Overview

In this problem set you will draw on all of the ideas you have learned about grammars, semantics, parsing, and lexical analysis to build a *recursive descent* parser for the provided LL(1) language. The main goals of this assignment are to:

- construct a lexical analyzer that will tokenize a file or string;
- construct a recursive descent parser that will parse the stream of tokens according to the rules of the grammar, the result of which should be an *abstract syntax tree*[1] (which we can later evaluate in some way).

To help you along your way, I have provided some starter code for the project. Please do *not* modify the package layout of this starter code. This starter code contains the front end, a bare-bones lexer and parser, and the basic support for an abstract syntax tree. JavaDoc documentation for the start code is available at `http://cs.merrimack.edu/~kisselz/csc3120-let-lang/`

The Language Specification

Consider the following grammar (EBNF) for a tiny language we will call the *Let Language* named after the concept of let expressions borrowed from functional languages:

Grammar 1: Let Language

<expr> → **let** <id> **:=** <expr> **in** <expr> | <term> {(**+** | **-**) <term>}
<term> → <factor> { (***** | **/**) <factor> }
<factor> → <id> | <int> | <real> | **(** <expr> **)**

Note: I have bolded tokens that are part of the language to offset them from the EBNF syntax.

The values of <id>, <int>, and <real> follow the lexical specification below:

[1] These are related to parse trees but, are more geared towards performing computation therefore the nodes of the tree are not in 1-1 correspondence with the non-terminals of the grammar.

Token	Regular Expression
<id>	[a-zA-z][a-zA-z_0-9]*
<int>	[0-9]+
<real>	[0-9]*[.][0-9]+ | [0-9]+[.]

The regular expressions are written using Java's regular expression syntax[2]

We express the semantics of our language using *denotational semantics*. The mapping function's semantic domain is $\mathbb{R}$. We denote by $e[x]$ the value of variable x in environment e. We further denote by $e[x \leftarrow v]$ variable x holds value v in environment e. The function $\mathsf{val}\,(v)$ returns the numeric value associated with numeric non-terminal v. We simplify our non-terminals in the syntactic domain to R for <real>, N for <int>, I for <id>, F for <factor>, T for <term>, and E for <expr>. We use subscripts with the variables to aid in disambiguation.

The complete denotational semantics are:

$$\llbracket R \rrbracket_e \triangleq \mathsf{val}\,(R)$$
$$\llbracket N \rrbracket_e \triangleq \mathsf{val}\,(N)$$
$$\llbracket I \rrbracket_e \triangleq e[I]$$
$$\llbracket (E) \rrbracket_e \triangleq \llbracket E \rrbracket_e$$
$$\llbracket F_1 * F_2 \rrbracket_e \triangleq \llbracket F_1 \rrbracket_e \times \llbracket F_2 \rrbracket_e$$
$$\llbracket F_1 / F_2 \rrbracket_e \triangleq \llbracket F_1 \rrbracket_e \div \llbracket F_2 \rrbracket_e$$
$$\llbracket T_1 + T_2 \rrbracket_e \triangleq \llbracket T_1 \rrbracket_e + \llbracket T_2 \rrbracket_e$$
$$\llbracket T_1 - T_2 \rrbracket_e \triangleq \llbracket T_1 \rrbracket_e - \llbracket T_2 \rrbracket_e$$
$$\llbracket \textbf{let } I \textbf{ := } E_1 \textbf{ in } E_2 \rrbracket_e \triangleq \llbracket E_2 \rrbracket_{e[I \leftarrow \llbracket E_1 \rrbracket]}$$

Phase I: The Lexer

In this phase of the project you should update `Lexer.java` and `TokenType.java` so the lexer correctly classifies real tokens, let and in reserved words, and the assignment operator (`:=`). For the assignment operator you will want to modify the `lookup` method of `Lexer.java`. For all of the other new tokens you will need to modify the `nextToken` method of `Lexer.java`.

Any new token types you create must be created in `TokenType.java`. For all tokens you add to `TokenType.java` please be sure to update the `toString` method of the `Token` class. If the value of token is important (e.g. a numeric) you should display the value of the token as well.

[2] https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/regex/Pattern.html

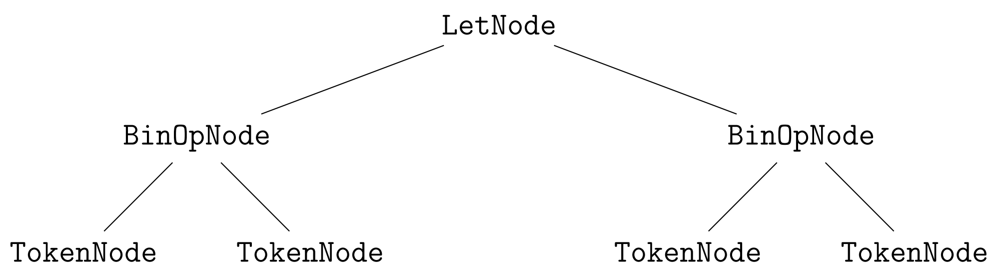
As a note, you may want to add a `private` helper method to `Lexer.java` to help with keywords.

