Photo by Marvin Meyer on Unsplash

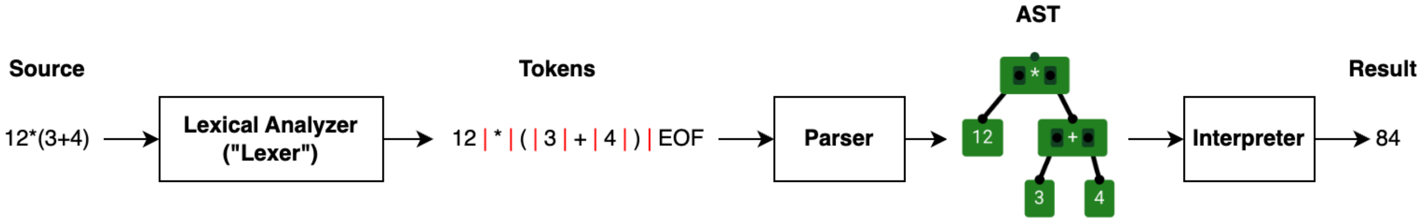
Lab 12

Recursive Descent Parser

A. Recursive Descent Parser

This lab continues a sequence of labs that builds a “toolchain” for a very simple programming language of integer arithmetic expressions.

As a reminder, 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. Every `Node` has a method `evaluate`, which acts as our **interpreter** and produces an integer result.
In Lab 11, we developed the **lexical analyzer** which can tokenize our source string.
In this lab, we will develop the missing stage: the **parser**.

A parser turns tokens into an Abstract Syntax Tree (AST). There is a lot to say about parsers, languages, and their complexity (you will take a peek of that in a course next semester). Here we will develop a **recursive descent parser**, using mutually recursive methods and loops. Therefore, this lab is a good exercise to practice recursion, loops, conditionals, and mutation. It is not going to be easy, but the ultimate result is quite rewarding!

This lab builds on top of the lexical analyzer and the AST. It's essential you do them before working on this lab. If you still have not done them, now is the right moment.

Do the following:

- **Copy** the `src/main/java/lab/nodes` folder from your **`lab-06`** to `lab-12` at the same path.
- **Copy** the `src/main/java/lab/lexer` folder from your **`lab-11`** to `lab-12` at the same path.

Warning: the code will only compile after you add your code as described above.

Another warning: we are going to develop our parser incrementally; therefore, it is important that you follow the tasks in order, as the input/output pairs only reflect what has been developed up to that point.

Before taking a stab at our big problem (parsing the full language), let's see how this would work for an even simpler language.

Programming languages are often specified in a formal way, at least to some degree. Especially their syntax (their grammar) is specified formally, often in a notation called the Backus-Naur Form (BNF) or the Extended BNF (EBNF).
A grammar consists of productions (rules) that describe how pieces of a program (e.g., expressions) are made up of smaller pieces.

Concretely, here is the grammar for a "demo language":

```
EXPRESSION ::= Literal ( "+" | "-" ) Literal
```

There is only one production for `EXPRESSION`: it says an `EXPRESSION` is made of a `Literal`, then either a `+` or a `-`, followed by a `Literal`.

The full meaning of the meta-symbols of EBNF is the following:

- *Non-terminal* symbols (written in `ALL_CAPS`) represent the names of productions.
- *Terminal* symbols (written as `Normal` names) or literals (in "double quotes") represent the lexical tokens a program is made of.
- `::=` can be read as "is defined as" (it means that the non-terminal symbol on the left is made up of the part on the right).
- Parentheses `(...)` simply group related pieces together (like in math).
- A vertical bar `|` separates alternatives.

Task A1

Class:	`lab.parser.DemoParser`
Task:	In the demo language, an expression is either an addition or a subtraction of two literals. Let us then first **implement** a method that parses a literal: `Option<Node> parseLiteral();` **Important: ALL** our `parseXXX` methods throughout this lab: • **assume** that the lexer's current token is the one that needs to be parsed. • **advance** the lexer after completing the parsing (that is: remember to call `lexer.fetchNextToken()` after parsing!). • **handle errors** without crashing by returning `Options.none()` **Call** the method `parseLiteral` you just implement from the main `parse` method (just after the first token is fetched). **Hint**: only produce a `Lit` node if there is a token and is of type `TokenType.LITERAL`.
Run in JShell:	`new DemoParser("12").parse()`
Output:	`==> Some[value=Lit[value=12]]`
Run in JShell:	`new DemoParser("abc").parse()`
Output:	`==> None[]`

Before proceeding, a hint for all the tasks.

It is not strictly necessary, but your code can probably be simplified by using the `map` and `flatMap` methods on `Option`.

