EECS 293

Software Craftsmanship

2019 Spring Semester

Programming Assignment 2

Due at the beginning of your discussion session on

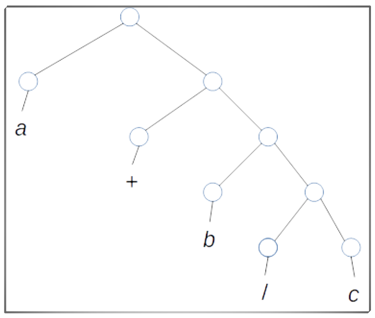
January 28-February 1, 2019

Reading

- Chapter 14 in Code Complete
- Items 48 and 50 in Effective Java
- Java's BigInteger documentation (introduction only)
- Section 19.1 in Code Complete (excluding "Forming Boolean Expression Positively", "Guidelines for Comparison to 0", "In C-derived languages …", "In C++, consider ...")
- Section 15.1 ("Plain if-then Statements" only) in Code Complete
- Sections 17.1, 19.4 in Code Complete (excluding "Simplify a nested if …")

Programming

Numerical expressions are pervasive in source code and applications. In this project, you will write a numerical expression manipulator to parse and modify expressions. The project will span programming assignments 2 through 5.



In this project, expression are represented either as a sequence or as a tree. An expression such as *a*+*b*/*c* is represented as a sequence as [*a*, +, *b*, /, *c*], and as a tree as in the figure.

A project objective is to translate a sequence representation into a compact tree representation. Programming Assignment 2 will guide you to lay the foundations needed for the rest of the project.

Package

You should organize your project in a package. The package name is up to you: it could range from the simple ('parser') to the detailed ('edu.cwru.<cwruid>.parser').

Cache

At different points of the project, constructors should avoid creating multiple immutable objects with the same non-null parameter. To support this aspect of the project, create a package-private `final class Cache<T, V>` that contains a `private Map<T, V> cache` initially empty. The cache has a method

```
V get(T key, Function<? super T, ? extends V> constructor)
```

that returns the cached object corresponding to `t` if present in the cache, and otherwise creates a new object with the provided constructor. The add method throws a `NullPointerException` with an appropriate error message if `t` or `constructor` are null.

Tokens

Tokens are the elements, such as *a* or +, that appear in the list representation of an expression. Create separate `public enum TerminalSymbol` with values `VARIABLE, PLUS, MINUS, TIMES, DIVIDE, OPEN, CLOSE`. Create a `public interface Token`. The `Token` has the definition but not the implementation of the following methods:

- `TerminalSymbol getType(), and`
- `boolean matches(TerminalSymbol type)`, which is meant to return whether the argument is equal to `getType()`.
- Then, create a `public abstract class AbstractToken` that implements
  `public final boolean matches(TerminalSymbol type)`

Variables

Variables are token such as *a*, *b*, or *c* in the list representation [*a*, +, *b*, /, *c*]. The `public final class Variable` extends `AbstractToken`. `Variable` has type `VARIABLE` (no surprise there) and implements `public TerminalSymbol getType()` correspondingly. It has a `private final String representation`, such as the string "a", which represents the variable name, and which is supposed to be non-null. The `representation` has a public final getter. `Variable` has a private constructor that sets the string representation, and

`public static final Variable build(String representation)`

that returns a variable with the given representation, or throws a `NullPointerException` with an appropriate error message if the `representation` is null. The build method avoids creating multiple variables with the same representation. In other words,

```
Variable.build("a") == Variable.build("a")
```

Define a `private static Cache<String, Variable> cache` to avoid multiple redundant variables.

`Variable` overrides `toString()` to return the variable's representation.

Connectors

Connectors are the other symbols that can appear in the list representation, such as +, -, *, /, (, and). Create a `public final class Connector` that extends `AbstractToken`. `Connector` has a private constructor that sets the type, a `public TerminalSymbol getType()`, and a `public static final Connector build` method that returns a connector of the given type, throws a `NullPointerException` if the type is null, and throws an `IllegalArgumentException` if the type is anything but one of `PLUS`, `MINUS`, `TIMES`, `DIVIDE`, `OPEN`, or `CLOSE`. All exceptions should have an appropriate error message.

All connectors of the same type are equivalent to each other. For example, a `PLUS` connector is equivalent to any other `PLUS` connector. Therefore, the build method should use a `Cache` to avoid creating multiple connectors of the same type.

`Connector`'s public `toString()` method returns the appropriate character +, -, *, /, (, or).

Nodes

Nodes appear in the tree representation. Create a `public interface Node`, with a method `List<Token> toList()`, which is supposed to return the list representation of the subtree rooted at the given node. Nodes are either *leaves*, which correspond to tokens, or *internal nodes*.

Leaves

Create a `public final class LeafNode` that implements `Node`, and which contains a `private final Token` with its getter, a private constructor that sets the `Token` value, and a build method that returns a new leaf with the given token, or throws a `NullPointerException` if the argument is null. Note that different leaves can share the same `Token`. In other words, although tokens are cached, leaf nodes are not. A `LeafNode` delegates `toString` to its `Token`. Its `public final toList` return a list containing as a single element its `Token`.

InternalNodes

Create a `public final class InternalNode` that implements `Node`. An internal node contains a `private final List<Node> children` that represent the node children, and a getter that returns a copy of the private children. The `InternalNode`'s private constructor sets the children to an unmodifiable copy of its argument, and the build method returns a new internal node with the given children, or throws a `NullPointerException` if the argument is null. Internal nodes are not cached.

The `public final toList` method return the concatenation of the children's lists. Since the list representation does not change, it should be computed from scratch only the first time that this method is invoked, and subsequent invocations should result in the return of a copy of the precomputed list. The `toString` methods invokes the `toString` method on the children, and puts the result in square parenthesis separated by commas. For example, the string representation of the tree in the figure is [a,[+,[b,[/,c]]]]. Since the string representation does not change, it should be computed from scratch only the first time that this method is invoked, and subsequent invocations should result in the return of the precomputed string.

In the next assignment, you will add to the project the conversion from list to tree representation.

Group Assignment

Your discussion leader will randomly group the section into two student teams. The team will be jointly responsible for Programming Assignments 2 through 5.

General Considerations

These classes may contain as many auxiliary private and package-private methods as you see fit, and additional package-private helper classes may be defined. However, any modification or addition to public classes and methods must be approved by the instructors at the discussion board.

You should write JUnit tests to make sure that your primary methods work as intended. However, we will revisit testing later on in the course, so extensive testing is not yet recommended. Similarly, your code should have a reasonable number of comments, but documentation is going to be the topic of a future assignment. As a general guideline at this stage of the course, comments and tests should be similar to those accepted in EECS 132. Additionally, comments should only be applied to the code sections that you feel are not self-documenting.

Canvas Resources

The module on Java Language Features contains a folder on useful Java features, such as enumerated types, regular expressions, and (under Java 8) the optional class. A discussion board can be used to ask questions and post answers on this programming assignment.

Discussion Guidelines

The class discussion will focus on:

- High-level code organization
- The design and documentation of the implementation
- Straight-line code, conditional code

Submission

Submit an electronic copy of your program to Canvas. In addition to your code, include a README file explaining how to compile and run the code. The code should be handed in a zip, tar.bz2, or tar.gz archive. Archives in 7z cannot be accepted. You can either bring a laptop to discussion to display your project on a projector, or present your project from the canvas submission.