EECS 293
Software Craftsmanship
2019 Spring Semester

Programming Assignment 4

Due at the beginning of your discussion session on

February 12-15, 2019

Reading

Read Chapter 5, Section 19.6 in Code Complete, and Items 28, 58, and 64 in Effective Java.

Grading Guidelines

Starting with this assignment, an automatic C (or less) is triggered by:

- Any routine with complexity greater than 4, or by
- Any piece of code that is essentially repeated.

Programming

First, make all the changes discussed in your discussion section. Additionally, you should refactor your code to make sure that it adopts the principles covered in the reading assignments.

Parse Tree Simplification

The first part of this assignment involves two techniques to simplify the parse tree during its construction. The first technique involves the removal of all internal nodes that have no children. The second technique involves the replacement of internal nodes with a single child with their child.

First, add the following two methods to `Node` (and their implementation to internal and leaf nodes):

- `List<Node> getChildren()`, which returns a copy of this node children (or null in the case of leaves)

- `boolean isFruitful()`, which returns true if this node is an internal node with a least a child or if this node is a leaf (in other words, the only nodes that are not fruitful are internal tree nodes with no children).

In the `InternalNode` class, create a public static nested class called `Builder`. The `Builder` has a `private List<Node> children`, initially empty, and

- `public boolean addChild(Node node)` to append a new node to the children,
- `public Builder simplify()` that returns this `Builder` after:
  - Removal all childless nodes from the children list, and
  - If at this point the children list contains only a single internal node, replace it with its children, and
- `public InternalNode build()` method that returns a new `InternalNode` with the builder's simplified children list.

Use the `Builder` in `SymbolSequence::match` to incrementally create the children list and then to simplify it just before creating the new `InternalNode`.

Run again your code against the input [*a*, +, *b*, /, *c*]. During the discussion, you can be asked to answer the question as to whether there are still redundancies in the final parse tree.

Simplified Productions

Another source of inefficiency is invisible in the parse tree, but it is clear in the parsing process: internal nodes try to match all productions against the input whereas there is only one production that applies. For example, if the remainder starts with a), an `EXPRESSION_TAIL` should skip the productions that start with + and -, and move on directly to the `EPSILON`.

To support faster production matching, replace the `productions` variable in `NonTerminalSymbol` with a

```
private static final Map<NonTerminalSymbol, Map<TerminalSymbol, SymbolSequence>> productions;
```

The main idea is that

```
productions.get(nonTerminalSymbol).get(lookAhead)
```

should return the right production to use. Here are three examples:

a. `productions.get(EXPRESSION).get(OPEN)`should return the production `TERM EXPRESSION_TAIL`,

b. `productions.get(EXPRESSION_TAIL).get(PLUS)`should return the production `+ TERM EXPRESSION_TAIL` and
c. `productions.get(EXPRESSION_TAIL).get(MINUS)`should return the production `- TERM EXPRESSION_TAIL`.

Furthermore, a null look-ahead symbol represents the scenario when there are no more tokens in the input. For example,

```
productions.get(EXPRESSION_TAIL).get(null)
```

should return `EPSILON`. All `NonTerminalSymbols` should have entries in the map, even though they may have only one appropriate `SymbolSequence`. If

```
productions.get(nonTerminalSymbol).get(lookAhead)
```

has no production, a `FAILURE` should be triggered.

It is your job to derive and implement the right look-ahead symbols and to initialize accordingly the `productions`. In particular, you will need to derive the look-ahead symbols that can preface each non-terminal symbol. Furthermore, you should make sure that productions are non-overlapping, that is there should be only one `SymbolSequence` associated with a non-terminal and a look-ahead token.

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.

Submission

Bring a copy to discussion to display on a projector. Additionally, 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.