Assignment #5: Advanced C++

Due Date: Friday, November 19th, 2021, 5:00 pm
Online Quiz: Monday, November 22, 2021, 5:00 pm

Topics that must have been completed before starting:

1. Advanced C++: Unique and shared pointers (smart pointers)
2. Advanced C++: RAII: Resource Acquisition Is Initialization
3. Advanced C++: Exception Safety
4. Advanced C++: STL Algorithms

Learning objectives:

- Advanced C++: Smart pointers and RAII
- Advanced C++: Exception Safety
- Advanced C++: STL Algorithms

- **This assignment has only one due date. All questions are due on the same due date. You must submit the online quiz on Learn by the Quiz date.**
- 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.
- You may include the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<utility>`, `<stdexcept>`, `<vector>`, `<map>`, `<set>`, `<memory>`, and `<algorithm>`.
- You are **not allowed** to use `new` or `delete`. You must manage memory with stack variables and/or smart pointers.
- 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, `~/cs246/f21/a5`. Just remember that, for each question, you should be *in* that directory when you create your ZIP file, so that it does not contain any extra directory structure.
- You are required to submit a `Makefile` along with every code submission. Marmoset will use this `Makefile` to build your submission.

Question 1

(50%)

Some early handheld calculators, as well as several programming languages, use a system for describing arithmetic expressions called *reverse Polish notation*. Reverse Polish notation is easiest thought of as describing a mathematical expression in a sequence of steps that use a stack. For instance, you might push 5 to the stack, push 3 to the stack, then perform the "add" step, which pops the top two elements from the stack and adds them together, then pushes the result to the stack. In reverse Polish notation, this sequence of actions is written as "`5 3 +`". All steps are either pushing a number onto the stack, or popping two numbers to perform an operation, pushing the result. These sequences can be arbitrarily long, and because of the stack-like action of pushing and popping, no parentheses are needed to show precedence. To be valid, a reverse Polish notation expression must result in exactly one element on the stack.

For instance, the mathematical expression "`(5+2)/(4-1)+7`" can be rewritten as "`5 2 + 4 1 - / 7 +`". Since the `+` and `-` each pop their respective two numbers and push one, the `/` operates over the results of the `+` and `-`. Here are the steps in evaluating that expression, one by one, assuming `/` is `int` division:

Current operation	Stack after operation
5	5
2	5 2
+	7
4	7 4
1	7 4 1
-	7 3
/	2
7	2 7
+	9

You will write a reverse Polish notation calculator, which additionally reformats reverse Polish notation expressions to standard mathematical expressions, also called *infix notation*. For instance, the sequence "`5 3 +`" from above is written as "`5+3`" in infix notation, and evaluates to `8`. Your calculator must support the standard four arithmetic operators: `+`, `-`, `*`, and `/`, operating over `int`s.

Your solution does not need to handle inputting negative numbers, but negative numbers must be supported in the output. Please note that *no base code at all* is provided. You must write the entire solution from scratch.

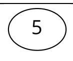
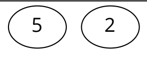
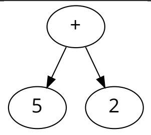
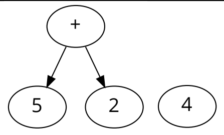
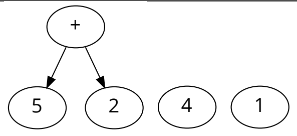
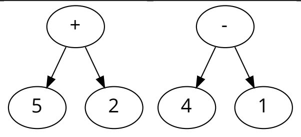
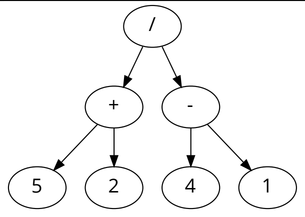
Your program will read lines in reverse Polish notation from standard input, and output lines containing both the infix expression and its solution. Correctly placing parentheses in infix expression is complex, so simply parenthesize *every* operator with its operands. For instance, if the user inputs `5 2 + 4 1 - / 7 +`, the program should output `(((5+2)/(4-1))+7) = 9`. Note that in our reverse Polish notation, all operations must be separated by spaces, but our infix notation will never use spaces.

