CS 24 : Problem Solving with Computers II
Programming Assignment #1
Due: November 10, 2017 (11:59 pm)

• Objectives

This project is the first projects you will be working on this quarter. There are a number of objectives to this assignment. First, to start gaining experience in developing programs in C++. Second, in case you haven't already you are going to work in pairs to get the sense of collaborative development of software. Third, because you are allowed to use any references you find online, this assignment will help you get a sense for just how many sources for programming in C++ are available via the web.

• Project specification

In this assignment you will develop a linked data structure that stores and processes a single-variable algebraic expression. In this part you will develop functionality to encode and decode a given expression to and from a linked data structure.

• Input specification

Input string is a fully parenthesized *infix* expression with a single variable, positive integer constants, and four basic operators $(+, -, *, /)$. All these will be separated by spaces. In the following the grammar of input expression is given. Each expression could be an atomic expression, e.g., $(x + 3)$, or a combination of some single expressions. Atomic expression, i.e, atomicExp, is a sequence of an open parenthesis, a variable, an operator, a positive number, and a close parenthesis. Operators are the four common ones, variable is a single variable x, and constants are all natural numbers.

`expression` ::= `atomicExp` | (`expression op expression`)
`atomicExp` ::= (`var op const`)
`op` ::= $+ \mid - \mid * \mid /$
`var` ::= x
`const` ::= $[0{-}9]^+$

Based on the grammar, here are some acceptable input strings:

- $(x + 3)$
- $((x + 3) * (x + 2))$
- $(((x + 3) + ((x * 2)/(x - 3))) * ((x + 7) - ((x + 2) * (x/2))))$

• Data structure

The internal representation of the expression in memory is a structure of linked nodes of a class *Node* which you need to design. Every node stores an operator (+,-,*,/) and two operands: left and right. Each operand is: (1) another expression (another Node), (2) variable

x, or (3) a positive integer constant (e.g. 56).

Figure 1 shows the stored linked list for the following expression: $(((x+3)*(x-5))-(x/2))$. To construct a linked list form a given expression, you can use the Algorithm 1.

Algorithm 1 Construct linked list from an infix expression

Input: a fully parenthesized *infix* expression
Output: the corresponding linked list

```
Let cursor points to the beginning of the string
current = new node
cursor = cursor.next
while cursor is not the end of the string do
    if cursor == '(' then
        create a new node n
        if current.left is null then
            current.left = n
        else
            current.right = n
        end if
        current = n
    else if cursor is an operator op then
        current.value = op
    else if cursor == x then
        current.left = x
    else if cursor is a constant c then
        current.right = c
    else if cursor == ')' then
        current = current.parent
    end if
    cursor = cursor.next
end while
```

- **Functionality of the program**

  Your program should have the following functionalities:

1. A method **String infixString()**: returns an infix string representation of the stored Expression
2. A method **String prefixString()**: returns a prefix string representation of the stored Expression
3. A method **String postfixString()**: returns a postfix string representation of the stored Expression

The prefix representation of an expression shows the operator before the operands, for example the prefix representation of the infix "$((x+2)+(x*3))$" is "$++x\ 2*x\ 3$". The postfix

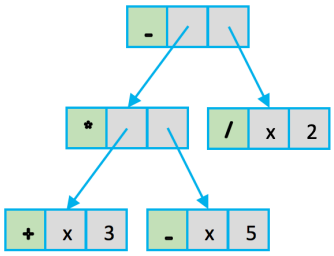

Figure 1: Linked list for a sample expression

representation shows the operator after the operands, for example the postfix representation of the infix "$((x+2)+(x*3))$" is "$x\ 2+x\ 3\ *\ +$". Note that, unlike the infix representation, the prefix and postfix representation do not require parentheses to specify an order of computation. In the output of *postfixString()* and *prefixString()*, do not use parentheses.

• **Implementation requirements**

To start gaining experience in how to structure your C++ projects, you are required to split the different aspects of this program and implement them in separate files. Then you will have to put them all back together to achieve the program functionality. By now, you should be familiar with the concept of **.h** and **.cpp** files (if not, it is a good moment to familiarize yourselves). You will need to create several **.h** and **.cpp** files for this project. We will help guiding you through this process. First you need to identify the important object-types and methods you will need for your implementation. In this project, your main object type is going to be class **Node**. Go ahead and create a file called **node.h** and declare class **Node** in it. The next step is to implement the functionality for creating the list (Algorithm 1). For this purpose, you need to create two more files - **list.h** and **list.cpp**. In **list.h** declare all the methods needed for building the list and in **list.cpp** define these methods. Then you should implement class **Expression** and implement the aforementioned methods, i.e, *infixString()*, *prefixString()*, and *postfixString()*. So, you need to create two more files - **expression.h** and **expression.cpp**. Now what's left is to put it all together by writing a main function. To do so, create **string.h** and **string.cpp**, declare your main function in **string.h** and define it in **string.cpp**. So all in all you will need eight files: **node.h**, **node.cpp**, **list.h**, **list.cpp**, **expression.h**, **expression.cpp**, **string.h**, and **string.cpp**.

• **Instructions for compilation**

Your program should compile on a CSIL machine with the following command **without any errors or warnings**.

$ g++ -o string string.cpp expression.cpp list.cpp node.cpp

• **Submission instructions**

Read carefully and strictly follow the submission instructions. Name your files as specified and follow the requested format. An automated grading will be performed and failure to follow the format for submission will result in points being taken off from your grade.

Only one copy of the project should be submitted by each group.

You need to submit 9 files. Eight of these are the implementation of your project assignment, namely **node.h**, **node.cpp**, **list.h**, **list.cpp**, **expression.h**, **expression.cpp**, **string.h**, and **string.cpp**. The ninth file must be called **README** (with upper case letters) and should contain information about the people in the group. The content of README must be of the following format:
FirstName1 LastName1, Perm#1, e-mailAddress1
FirstName2 LastName2, Perm#2, e-mailAddress2

Example:
$cat README
Tina Galli, 112233, tgalli@umail.ucsb.edu
Michael Smith, 445566, msmith@umail.ucsb.edu

- **Turn-in procedure**
  submit your project files on **Gauchospace** under the **project1 submission link**.