Photo by Marvin Meyer on Unsplash

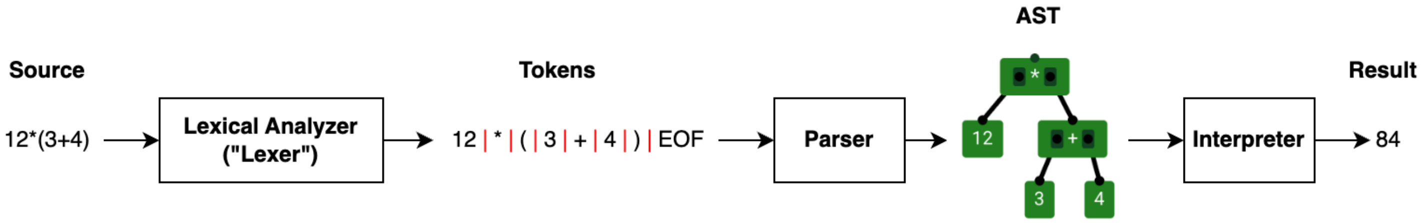
Lab 11

Lexical Analyzer

A. Lexical Analyzer

This lab continues a sequence of labs that builds a “toolchain” for a very simple programming language that supports integer arithmetic expressions (with variables) which we started in lab 6.

Here is how we go from the source code to executing our program:



In Lab 6 we developed some of the classes that represent the different nodes of the Abstract Syntax Tree (AST) of our language (the `Node`, `Lit`, `Add`, `Sub`, `Mul`, `Div` and `Neg` classes) using subtyping.
In this lab, we will develop the first stage: the **lexical analyzer**. It takes in the source code as a string and produces a sequence of tokens. A token is a consecutive sequence of characters. The process of "tokenization", of breaking a string into a sequence of tokens, is called "lexical analysis".

Note that the lexical analyzer only produces tokens, in a rather superficial way. It does not understand their meaning. For example, in the source code `size+2` it does not know whether `+` stands for integer addition or string concatenation, and it does not know whether `size` is the name of a method, a function, a variable, a type, ...).

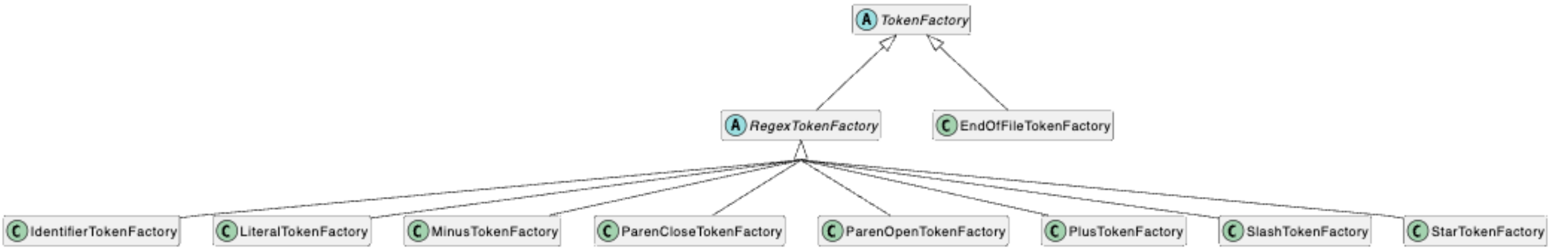
Warning: until you complete task A3, the starting code **will not compile** in its entirety. **That is totally okay**. Implement the various parts in the order they are described here, and “test” them piecewise using the provided snippets for JShell.

Task A1

Class:	`lab.lexer.Token`
Task:	**Define** the *record* class `Token`. We want to record three pieces of information about a token: • What is the `type` of the token (one of the instances of `TokenType`). • What is the `text` of the token (a `String`) • At which position in the source code the token starts (an `int`) **Implement** a method `int length()` that returns the length of the text of the token.
Run in JShell:	`new Token(TokenType.LITERAL, "12", 0)`
Output:	`==> Token[type=LITERAL, text=12, startPosition=0]`
Run in JShell:	`new Token(TokenType.EOF, "", 8)`
Output:	`==> Token[type=EOF, text=, startPosition=8]`

We will not keep creating these tokens manually. We can use an object-oriented design pattern, *Factory*, and create classes whose responsibility is to create new `Token` instances.
We will have to detect these tokens in the source code. How can we do that?
One token is especially easy to detect: at the end of the source code, we implicity assume the presence of an `<EOF>` (End Of File) token, which signals the end of the source and doesn't have any associated text.
Other tokens are just slightly more complex: for example, it's not too hard to detect a `*` by comparing each character in the source string.
Finally, other tokens are harder to detect: they span multiple characters and possibly have non-trivial variations. This "recognition task" is commonly solved in programming using **regular expressions** (often abbreviated as **RegEx**).

Here is a class hierarchy for modeling our various factories of tokens.



Each concrete class is responsible for producing tokens of one of our nine token types. The key functionality of a factory is being able to recognize a token at a certain position in the source code and, in case of success, produce the corresponding token object.

