CS 474: Object Oriented Programming Languages and Environments
Fall 2023

Second C++ project

Due time: 11:59 pm on Wednesday November 29, 2023

This is the fourth project of the Fall 2023 semester. As with Project 2, in this project you will implement a Language System (LS) for the Weird Language (WL). However, in this project you will code the LS in C++, instead of Ruby. As before, the LS will first a read a WL program from a file called "input.wl". The file will contain one WL instruction in each line. Next, the LS will process each instruction sequentially, starting from the first line of text in the file. The syntax and semantics of WL instructions is given below. Note that the instruction set of the WL has evolved slightly with the addition of a GOTO instruction. Also, this time you *must* implement the *Command* design pattern from the pattern system of the Gang of Four. The class diagram of the pattern is given below for your convenience.

WL programs use two kinds of data structures, namely numbers and singly-linked lists. List elements can be integers or nested lists. Objects of these types are bound to variables, which don't need to be declared.

The LS will keep two storage areas, a program memory that stores a WL program loaded from the input file, and a data memory holding identifiers declared in the program and the values bound to each identifier. In addition, the WL maintains a program counter (PC) storing the line number of the instruction being currently executed. Executing a WL program consists of the following two steps:

1. Read a WL program from the input file. The program is stored in the memory starting at address (i.e., line) 0, the first line of code read from the input file.

2. Execute a command loop consisting of three commands:

    o – Execute a single line of code, starting from line 0. The PC and the data memory are updated according to the instruction. The resulting values of the data memory and the PC are printed on the console.

    a – Executes all the instructions until a halt instruction is encountered or there are no more instructions to be executed. The values of the PC and the data memory are printed after the program terminates execution.

    q – Quits the command loop.

You must implement lists as single linked lists and you must use inheritance in your implementation. One good place for inheritance is to define a superclass for WL instructions with concrete subclasses for each particular instruction. You must implement deep copying of lists yourself, without relying on Ruby constructs for marshalling and unmarshalling data structures. You may assume that WL code that you must execute is correct, that is, you need not check for syntax errors. Beware of infinite loops in WL programs.

Here is an example of a simple WL program:

```
0. VARINT x 10
1. VARLIST list1 10, 20, 30
2. VARLIST list2 40, 50, 60
3. COMBINE list1 list2
4. GET y 2 list1
5. ADD x y
6. VARLIST list3 70, 80, 90
```

VARINT *x i*	This instruction assigns an integer *i* to a variable *x*. Parameter *i* can be an integer variable or an integer literal. Variable *x* is implicitly declared the first time it is used.
VARLIST *y arg1, arg2, ...*	This instruction assigns a list to a variable *y*. Variable *y* is implicitly declared the first time it is used. The list will contain a variable number or arguments provided in a comma-separated sequence. Each argument *arg1, arg2,* etc. can be either an integer constant or a variable denoting a list or an integer.
COMBINE *list1 list2*	Argument list *list1* is concatenated with *list2*. The resulting list is stored back into *list2*.
GET *x i list*	The i-th element of *list* is assigned to variable *x*. Parameter *i* can be an integer variable or an integer literal. Variable *x* is implicitly declared the first time it is used. Assume that index *i* is within bounds.
SET *x i list*	The *i*-th element of *list* is set to *x*. *x* could be an integer constant or a variable denoting either a list or an integer. Assume that index *i* is within bounds.
COPY *list1 list2*	The content of *list2* is deep copied into *list1*. *list1* is returned; *list2* is not modified.
CHS *x*	This statement changes the sign of the integer value bound to *x*. Assume that *x* is either an int constant or int variable
ADD *x y*	This statement adds the integers bound to the two arguments and stores the result in *x*. Arguments *x* and *y* can be integer variables or integer literals.
GOTO *i*	This is the unconditional jump. Transfer control to the instruction at line *i*.
IF *x i*	This is the conditional jump If *x* is an empty list or the number zero, jump to instruction at line *i*.
HLT	Terminates program execution.

Table 1: Instructions of the Weird Language.

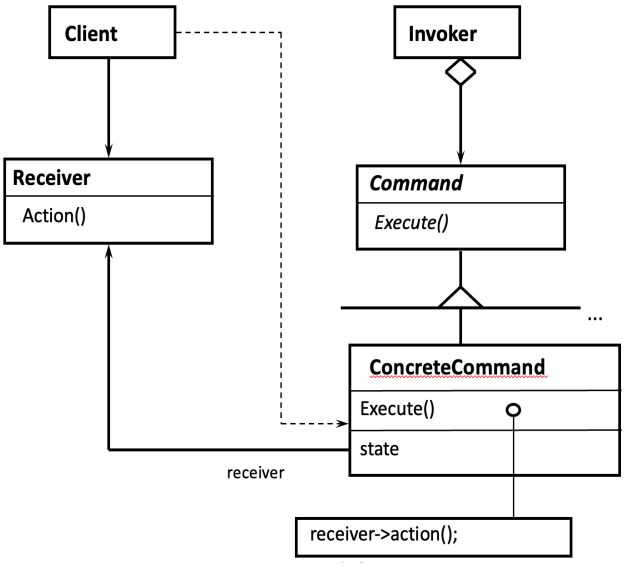
7. SET list3 1 list2
8. COPY list4 list2
9. GET list5 1 list4
10. SET 100 2 list5
11. HLT

When the WL program is executed, the result should show values 40 and 30 in *x* and *y*. Also the lists should contain the following values

1. list1 = (10, 20, 30)
2. list2 = (10, (70, 80, 90), 30, 40, 50, 60)
3. list3 = (70, 80, 90)
4. list4 = (10, (70, 80, 100), 30, 40, 50, 60)
5. list5 = (70, 80, 100)

Implementation details. You must use the *Command* design pattern from the Gang of Four system to implement your project. The class diagram of the pattern is shown below. In the diagram the abstract class *Command* matches an abstract class called *Instruction* in your code. Moreover, concrete subclass *ConcreteCommand* matches your various instruction subclasses (e.g., *AddInstruction, GotoInstruction, CombineInstruction,* etc.) Class *Receiver* matches class or classes implementing program memory, data

memory and registers in your project. Finally, class *Invoker* matches your command loop. You are free to add more data members and member functions to the ones shown in the diagram as long as you maintain the integrity of the pattern's design. All concrete instructions should expose the same API to their clients. Use both a header (.h) file and code (.cpp) file for each of your classes. Include a simple main program testing your code in your submission.



You must work alone on this project. You may ask questions on the spec and discuss the high-level design of your project with your classmates and on the Piazza blog. However, you must not discuss low-level design decision or share code with others. Do not post project code publicly on Piazza. (You may use private postings, however.) Your project code should be in a special archive called xxx_yyy.zip, where xxx and yyy denote your first and last names. Submit the file by clicking on the link provided with this assignment. No late submissions will be accepted.