Project 1. Simple Calculator Interpreter.
COSC 252
Jeremy Bolton

Over the course of the term you will design and build programming languages. The first is a simple calculator language. Specifically, you will design and build an interpreter for a Calculator as specified below.

Instructions:

I. Language Description. The simple calculator programming language, which we will refer to as calc, is described as follows:
   a. The basic syntax will consist of parenthetical operations in prefix notation. There will be binary and unary operations.
      i. Arithmetic Operations
         - +, -, *, /
      ii. Relational
         - >, <, >=, <=, =
      iii. Logical
         - OR, AND, NOT
      iv. Literals
         - Integers, floats, #t, #f
      v. Operations
         -
   b. Example Usage:
      i. (+ 3 4)
      ii. (- 2)
      iii. (> (+ 2 5) (- 3 9))
      iv. (not
            (or
              (> (+ 1 2) (/ 3 2))
              (not (= 3 (+1 2)))
)
)

II. Design and build cacl
   a. Design tokens and build tokenizer
      i. Create a list of all tokens. Use a two column table. The first column will list the token name or category and the second column will be either the token(s) itself (if it only takes 1 form) or a definition of the token using a regular expression or FSM, if it has many forms (all lexeme representations). For ease I encourage you to simply include this as a comment in your source code for the tokenizer.

Token	Lexeme(s)
id	(regex for ids)
plus_op	+
int	(regex for ints)
float	(regex for floats)
…	…

ii. Using C or C++, implement a tokenizer for cacl. Include your token list and definitions in the comments of your source code.
- Input to tokenizer: string
- Ouput: sequence of tokens

iii. Error Handling: If there is a tokenizer error, please throw a tokenizer error. (This will likely only occur if you encounter a symbol that is not in the input alphabet.)

b. Design grammar and build parser.

i. Define/Design the grammar using BNF. (For ease feel free to include this in comments in your source code for your parser.) Use a start symbol named <program>. The goal is to have an unambiguous grammar, where the standard operator precedence is observed, intuitive combinations of syntactic structures (sequentially and nested) are permitted, and all sentence structures described in the description can be derived using the productions.

ii. Using C or C++ implement a top-down recursive descent parser. Include the BNF production set in the comments at the beginning of your parser source code.

iii. NOTE:
- Revisit your grammar *to assure that it can be recognized by a top-down parser.* The crux of a top-down parse is to correctly predict a RHS when resolving a LHS abstraction. You will want to make this prediction as easy as possible. Try to keep the number of lookaheads to 1 (or less) for each possible decision.
- Error Handling: If there is a parse error, provide an intuitive error message should be printed to the console, the input buffer should be cleared, and control should be returned to the main method, where the interpreter will continue in interactive mode.
- Hint … Debugging help:
  a. Test your parser before adding semantics!!!
  b. Have your parser compose a string that illustrates the path of the function call chain during the parse, (this mimics the parse tree). Use the parse tree rendering using tool (http://mshang.ca/syntree/):

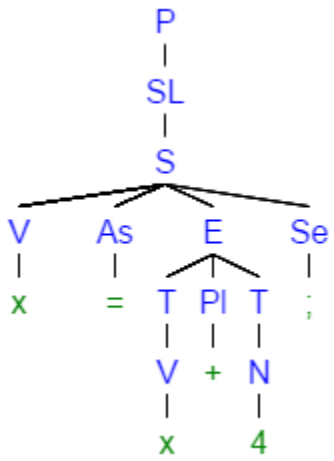
C++ Syntax Example (from previous term!) for testing / debugging your parser:

Sample Input: x = x + 4;

Resulting string to test parser (given some specified grammar):

[P [SL [S [V x] [As =] [E [T [V x]] [Pl +] [T [N 4]]] [Se ;]]]

Resulting Parse Tree Rendering using web tool:



c. Implement Semantic Analysis.
   i. Incorporate semantic evaluation directly into the parser.
   ii. Allow for interactive interface (similar to https://scheme.cs61a.org/)
   iii. Examples:

```
>>  (+ 3 2)
>> 5

>> (or
      (> (+ 1 2) (/ 3 2))
      (not  (==  3 (+1 2) ) )
   )
>> true
```

III. RUBRIC

Requirements		**Points Allocated**

RegEx design + Tokenizer	*Checkpoint(Design and Impl)*	5%
	Token Design	5%
	Tokenizer	15%
BNF + Syntax	*Checkpoint(Design and Impl)*	5%
	Grammar Design	10%
	Parser	25%
	Syn Error Messages	5%
Semantic Analysis		
	Correct Evaluation	20%
	Last Eval Printed	5%
	Error Message / Recovery	5%
TOTAL		**100%**