EECS 293
Software Craftsmanship
2019 Spring Semester

Programming Assignment 3

Due at the beginning of your discussion session on

February 5-8, 2019

Reading

Read Chapters 16, 19.5, and 19.6 in Code Complete, and Item 67 in Effective Java.

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. In particular, you should modify your code to avoid methods with high McCabe's complexity.

Parse State

The conversion from list to trees is an involved process, and the main topic of this assignment. The first component of parsing is the current state of the parsing process, which is captured by a package-private `final class ParseState` with a private constructor. The `ParseState` consists of:

- `private final boolean success` that denotes whether the parsing process was successful, with a getter,
- `private final Node node`, which is the result of parsing an initial prefix of the list, with a getter, and
- `private final List<Token> remainder`, which is the part of the initial token list that was left over after parsing the `Node`. The getter returns a copy of the private variable, and

the `final boolean hasNoRemainder` returns whether there is no remainder.

The `ParseState` has a `final static ParseState FAILURE` with `success` equal to false, as well as `node` and `remainder` equal to null.

The build method returns a non-failure parse state and specifically it returns a new `ParseState` with the given `node` and a copy of the given remainder list, or throws a `NullPointerException` with an appropriate error message if any of the arguments is null.

Symbols

The conversion of trees into lists is relatively straightforward. However, the reverse operation is quite intricate. *Symbols* are an essential building block for parsing lists into trees. The intuition is that a symbol can be viewed as an input parser that builds a tree node.

Create a package-private `interface Symbol`, with the single method `ParseState parse(List<Token> input)`. The main purpose of this method is to parse the `input` into a `node`, possibly leaving a `remainder`. The `ParseState`'s `success` will be true if the parsing process was successful and false otherwise.

Terminals

In accordance with the notion that a symbol is a builder of a tree node, a *terminal symbol* is a builder of a tree leaf. Make `TerminalSymbol` into an implementation of `Symbol`. Terminals parse a token list as follows. If the first input token matches the terminal type, then the parsing is successful, the node is a leaf containing the first matched token, and the remainder is the rest of the input list. In all other cases, the parsing process returns the failure. A terminal symbol is related to a token, yet differs from it: a token is an element of an input list, whereas a terminal symbol is a builder of leaves from tokens.

Symbol Sequences

Just as an individual symbol is a builder of one tree node, a *symbol sequence* can be regarded as a builder of multiple tree nodes. In particular, symbols sequences are used to build the children of an internal node. Create a package-private `final class SymbolSequence` with a `private final List<Symbol>`

`production`. The private constructor sets the production value, and the build method returns a new `SymbolSequence` with the given production, or throws a `NullPointerException` with an appropriate error message if the argument is null. For convenience, define a second build method that takes a variable number of arguments:

```
static final SymbolSequence build(Symbol… symbols)
```

Symbol sequences should not be cached. The

```
static final SymbolSequence EPSILON
```

is a symbol sequence with an empty production.

A `SymbolSequence` delegates the `toString` method to its production.

The core of `SymbolSequence` is the method

```
ParseState match(List<Token> input)
```

which returns a successful `ParseState` if all the symbols in the production can be matched with the input, and `FAILURE` otherwise. The pseudo-code of match is:

Algorithm match(input):
remainder ← input
children ← new empty list of nodes
for each symbol in the production **do**
    parse the remainder according to the symbol
    if the parsing process is unsuccessful **then**
        return failure
    otherwise
        add the parsed node to the children list
        remainder ← remainder after parsing
return a node with the given children and the final remainder

Moreover, the match method throws a `NullPointerException` with an appropriate error message if the input is null.

Non-Terminals

Just as a symbol can be viewed as a builder of one tree node, *non-terminal symbols* are builders of internal tree nodes. In other words, the non-terminal symbols are used to create the internal nodes

during parsing. Create an `enum NonTerminalSymbol` that implements `Symbol` and that has values `EXPRESSION`, `EXPRESSION_TAIL`, `TERM`, `TERM_TAIL`, `UNARY`, and `FACTOR`.

For non-terminals, the core of the parsing process is their productions: each non-terminal is associated to a list of symbol sequences according to the following table:

Non-Terminal	**Symbol sequences**
`EXPRESSION`	`TERM EXPRESSION_TAIL`

`EXPRESSION_TAIL`	`+ TERM EXPRESSION_TAIL`
	`- TERM EXPRESSION_TAIL`
	`EPSILON`

`TERM`	`UNARY TERM_TAIL`

`TERM_TAIL`	`* UNARY TERM_TAIL`
	`/ UNARY TERM_TAIL`
	`EPSILON`

`UNARY`	`- FACTOR` `FACTOR`
`FACTOR`	`( EXPRESSION )` `VARIABLE`

The productions are represented within the `enum NonTerminalSymbol`. It is your job to find a way to represent the productions within the enumerated type. You will find that an additional problem is the circularity of the symbol sequences: for example, `EXPRESSION` depends ultimately on `FACTOR`, which then depends on `EXPRESSION`. As a hint, you can use a table-driven approach similar to the one discussed in the lecture.

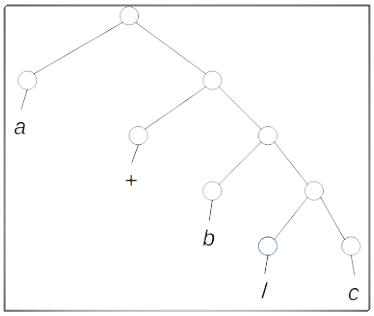
A non-terminal parses its input by going through its productions in the order given by the table and attempting to match them to the input. It returns the first successful parse state, or a failure if no production matches. Moreover, it throws a `NullPointerException` with an appropriate error message if the input is null.

Additionally, define for convenience a

```
static final Optional<Node> parseInput(List<Token> input)
```

which attempts to parse the input with an `EXPRESSION`, and returns the root node if the parsing process is successful and has not remainder, and an empty `Optional` otherwise. It also throws a `NullPointerException` with an appropriate error message if the input is null.

Run `parseInput` on the example in Programming Assignment 2: [*a*, +, *b*, /, *c*]. If all went well, you should be able to use the debugger to inspect the resulting tree rooted at an expression. However, you should also notice that the resulting parse tree is more complicated than the one in Programming Assignment 2 and shown on the side. The next two assignments will ask you to simplify the final parse tree.



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.

Discussion Guidelines

Although the discussion can range through the whole reading assignment, the emphasis will be on:

- Functions that exceed McCabe's complexity of 4 (if any)
- Non-structured programming constructs and break statements (if any)

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.

Grading Guidelines

Advance Warning: starting with Programming Assignment 4, an automatic C (or less) is triggered by any routine with complexity greater than 4 or by any substantially repeated piece of code.