CMPSCI 187: Programming With Data Structures

Lecture #12: Stacks and Expressions
David Mix Barrington
1 October 2012

Stacks and Expressions

- The Balanced Parenthesis Problem
- The `Balanced` Class and the `test` Method
- The Basic Algorithm for `test`
- Details: Boxing, Unboxing, the `indexOf` Method
- A Driver Class for `Balanced`
- More General Expression Testing
- Infix, Prefix, and Postfix Expressions

The Balanced Parenthesis Problem

- We use parentheses in mathematics, programming and in ordinary text to separate out something that is a unit. We have an “opening parenthesis” such as “(“, then the separate stuff, then the “closing parenthesis” such as “)”. Every opening parenthesis must have a matching closing one. In a text with lots of parentheses (the ultimate example is a program in a Lisp-family language like Scheme), it can be hard to tell whether every opening parenthesis is matched, and which closing one matches which opening one.

- When there is only one kind of parenthesis, it’s enough to read the text and keep track of the **nesting level** of the parentheses. Start at 0, add one for each “(“, and subtract one for each “)”. At the end you should be at 0 again. And at no point in the middle should your count be *negative*, because then you would have more openers than closers up to that point, and one of those closers must lack a matching opener. In CMPSCI 250 we’ll be able to *prove* that if a sequence meets these conditions, the parentheses do match up.

- Things get a bit more complicated if there are *more than one kind* of parens.

The `Balanced` Class and the `test` Method

- Following DJW, we're going to build a class that allows us to test whether a text is **well-formed**, which in this case means that every opening parenthesis has a matching closing parenthesis *of the same type*. Furthermore, pairs of parens may **nest** but not **interlock**. If the opening paren of a pair is inside another pair, the closing paren of the first pair must be inside the second as well.

- A `Balanced` object will have a particular set of opening parens, and a particular set of closing parens, with each closer matching exactly one opener. We establish these sets when we construct the object, and we can then call the `test` method of the object on a text.

- The test method will return its result in the form of an int: 0 if the text is well-formed, 1 if it contains a mismatch, and 2 if there is no mismatch but it ended with some openers still unmatched by closers. A mismatch can be either a closer with the wrong opener, or a closer that cannot match any opener. One could argue that DJW's 0, 1, and 2 should be replaced by named constants.

The Basic Algorithm for `test`

- If two opening parens occur consecutively in the text, we must match the later one before we match the earlier one. This **last-in-first-out** rule is exactly what a stack does.

- Here's the idea: Whenever we see an opening paren, we push it onto the stack. When we see a closing paren, we pop the top element off the stack and check it against the closing paren. If they don't match, the string has a mismatch (and so we output a 1). If they do match, we keep going.

- If we ever see a closing paren when the stack is empty, there is a mismatch.

- If the text ends and the stack is not empty, we output a 2 because there has been no mismatch yet, but some opening parens remain unmatched.

- How clear is it that this algorithm is correct? We don't yet have the mathematical tools to answer (but we'll develop them in CMPSCI 250).

Details: Boxing, Unboxing, the `indexOf` Method

- The `Balanced` object is storing one string containing the opening paren characters, and another with the closing ones. For every number i, the i'th character in the string of openers matches the i'th character in the string of closers.

- When we see a text character, we can use the `indexOf` method of the `String` class to see whether it is in either the string of openers or the string of closers. This method returns an `int`, which is -1 if the parameter character is not in the string, or its index if it is there. Rather than store characters on the stack, we can store these `int` values in a `Stack<Integer>`.

- Java allows us to use `int` and `Integer` values pretty much interchangeably. When we push an `int` onto the `Stack<Integer>` it **boxes** it, and when we assign the top value to an `int` variable it **unboxes** the `Integer`. This works for any of the **wrapper classes** corresponding to the primitive types.

A Driver Class for `Balanced`

- Once we have the class, we can use it in an application that will test texts for being well-formed. DJW give an application `BalancedApp` that constructs a Balanced object with openers `“([{“` and closers `“)]}”`, then requests expressions from the console and writes to the console whether they are well-formed.

- The strings are read by a `Scanner` attached to `System.in`, and the answers are in English rather than the 0, 1, and 2 values we used internally.

- It would be a simple matter to use a GUI instead the console for input/output.

```
Enter an expression to be evaluated:
(xx[yy]{ttt})
The symbols are balanced.

Evaluate another expression? (Y = Yes): Y
....
```

More General Expression Testing

- Testing expressions in this way is one special case of an operation called **parsing**. A **formal language** is a set of strings governed by explicit rules, which determine which strings are in the set and may assign a **meaning** to each string. A parsing algorithm determines the meaning of a given string, by working out how it was formed according to the rules.

- The rules for well-formed expressions can be written as (1) a character that is not an opener or a closer is well-formed, (2) if O is an opener and C is the matching closer, and w is a well-formed string, then "OwC" is well-formed, and (3) the concatenation of well-formed strings is well-formed.

- A similar but more complicated set of rules governs arithmetic expressions in Java. You know that an expression must obey certain rules to be valid -- every binary operator needs two operands, parentheses must be closed, etc.

Infix, Postfix, and Prefix Expressions

- Java expressions are **infix**, in that binary operators come *between* their operands. In the expression "2 + (3 * 17)", the "+" operates on the 2 and the subexpression in parentheses, and the "*" operates on the 3 and the 17.

- If there were no parentheses, "2 + 3 * 17" could also possibly be interpreted as "(2 + 3) * 17". But Java has a **hierarchy of operations** that tells us that multiplication comes before addition, unless there are parentheses that tell us otherwise.

- We could process infix expressions with a stack, but it is more complicated than working with another form of expressions called **postfix** (or **reverse Polish**). In a postfix expression the operator comes *after* the two operands, and it turns out that we never need parentheses to tell where those two operands begin and end. The postfix equivalent of "2 + (3 * 17)" is "3 17 * 2 +". We'll do more with infix and postfix expressions in Discussion #4.