CS 474: Object Oriented Programming Languages and Environments

Fall 2018

Second C++ project

Due time: **9:00 pm on Sunday 12/2/2018**

Instructor: Ugo Buy
TA:Guixiang Ma

Total points: 120
Full credit: 100 points (20 points extra credit)

For this project you will implement again the *Programmable Calculator (PC)* that you already designed and implemented in Smalltalk, but in C++11 this time. Recall that the PC is similar to a traditional calculator; however, it also lets users specify sequences of arithmetic operations on 4 registers, denoted by *x, y, w* and *z*. The main difference with respect to the Smalltalk version is that the C++ version does not include a GUI. Instead, your new PC will read instruction sequences from a file located in the same directory as your code executable. Also, the new PC implements a simple read-command loop with commands for (a) reading a new instruction sequence from file, (b) executing one instruction (e.g., when debugging your instruction sequence), (c) executing the entire instruction sequence from the current state, (d) printing the instruction sequence and register content, and (e) exiting the PC. Instructions appear in separate lines in the input file. Lines are numbered from 1 onward. Registers *x, y, w* and *z* are all initialized to zero. After executing an instruction the modified values of *x, y, w* and *z* are displayed again. The order of execution of expressions is sequential from the first expression (on Line 1), except when an expression with the question mark is encountered. (See below.)

The syntax of each arithmetic expression conforms to the following four patterns, where *id* can be one of *x, y, w* and *z* and *op* can be one of +, -, *, ** and /:

1. *id = id op id.*
2. *id = id op constant.*
3. *id = constant.*
4. *id ? go int.*

The first two forms above supports addition, subtraction, multiplication, division and exponentiation of two numeric operands. The operands must be of integral or floating point type (i.e., no fractions or complex numbers.) The third form stores a numeric constant in one of the registers. The fourth form allows control transfers to arbitrary lines in the expression sequence entered by the user. If value associated with the *id* appearing on the left of the questioni mark is different from zero, the expression located at the line denoted the *int* operand is executed instead of the next expression. Expression execution ends when the last expression in the sequence has been evaluated. However, to prevent infinite execution loops, you should halt expression execution also after 100 expressions have been evaluated.

(The periods at the end of each expression type should make it easier for you to parse the sequence of expressions extracted from the text editor widget.) Here is an example of a simple PC instruction sequence. At the end of this execution *x, y, w* and *z* should hold the values 4, 2, 0 and 16 respectively.

```
1:   x = 3.
2:   y = 2.
3:   w = 2.
4:   z = x ** y.
5:   w = w - 1.
6:   x = x + 1.
7:   w ? 4.
8:   x = x - 1.
```

Your command loop should handle the following commands:

1. `i <file_name>` – *Input file* – This command reads an input file containing an instruction sequence. The instruction sequence is stored in the PC and printed on the user's console. The input file does not contain line numbers; however, you should display line numbers when printing the instruction sequence on the display.

2. `d` – *Debug* – Execute one instruction and display both the content of the the 4 registers and the index position of the next instruction to be executed. This command has no effect, except for printing an error message, if no instruction sequence was entered previously with the `i` command. However, `d` can be called even after the R command has caused the 100-instruction limit to be reached.

3. `r` – *Run* – Execute the instruction sequence from the index position of the next instruction to be executed until the completion of the instruction sequence or the limit of 100 instructions is reached. Instructions that may have been previously executed in debug mode are not count toward the 100 instruction limit. This command has no effect, except for printing an error message, if no instruction sequence was entered previously with the `i` command.

4. `c` – *Continue* – This command continues the execution of the program for an additiona 100 instructions. The command should be called after the `r` command has caused the 100-instruction limit to be reached. If `c` is called in any other circumstance, it has no effect.

5. `p` – *Print* – This command prints the instruction sequence (with line numbers) and the content of the registers on the user's console (i.e., the standard output stream).

6. `q` – *Quit* – This exits the PC.

You may assume that expression sequences in the PC by PC users are syntactically correct. (You are not required to perform error diagnosis or correction). Also, you must use inheritance and at least one abstract superclass in your implementation. One good way for complying with this requirement is to define an abstract superclass for expressions with concrete subclasses for each particular type of expression. Make sure that an abstract superclass you define includes at least one concrete method, and that this method is actually executed during normal operation of the PC.

Extra credit. If you implement correctly the *Command* pattern from the Gang-of-Four book, you will receive a 20% increase on your project grade, up to 20 extra points.

You must work alone on this project. Save all your code in a collection of header and code files and submit a zip archive with a (short) readme file containing instructions on how to use your PC. The archive should also contain a sample *input.in* file. Submit the archive by clicking on the link provided with this assignment. Your code should compile under the GNU C++11 or the CLANG C++11 compiler. No late submissions will be accepted.