A demonstration of an interaction with the program (lines in reverse Polish notation are input, and lines with infix notation and solution are program output):

```
5 2 + 4 1 - / 7 +
(((5+2)/(4-1))+7) = 9
5 3 +
(5+3) = 8
1 2 3 4 5 6 7 + - * / + -
(1-(2+(3/(4*(5-(6+7)))))) = -1
1 2 + 3 - 4 * 5 / 6 + 7 -
((((((1+2)-3)*4)/5)+6)-7) = -1
49 8 * 1000 * 3 100 * 5 12 * 7 + - 3 * + 8 *
((((49*8)*1000)+(((3*100)-((5*12)+7))*3))*8) = 3141592
```

To do this, you will have to create a tree from the reverse Polish notation input, then traverse the tree. **You must use the visitor pattern to traverse the tree**, as this will make it easy to write one visitor that creates the infix notation expression, and another that gets the solution. **You must use polymorphism effectively to make good use of the visitor pattern.** Note that it is possible to solve reverse Polish notation expressions with no data structures, but not to rewrite them in infix notation, so you will need to create a tree.

To build a tree from a reverse Polish notation expression, simply follow the same stack operations as described above, but pushing or popping trees instead of pushing or popping values. For an operation, rather than actually performing the operation, make a new tree with the two trees you popped as its children. Here are the steps of converting "`5 2 + 4 1 - /`" to a tree, visually:

Current operation	Stack after operation
5	
2	
+	
4	
1	
-	
/	

You will likely want a subclass of your tree class for numbers, and subclasses for each of the valid operations. To solve, the visitor of an operation should call the visitor of each child, and return the result of the operation as performed on the returns from the visitor calls to each child. To create an infix notation expression, visitors should instead return strings, and concatenate them as appropriate.

You will need to check that the input uses the stack correctly (i.e., it never attempts to use an operation with only one element on the stack, and ends with exactly one element on the stack), but exactly what to do in this case is not specified.

Due on the due date: Write this program in C++. Save your solution in a file named `a5q1.zip`. It must contain a `Makefile` that creates an executable named `a5q1` when the command `make` is given.

Question 2

(50%)

Regular expressions, such as those used by `egrep`, are checked by *finite automata* (FAs). A *finite automaton* (FA) is a directed graph in which each edge is labeled with a character. You'll see, or you have seen, a lot more about FAs, and in particular the distinction between nondeterministic and deterministic FAs, in CS241. In this problem, you'll be using a simplified form of FAs backwards, so it's not important whether you've already heard of them.

Given the description of a directed graph with characters labeling each edge (that is, an FA), in this problem, you'll be finding all of the strings of a certain length represented by that graph. That is, find every path of some length, and for each of those paths, give the string represented by the characters along that path. Note that in real FAs, there are "accepting" nodes, so not every path is actually important, but for this problem, we simply want every possible path.

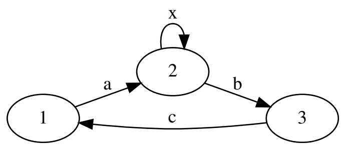
You are to implement this graph in the class `FiniteAutomaton`, defined in the header `finiteautomaton.h`, which provides `addNode` and `addEdge` methods for building the graph, and a `search` method to search for matching strings of a given length. This class also contains a nested class `Node`, representing a node in the graph, but the implementation of `Node` is entirely up to you; users of `FiniteAutomaton` should not attempt to call any methods or access any fields directly on a `Node`.

