CS246—Assignment 4 - Part 1(Fall 2017)

Due Date 1: Monday, November 13, 5pm
Due Date 2: Monday, November 20, 5pm

Note that this is Part 1 of Assignment 4. Part 2 of assignment 4 will be released separately. This will allow you to work on at least part of the assignment as we finalize the rest.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.

Note: You may include the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<utility>`, `<stdexcept>`, and `<vector>`.

Note: For this assignment, you are **not allowed** to use the array (i.e., `[]`) forms of `new` and `delete`. Further, the `CXXFLAGS` variable in your `Makefile` **must** include the flag `-Werror=vla`.

Note: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. For this reason, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.

Due on Due Date 1: Submit the name of your project group members to Marmoset. (`group.txt`) **Only one member of the group should submit the file. If you are working alone, submit nothing.** The format of the file `group.txt` should be

```
userid1
userid2
userid3
```

where `userid1`, `userid2`, and `userid3` are UW userids, e.g. `j25smith`.

1. In this problem, you will write a program to read and evaluate arithmetic expressions. There are four kinds of expressions:

   - lone integers
   - variables, which have a name (letters only, case-sensitive, but cannot be the words `done`, `ABS`, or `NEG`)
   - a unary operation (`NEG` or `ABS`, denoting negation and absolute value) applied to an expression

- a binary operation (`+`, `-`, `*`, or `/`) applied to two expressions

Expressions will be entered in reverse Polish notation (RPN), also known as postfix notation, in which the operator is written after its operands. The word `done` will indicate the end of the expression. For example, the input

```
12 34 7 + * NEG done
```

denotes the expression $-(12 * (34 + 7))$. Your program must read in an expression, print its value in conventional infix notation, and then initiate a command loop, recognizing the following commands:

- `set var num` sets the variable `var` to value `num`. The information about which variables have which values should be stored as part of the expression object, and not in a separate data structure (otherwise it would be difficult to write a program that manipulates more than one expression object, where variables have different values in different expressions).
- `unset var` reverts the variable `var` to the unassigned state.
- `print` prettyprints the expression. Details in the example below.
- `eval` evaluates the expression. This is only possible if all variables in the expression have values (even if the expression is `x x -`, which is known to be 0, the expression cannot be evaluated). If the expression cannot be evaluated, you must raise an exception and handle it in your main program, such that an error message is printed and the command loop resumes. Your error message must print the name of the variable that does not have a value (if more than one variable lacks a value, print one of them).

For example (output in italics):

```
1 2 + 3 x - * ABS NEG done
-|((1 + 2) * (3 - x))|
eval
x has no value.
set x 4
print
-|((1 + 2) * (3 - 4))|
eval
-3
set x 3
print
-|((1 + 2) * (3 - 3))|
eval
0
unset x
print
-|((1 + 2) * (3 - x))|
eval
x has no value.
```

(Note: the absolute symbol is vertical bars but appears slanted above since all output is shown in italics)

Numeric input shall be integers only. If any invalid input is supplied to the program, its behaviour is undefined. **Note that a single run of this program manipulates one expression only. If you want to use a different expression, you need to restart the program.**

To solve this question, you will define a base class `Expression`, and a derived class for each of the the four kinds of expressions, as outlined above. Your base class should provide virtual methods `prettyprint`, `set`, `unset`, and `evaluate` that carry out the required tasks.

To read an expression in RPN, you will need a stack. Use cin with operator `>>` to read the input one word at a time. If the word is a number, or a variable, create a corresponding expression object, and push a pointer to the object onto the stack. If the word is an operator, pop one or two items from the stack (according to whether the operator is unary or binary), convert to the corresponding object and push back onto the stack. When `done` is read, the stack will contain a pointer to a single object that encapsulates the entire expression.

Once you have read in the expression, print it out in infix notation with full parenthesization, as illustrated above. Then accept commands until EOF.

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

Note: The design that we are imposing on you for this question is an example of the Interpreter pattern (this is just FYI; you don't need to look it up, and doing so will not necessarily help you).

Due on Due Date 1: A UML diagram (in PDF format `q1UML.pdf`) for this program. There are links to UML tools on the course website. Do **not** handwrite your diagram. Your UML diagram will be graded on the basis of being well-formed, and on the degree to which it reflects a correct design.

Due on Due Date 2: The C++ code for your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4q1`.

2. This problem continues Problem 1. Suppose you wish to be able to copy an expression object. The problem is that if all you have is an `Expression` pointer, you don't know what kind of object you actually have, much less what types of objects its fields may be pointing at. Devise a way, given an `Expression` pointer, to produce an exact (deep) copy of the object it points at. Do not use any C++ language features that have not been taught in class.

   When you have figured out how to do it, implement it as part of your solution for Problem 1, and add a `copy` command to your interpreter. If the command is `copy`, you will execute the following code:

```
Expression *theCopy = ( ... create a copy of your main Expression object ...)
cout << theCopy->prettyprint() << endl;
theCopy->set("x", 1);
cout << theCopy->prettyprint() << endl;;
cout << theCopy->evaluate() << endl;
delete theCopy;
```

   The copy will contain the same variable assignments as the original expression. However, setting the variable `x` to 1 in the copy should not affect the value of `x` in the original expression.

`x` may or may not be the only unset variable in the expression. If there are other unset variables, then naturally, an exception will be raised, and you should handle it in the same way as previously.

This problem will be at least partially hand-marked for correctness. A test suite is not required. **Note that Marmoset will provide only basic correctness tests of output. If it is found during handmarking that you did not complete this problem according to our instructions, you correctness marks from Marmoset will be revoked.**

Due on Due Date 2: Your solution. Submit it as part of your Q1 solution. Your Q1 UML should not include your solution for Q2.