CSC 20: Project 4 - Building a Circuit

This semester we are going to build a simple logic simulator. The steps to using the simulator will go something like this:

~ Users of the simulator will specify a boolean logic formula using text.
~ The simulator will build an equivalent logic circuit in memory.
~ The user of the simulator will specify input values for the circuit.
~ The simulator will evaluate the circuit and print the result.

In this step of the project, you will write methods that allow you to take a logic expression as a String and build two things from it: an equivalent logic circuit and a Map that maps variable names found in the logic expression to buffers holding the variable's current value.

This sounds complicated, but actually is not a lot of code. You will write two public methods. One of them will convert infix logic expressions into postfix logic expressions using the shunting-yard algorithm. The second will convert postfix logic expressions into circuits.

Specification:

You should write a class called `CircuitBuilder`. This class is not intended to be instantiated into objects, but instead should contain two static utility functions.

```
public static String toPostfix(String infix)
public static GateOutput buildCircuit(String postfix, Map<String,GateInputSingle> variables)
```

`toPostfix` should take an infix expression as input and return the equivalent postfix expression constructed by following Dijkstra's Shunting-Yard algorithm. To simplify the problem a bit, you may assume that the input is a well-formed expression, has no errors, and will consist entirely of upper-case letters and the symbols `+`, `*`, `-`, `(`, and `)`. (The `-` symbol will be used as a unary negation operator only.) When an upper-case letter occurs, it will not be preceded or followed by another upper-case letter. For example, on input `A*-(B+C)` the method should return `ABC+-*`.

`buildCircuit` should be passed a postfix expression produced by `toPostfix` (or any other postfix expression respecting the same rules as `toPostfix`) and an allocated but empty Map. The method should build an equivalent circuit in memory and return a reference to the logic gate that produces the circuit's output. In the process of building the circuit, the method should build the map to include each variable and the buffer that holds its value (more on this later). The letters `T` and `F` should be interpreted as true and false and don't represent variables. The operator `*` should be interpreted as logical-and, `+` as logical-or, and `-` as logical-not. Not has highest priority and Or has lowest. The binary operators associate from left-to-right, the unary from right-to-left.

Robust versions of these methods should throw exceptions when inputs are not correct, but to simplify the problem a little you may assume that all inputs are correctly formed.

Note that these two methods are largely independent of each other. You should write and test one of them thoroughly before progressing to the other.

Buffer:

After building the circuit in memory, the client will want to assign values to each of the variables in the expression and read the result.

```
Map<String,GateInputSingle> vars = new HashMap<String,GateInputSingle> ();
String post = toPostfix("A*B");
GateOutput out = buildCircuit(post, vars);
GateInputSingle a = vars.get("A");
```

```
GateInputSingle b = vars.get("B");
a.input(true);
b.input(false);
System.out.println("When A=true and B=false, A*B=" + out.output());
```

How does this work? On the calls to `input` we need something to hold the value we pass in. I've been calling this a buffer. It's an extra gate that has no function other than remembering a value. So, for example, `a.input(true)` passes true into the buffer, and whenever the buffer's `output` method is called, it returns true.

The simplest way to implement this behavior is to write a `Buffer.java` class that implements `GateInputSingle` and `GateOutput` and behaves just like `Not` except that it doesn't negate its input. (Be sure to turn in `Buffer.java` along with `CircuitBuilder.java`.)

So, each time you see a new variable in your expression, you will create a `Buffer` for it and put it in the pair in the map.

Algorithms:

The Shunting-Yard algorithm is described here. Although we are using these operators with boolean values rather than numbers, the algorithm is the same (actually simpler since there are fewer operators).

Once you have a postfix expression, you follow the algorithm seen in class, but modified for circuits: If the next symbol in your expression is an upper-case letter, push a buffer onto the stack representing it. If the next symbol is a unary operator, pop the top of the stack, combine it with a gate for the unary operator, and push the result back on the stack. If the next symbol is a binary operator, pop the top two from the stack, combine them with a gate for the binary operator, and push the result back on the stack. When the postfix expression is done being processed, there should be a single gate on the stack, and it should be the one that produces the circuit result.

To Receive Credit:

Follow the directions in Project Requirements and DBInbox Submission, and submit by 11:59pm, Tuesday, April 25, 2017.

Questions?

If something is not clear, ask questions in class, on Piazza, or in office hours. Do not wait until the last minute to clear things up. Start early!

first published 5pm, April 4, 2017