are not needed in pre- or postfix forms.

Once the statement has been resolved, the postfix buffer will then contain the full postfix equation. When the call to print the prefix version is made, I run an algorithm to reverse the postfix version and convert it to the prefix version, which is stored in a stack. I then print out this prefix version. The algorithm is shown in the flow chart below.

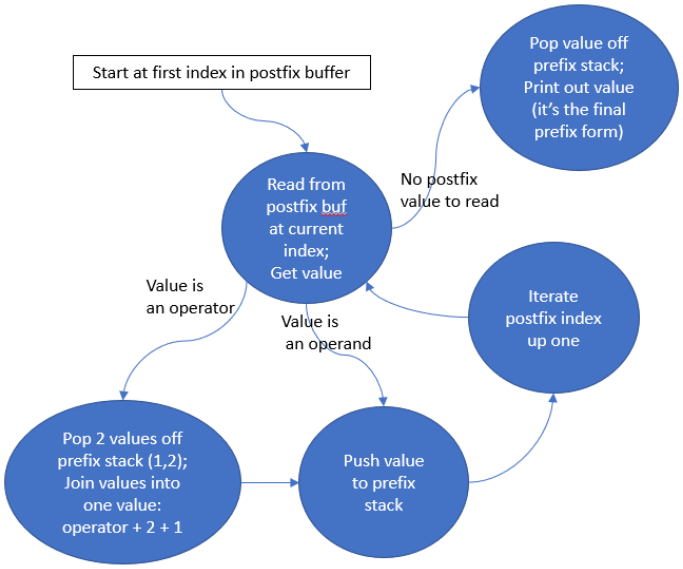

Fig. 1: General algorithm to convert postfix to prefix

Within this print prefix function, there are several helper functions used to determine special cases not explicitly shown in the flow chart. The code structure and comments make it clear what these special cases are for, but the general algorithm above still holds. Within this algorithm are checks used to prevent buffer overflows with the value concatenation. This is because all the buffers and the stack used in this implementation have a finite amount of space, which is a major drawback. If the prefix equation gets too large for the stack index, the size check functions will cut off data to prevent overflow. The max sizes for these buffers are made very large

so this cutoff won't occur easily. Finite size data structures were chosen for ease of implementation and to avoid potential complications involved with dynamically allocating and managing the buffers.

Results:

The overall results of this lab are that both versions of the calculator successfully perform all the required operations. That is, they are able to process infix expressions in the correct order with custom user variables and the use of any of the supported operators (+, -, *, /, !, **). The prefix converter also works, by correctly generating a prefix version of the infix equation.

To verify this correctness, I ran many test cases through the calculators. I first used the simple test cases provided in the lab to verify basic functionality of all the operators. I then went online and found long equations that required the use of order of operations. These equations included all the operators specified in the lab and used constants and variables throughout. I entered the equations first into Python's interpreter to verify the results, and then entered them into my calculator. Every one of the equations outputted the same answer provided by Python.

I specifically tested the lex and yacc rules by entering in characters and expressions I knew should cause an invalid character or syntax error. I entered strings for each lex rule and verified they only recognized the tokens I wanted for variable names, operators, and integers. For the yacc rules, I saw that all the malformed equations I entered correctly resulted in syntax errors when they did not explicitly follow the grammar rules. This included missing parentheses, multiple operators in a row, operators not preceded or followed by expressions, and bad variable assignment.

For the prefix notation, I found online problems for converting infix equations to prefix notation with provided solutions. I entered the infix notation into my calculator and saw that the output matched the online solutions. I then converted the prefix notation back to infix by hand to make sure they were correct. I also verified that the simple prefix examples provided in the lab worked.

Finally, I ran tests to check the limits of the finite array sizes, the number of variables, and variable name lengths. In the finite array tests, I found my bounds checking properly detects and stops buffer overflow when very long equations are entered. For both the infix and prefix calculators, the equation expansion is limited to 200 stack or buffer items. This is a large enough limit to preform most reasonably long equations and convert them to infix notation without issue. I also tested the max variable name length and number of declared variables and saw my checks properly prevented these limits from being exceeded. These are a limit of 30 total variables declared at one time and a variable name being at most 20 characters. Again, these finite limits are artificial, and with more time, a dynamic memory allocation system can be created to allow arbitrarily long expressions and variables. Beyond these limits, both the infix and prefix calculators operate as expected of the lab and produce the correct results.

Conclusion:

Overall, I thought both parts of this lab were challenging to do. My implementations for the infix and prefix calculators produce the correct results, but they are limited by the finite buffer sizes. This limitation is the main weakness of my project, and with more time I would create a system to dynamically allocate the postfix buffer and prefix stack to allow an arbitrary amount of

operations to be processed. This would also allow for an arbitrary number of variables and length of variable names. One strength of my project is that there seem to be no errors in the functionality when operating within the finite sizes. Another strength of the lab is the bounds checking I did, which should prevent a memory error from occurring when one of these limits is reached. Beyond this, the areas that gave me the most difficulty were the interfacing functions between lex and yacc, understanding the yacc grammar syntax and functionality, and the initial implementation of converting infix to prefix notation. In general, my calculators correctly perform all the operations required by the lab, so I am satisfied with the overall outcome.