CS5130, Spring 2019, Dr. W. He

Project 1. Parsing Arithmetic Expressions

(Due 3/17/2019, Sun)

Description:

In this project, we will practice how to solve a problem with a special *data structure* combined with a *sophisticated algorithm*. We will solve this problem: Given a string that is a regular *arithmetic expression* containing operands, operators and parentheses, we will write a program to do the calculation following the arithmetic rules.

The method contains the following key steps:

- Convert the standard arithmetic expressions from the *infix* form to the *postfix* form.
- Evaluate the *postfix* expression.

Note:

You can use *Java, C/C++*, or *Python* to write the program.

Requirements:

1. (*Input*) When your program starts, it prompts the user to enter an arithmetic expression as a string. In order to make your program simpler, you do not need to write any special interface, just a prompt message. After your program finishes processing one expression, it stops.

2. (*Skip Data Validation*) The validation step could be very complicated. But we want to make the program focus on the main technical part. So we require the input string has the following properties:

   - The operands must be non-negative integers.
   - The operators can only be: `+, -, *, /`.
   - The delimiters can only be (and).
   - The whole input string is a valid arithmetic expression.

   Your program does not need to check for all the above rules. We just assume them for simplicity. But when you enter an expression for testing, make sure that it follows all the above rules.

3. (*Preparation*) You take your input as a `string` variable, and then go through it character by character. In this way, you do not set a limit on the number of characters in the string. When you parse the string, be aware of the case that an operand can have multiple digits, e.g. `215`.

4. (*Postfix Notation Conversion*) Assume that you have a valid *infix* expression. You need to convert it to a *postfix* expression using the following rules:

   (*Preparation*) You need to use a string variable and a stack to store the intermediate results. The string variable is used to store the content of the postfix expression. In the discussion below, I will call it *buffer*. (I want to avoid using *output* to call it, because it not our final output.)

   - When you see an operand, write it to the *buffer* (postfix).
   - When you see an open parenthesis (, push it on the stack.
   - When you see a close parenthesis), do this:
     ```
     while-loop
       pop an item,
       if item is not (, write it to the buffer
     quit-loop if item is (
     ```
   - When you see an operator, do this: (*view the details in notes*)
   - When there is no item in your input string, while the stack is not empty, pop item by item, and place it one by one in the buffer. (It is recommended to use a comma to delimit the items in the buffer string.)

5. (*Evaluate Postfix Expression*) Now you have a *postfix* expression of an arithmetic expression. You need to evaluate using the following rules:

   - You use a stack to store your operands in the expression.
   - When you see an operand from the postfix expression, push it onto the stack.
   - When you see an operator, pop the top two operands from the stack and apply the operator to them with the lower one as the first operand and the upper one as the second operand. Push the result back into the stack again.
   - When you are done, pop the stack to obtain the answer.

6. (*Output*) You just simply output the answer on the screen. When you have a division, you need to make sure that the divisor is not zero. If it is zero, print out a message and stop the program. When the divisor is not zero, you just do the integer division. In this way, you do not need to worry about the floating-point numbers.