Phase II: Abstract Syntax Tree Nodes

An abstract syntax tree (AST) is related to a parse tree but, instead of placing non-terminals in the tree, we place a node that represents the computation in the tree. For this assignment you will want to create several types of nodes, each `extends` `SyntaxNode` and live in the package `ast.nodes`. Each of these node types must implement their own constructor (which should have a parameter for each attribute), have the appropriate attributes that hold references to other nodes in the tree and, define the `evaluate` method. The node types, and their attributes, you must create are as follows:

- `TokenNode` a node that represents a token from the language (e.g, a real number). The attribute is simply a `Token`.
- `BinOpNode` that represents a binary operation in the language (i.e., addition, subtraction, multiplication, or division). The attributes you need are a `SyntaxNode` to represent the left and right operand and a `TokenNode` to represent the operation.
- `LetNode` that represents a let expression. The attributes are a `Token` that holds the variable (this is *not* a reference to another node), a `SyntaxNode` to represent the expression the variable is bound to, and a `SyntaxNode` to represent the expression that uses the value stored in the variable.

By way of an example, the AST for the expression `let x := 3 * 2 in 5 + x` is:



The `evaluate` method of each node type should determine the value of node. This may involve calling the evaluate method on the children nodes. Note that doing so creates a depth first traversal of the AST. For our purposes the `evaluate` method will simply return a `String` that describes the node and its attributes. For example, the result of evaluating the whole syntax tree for the expression

```
let x := 3 * 2 in 5 + x
```

should be:
`LetNode(ID(x), BinOpNode(TokenNode(INT(3)), MULT, TokenNode(INT(2))),`
`BinOpNode(TokenNode(INT(5)), ADD, TokenNode(ID(x))))`

> **!** The above output has been word wrapped, I do *not* require you to word-wrap your output.

Phase III: The Parser

The parser is the part of the project responsible for converting the stream of `Token`s to an AST. For this part of the project you will use *recursive descent* parsing to construct the AST. Recall that in recursive descent parsing every non-terminal has an associated `private` helper method in the `Parser` class. Since, we wish to use this process to build an AST, the methods associated with the non-terminals must return a `SytnaxNode` (i.e, every node should have `SyntaxNode` as a return type). These `SyntaxNode`s should be of the appropriate derived class type. The `parse` method will be responsible for calling the `private` method that handles the *start* non-terminal. Provided the EOF token has be found, the resulting `SyntaxNode` returned from the method handling the start non-terminal should be used as the *root* of a `SyntaxTree` which is returned from the `parse` method. If the EOF is not found, an error should be logged (i.e. call `logError`) informing the user of the bad syntax (see test cases).

Though not required, you should feel free to add any `private` methods, beyond those listed above, that you may feel you need.

Testing

A few programs written in the let language have been provided for you along with the expected output. You should be sure your program passes these test cases. You should also be sure to test against your own test cases.

General Reminders

- Please don't change the package organization I have provided you.
- You should thoroughly test your code with various programs. If you follow my recommended schedule, a day has been reserved for testing.
- Make sure to check your code against the rubric *before* submission.

- You are encouraged to see me for help with the assignment.
- Don't forget about the JavaDocs when you need to recall a method or class (`https://docs.oracle.com/en/java/javase/16/docs/api/index.html`).

Submission

Submissions of this problem should be digital *only*. Since the project has multiple files that need to be graded, I ask that you submit a zip file containing the project folder associated with the assignment. In Netbeans it is easy to construct a zip file for a project; go to File → Export Project → To Zip and select the project you want to zip and we the zip file should be placed. You do not have to use Netbeans but, if you do not there must be a build script included with your submission. The zip file **must be** named in such a way that your last name is present. Failure to follow these directions will result in a deduction of up to **5 points from your grade**. The directions above are for the purpose of ensuring a timely and correct grading of your project.

Grading

Your grade on this assignment will be determined as follows:

- Code was written using good design principles and style (10 points).
- The code produces the correct output in testing (10 points).
- The lexer works correctly and supports all requested features (30 points).
- The `BinOpNode` is correctly implemented (10 points).
- The `LetNode` is correctly implemented (10 points).
- The `TokenNode` is correctly implemented (5 points).
- The parser is correctly implemented per the guidelines (25 points).