`Node *addNode()` adds a node to the `FiniteAutomaton` and returns a pointer to it. Remember that for this entire assignment, you may not use `new` or `delete`, so you are expected to allocate this `Node` using smart pointers. Nodes are opaque to users of `FiniteAutomaton`, so this method has no arguments, and the returned `Node` is only useful for passing back into the `FiniteAutomaton`'s other methods.

`void addEdge(Node *a, Node *b, char c)` adds an edge in the graph from node `a` to node `b` labeled by the character `c`. The behavior is not defined if `a` or `b` are not part of this `FiniteAutomaton` or are otherwise invalid, or if an edge from `a` to `b` labeled by `c` already exists. However, it is valid to have multiple edges between the same pair of nodes with different character labels, or for a node to link to itself, or for a node to have multiple edges labeled by the same character.

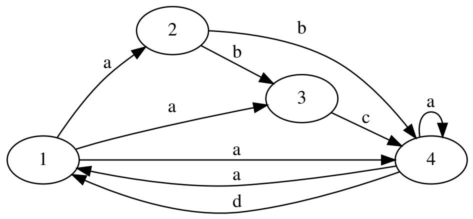
`vector<string> search(Node *start, int len)` searches for strings of length `len`. To do this, it must find all paths starting from the node `start` of length `len`, and for each, generate a string which contains the characters of the labels of the edges along that path, in the order that they appear in the path. The vector of strings must be in lexicographic order, and every string in it must be of exactly `len` characters long. Note that it is possible to generate the strings in lexicographic order in the first place, but is also acceptable to simply sort the strings after you've found them all. The vector may be empty, if there are no paths of the given length. A string may appear multiple times, if there are multiple distinct paths labeled by the same characters.

For instance, consider this finite automaton:



If we call `search(n, 5)`, where `n` points to the node labeled "1", we should get a vector that contains the strings "abcab", "abcax", "axbca", "axxbc", "axxxb", and "axxxx". These strings, in turn, represent the paths `1-2-3-1-2-3`, `1-2-3-1-2-2`, `1-2-2-3-1-2`, `1-2-2-2-3-1`, `1-2-2-2-2-3`, and `1-2-2-2-2-2`.

Another example:



If we call `search(n, 3)`, where `n` points to the node labeled "1", we should get a vector that contains the strings "aaa", "aaa", "aaa", "aaa", "aaa", "aad", "aba", "aba", "abc", "abd", "aca", "aca", "acd", "ada", "ada", and "ada". These strings, in turn, represent the paths `1-4-4-4`, `1-4-4-1`, `1-4-1-2`, `1-4-1-3`, `1-4-1-4`, `1-4-4-1`, `1-2-4-4`, `1-2-4-1`, `1-2-3-4`, `1-2-4-1`, `1-3-4-4`, `1-3-4-1`, `1-3-4-1`, `1-4-1-2`, `1-4-1-3`, and `1-4-1-4`. There's no way to determine which string came from which path, so the exact order of these paths isn't important when they generate the same string.

A sample test harness is provided in `codeForStudents` directory, in `fasearch.cc`. A compiled version is also provided, named `fasearch`, to test against. Compile `fasearch.cc` along with your own code to create your own `fasearch`, which is a simple command-line application which creates a single `FiniteAutomaton` and handles simple commands. Because of the simplicity of the command-line interface, it does not support spaces or other whitespace characters as the labels for links, but your library must support any characters in labels. The commands are:

Command	**Explanation**
`n`	Creates a new node.
`e` *from to label*	Adds a link from and to the given nodes, labeled by the given character.
`s` *from length*	Searches for strings of the given length starting from the given node, and outputs them to standard output.
`c`	Clears the graph by creating a new `FiniteAutomaton`.
`q`	Quits.

Due on the due date: Write the library in C++. Save your solution in a file named `a5q2.zip`. It must contain a `Makefile` that creates an executable named `a5q2` when the command `make` is given, and must include the `fasearch.cc` test harness, unmodified.