CS 382 Team Project: Lexical Analyzer and Parser (100 pts)

(due on the class on 3/31/23)

This is a team project, and each team consists of two students.

1 Part 1: Lexical Analyzer (70 pts)

In the first part, you will implement a simple lexcial analyzer that can recognize the following tokens: identifier, integer literal, left parenthesis, right parenthsis, additive operator, and multiplicative operator. Specifically, we have the following grammar to define the above tokens.

```
<id>  -> <letter> { <letter> | <digit> }*
<intLit> -> <digit> { <digit> }*
<letter> -> a | b | ... | z | A | ... | Z
<digit>  -> 0 | 1 | ... | 9
<leftP>  -> (
<rightP> -> )
<addOp> -> + | -
<mulOp> -> * | /
```

The main task of this part is the implmentation of the following functions:

- lex(). The function lex() runs the state diagram to update the content of lexeme and nextToken according to the char class.
- getChar(). The function gets a char from the input file, saves it to nextChar, and decides the char class, upon each call.
- addChar(). The function adds the value of nextChar to the end of lexeme, upon each call.
- lookup(char). The function first calls addChar(), and then decide nextToken based on the parameter char using a switch statement.
- getNonBlank(). A function used to skip blank space.

2 Part 2: Parser (30 pts)

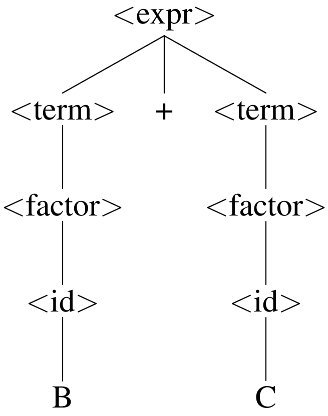
In the second part, you will implement a recusive-descent parser to parse assignment statements. Recall that a recusive-descent parser is a top-down parser consisting of a collection of subprograms.

The grammar of the parser used in the project is the following:

```
<expr>   -> <term> { ( + | - ) <term> }*
<term>   -> <factor> { ( * | / ) <factor> }*
<factor> -> <id> | <intLit> | ( <expr> )
```

Note that ¡id¿ and ¡intLit¿ are defined in the grammar of the lexical analyzer in the first part.

For the input expression B + C, as an example, the above gammar will give you the following parse tree.



The above parse tree is drawn in Latex syntree[1] package. The basic syntax used in syntree package is

```
[root
  [child1]
  [child2]
  [child3]
]
```

where each child can be recusively expanded to be a subtree.

Given an input assignment, your program is required to output its syntree description. For simplicity, you can ignore the angular bracket ⟨⟩. For example, for the input assignment B + C, your program should output the following content and save it to a data file:

```
[expr
  [term
    [factor
      [id [B]]
    ]
  ]
  [+]
  [term
    [factor
      [id [C]]
    ]
  ]
]
```

The main task in this part is the implementation of three user-defined functions: i) expr(), ii) term(), and iii) factor(). You need to revise the printf functions in each function shown in the texbook to output the right form mentioned above.

You can debug/test the code from simple inputs, for example,

[1] syntree: http://www.matijs.net/software/synttree/

1. A + B
2. A + B * 100
3. A * (B + 100)

On Submission:

- Submit the source code and output data files to blackboard.