This is indeed the contract promised by the abstract `TokenFactory` at the top of our hierarchy:

```
Option<Token> matchStartingFrom(int position);
```

Task A2

Class:	`lab.lexer.EndOfFileTokenFactory`
Task:	**Implement** `matchStartingFrom` in `EndOfFileTokenFactory`. This factory produces a token (an EOF token) iff the position is at the end of the source string. The return type of the method makes it clear that we cannot always produce and return a token. Return `Options.none()` when the token cannot be produced, and `Options.some(...)` otherwise.
Run in JShell:	`new EndOfFileTokenFactory("12").matchStartingFrom(0)`
Output:	`==> None[]`
Run in JShell:	`new EndOfFileTokenFactory("12").matchStartingFrom(2)`
Output:	`==> Some[value=Token[type=EOF, text=, startPosition=2]]`

Task A3

<table>
<tr><td>Class:</td><td><code>lab.lexer.*TokenFactory</code></td></tr>
<tr><td>Task:</td><td><b>Open and carefully peruse</b> the abstract class <code>RegexTokenFactory</code>. The method <code>matchStartingFrom</code> is already implemented. It uses a <code>Matcher</code>, part of the standard Java library, to check whether the source code at a given position matches against a specific regular expression (the “rule”).

<b>Implement</b> the constructor of the eight concrete subclasses, calling the constructor of the parent class (<code>RegexTokenFactory</code>) using <code>super</code> with the right token type and the suitable regular expression.

Here are the regular expressions you need for all the factories. (Note: some of them look more complex than they need to be. That’s because they contain escape sequences.)

<table>
<tr><th>Class</th><th>Regex</th></tr>
<tr><td><code>IdentifierTokenFactory</code></td><td><code>"[A-Za-z_]\\w*"</code></td></tr>
<tr><td><code>LiteralTokenFactory</code></td><td><code>"([1-9]\\d*)|0"</code></td></tr>
<tr><td><code>MinusTokenFactory</code></td><td><code>"-"</code></td></tr>
<tr><td><code>ParenCloseTokenFactory</code></td><td><code>"\\)"</code></td></tr>
<tr><td><code>ParenOpenTokenFactory</code></td><td><code>"\\("</code></td></tr>
<tr><td><code>PlusTokenFactory</code></td><td><code>"\\+"</code></td></tr>
<tr><td><code>SlashTokenFactory</code></td><td><code>"/"</code></td></tr>
<tr><td><code>StarTokenFactory</code></td><td><code>"\*"</code></td></tr>
</table>
</td></tr>
<tr><td>Run in JShell:</td><td><code>new LiteralTokenFactory("(x+y)*456").matchStartingFrom(6)</code></td></tr>
<tr><td>Output:</td><td><code>==> Some[value=Token[type=LITERAL, text=456, startPosition=6]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new IdentifierTokenFactory("(x+y)*456").matchStartingFrom(1)</code></td></tr>
<tr><td>Output:</td><td><code>==> Some[value=Token[type=IDENTIFIER, text=x,</code>
<code>   startPosition=1]]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new IdentifierTokenFactory("(x+y)*456").matchStartingFrom(0)</code></td></tr>
<tr><td>Output:</td><td><code>==> None[]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new ParenOpenTokenFactory("(x+y)*456").matchStartingFrom(0)</code></td></tr>
<tr><td>Output:</td><td><code>==> Some[value=Token[type=PAREN_OPEN, text=(,</code>
<code>   startPosition=0]]</code></td></tr>
</table>

Task A4

Class:	`lab.lexer.LexicalAnalyzer`
Task:	We can now turn our attention to the main class, `LexicalAnalyzer`. Here is the basic idea: the lexer maintains a sequence containing **all** the factories for the various kinds of tokens. The lexer “moves” over the source, starting at the beginning. It does this by keeping a *mutable* instance variable `position` that indicates the position on the source code the lexer is about to analyze. The lexer does not immediately tokenize the whole source code. It only produces *one token at a time*, when `fetchNextToken` is called. The token fetched is stored in a *mutable* `currentToken` field. Note that `currentToken` has type `Option<Token>`, which captures the fact that we might not have a token (that’s the case at the very beginning, before fetching even the first token). How does the lexer decide which token to produce? It knows all the various factories (field `factories`). It asks all of them to try to produce a `Token` starting from the current position. Most of the factories will not produce any token. If the source is valid, at least one factory will produce a token. When there are multiple tokens that can be produced, the lexer prefers the longest token (a principle known as “maximal munch” or “longest match”). **Implement** the helper method `findLongestToken` that returns the longest token among a sequence of (optional) tokens. Note that even a token of length 0 (e.g., the EOF token) is considered longer than no token at all. **Use** `Options.fold` (which can be nested) to deal with `Option`.

Run in JShell:	`Token t1 = new Token(TokenType.LITERAL, "12", 0);`
	`Token t2 = new Token(TokenType.LITERAL, "123", 0);`
	`new LexicalAnalyzer("123").findLongestToken(of(some(t1), some(t2)))`

Output:	`==> Some[value=Token[type=LITERAL, text=123,`
	`startPosition=0]]`

Run in JShell:	`new LexicalAnalyzer("123").findLongestToken(of(none()))`
Output:	`None[]`
Run in JShell:	`new LexicalAnalyzer("123").findLongestToken(of(none(), some(t1)))`
Output:	`==> Some[value=Token[type=LITERAL, text=12, startPosition=0]]`

Run in JShell:	`Token eof = new Token(TokenType.EOF, "", 0)`
	`new LexicalAnalyzer("123").findLongestToken(of(none(), some(eof)))`

Output:	`==> Some[value=Token[type=EOF, text=, startPosition=0]]`

Task A5

Class:	`lab.lexer.LexicalAnalyzer`
Task:	**Implement** `findToken`, the core of the lexer. It needs to do the following: • Ask each factory to try to produce a token starting from the current. Use `TokenFactory.matchStartingFrom` and `Sequences.map`. • Find the longest token using the `findLongestToken` method you just implemented. • “Advance” / “Move” the lexer (updating the `currentPosition` field) by the length of the text of the produced token. • Return the produced token.
Tests:	The lexer is now complete. All the **tests** in `LexicalAnalyzerTest` should now pass.