When you have a function that works on the value inside Some (but cannot be applied to the None case), you can use the `map` method to avoid specifying the none case, which just shortcuts to `Options.none()`.

Example:

```
Option<String> makeBig(Option<String> str) {
  return str.fold(s -> s.toUpperCase(), Options.none());
}
```

can be simplified as:

```
Option<String> makeBig(Option<String> str) {
  return str.map(s -> s.toUpperCase());
}
```

When you have a function that works on the value inside Some and itself produces an Option, you can use the `flatMap` method to “flatten” the two nested Option.

Example:

```
public Option<Character> firstChar(Option<String> str) {
  return str.fold(s -> s.length() == 0
                     ? Options.none()
                     : Options.some(s.charAt(0)),
                 Options.none());
}
```

can be simplified as:

```
public Option<Character> firstChar(Option<String> str) {
  return str.flatMap(s -> s.length() == 0
                        ? Options.none()
                        : Options.some(s.charAt(0)));
}
```

Task A2

<table>
<tr><td>Class:</td><td><code>lab.parser.DemoParser</code></td></tr>
<tr><td>Task:</td><td>Let’s now <b>implement</b> the method
<code>Option<Node> parseExpression();</code>
that parses a simple expression in our demo language (i.e., the addition or subtraction of two literals):
<code><b>EXPRESSION</b> ::= Literal ("+" | "-") Literal</code>
Here is a sketch for the implementation:
• Call <code>parseLiteral</code> to get a node for the left operand.
• Memorize (e.g., in a local variable) the token type for the operator. We cannot immediately produce an addition or subtraction node at this point, because we still have not produced the node for the right operand. Move the lexer to the next token.
• Call <code>parseLiteral</code> to get a node for the right operand.
• Create an <code>Add</code> or a Sub node depending on the operator type.
<b>Update</b> the main parse method so that it calls <code>parseExpression</code> now, instead of just <code>parseLiteral</code>.</td></tr>
<tr><td>Run in JShell:</td><td><code>new DemoParser("1+2").parse()</code></td></tr>
<tr><td>Output:</td><td><code>==> Some[value=Add[left=Lit[value=1], right=Lit[value=2]]]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new DemoParser("1-2").parse()</code></td></tr>
<tr><td>Output:</td><td><code>==> Some[value=Sub[left=Lit[value=1], right=Lit[value=2]]]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new DemoParser("1*2").parse()</code></td></tr>
<tr><td>Output:</td><td><code>==> None[]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new DemoParser("1+").parse()</code></td></tr>
<tr><td>Output:</td><td><code>==> None[]</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new DemoParser("12").parse()</code></td></tr>
<tr><td>Output:</td><td><code>==> None[]</code></td></tr>
</table>

Task A3

Class:	`lab.parser.DemoParser`
Task:	Our parser seems to work well, but it still does not check that we do not have extra spurious tokens at the end of an expression. At the moment, `new DemoParser("1+2+").parse()` gives us back `Some[value=Add[left=Lit[value=1], right=Lit[value=2]]]` despite not being a valid expression according to the grammar. **Modify** the main parse method so that it checks that after parsing the expression the lexer has effectively reached the end of the source (i.e., the current token is EOF).
Run in JShell:	`new DemoParser("1+2").parse()`
Output:	`==> Some[value=Add[left=Lit[value=1], right=Lit[value=2]]]`
Run in JShell:	`new DemoParser("1+2+").parse()`
Output:	`==> None`
Tests:	Our parser for the demo language is now complete. All the **tests** in `DemoParserTest` should now pass.

Parser for Arith Language

Now that we have a working parser for the demo language, we can look at the full grammar for our language of arithmetic expressions.

The language has this syntax, in EBNF:

```
EXPRESSION   ::= [ "+" | "-" ] TERM { ( "+" | "-" ) TERM }
TERM         ::= FACTOR { ( "*" | "/" ) FACTOR }
FACTOR       ::= Literal
                | "(" EXPRESSION ")"
```

The production for EXPRESSION indicates that an expression is made up of an **optional** "+" **or** "-", followed by a TERM, followed by **zero or more** pieces consisting of "+" or "-" followed by a TERM.

Indeed, this grammar uses two more meta-symbols compared to the Demo one from before:

- Square brackets [...] mean optional, meaning zero or one.
- Curly braces {...} surround potentially repeated pieces, meaning zero or more.

A note on Concrete vs Abstract Syntax

There is a close connection between the symbols in a grammar and the AST node classes:

- It looks like there almost is an AST node class for each symbol.
- It looks like *non-terminal* symbols represent interior nodes (nodes with children) of the AST.
- It looks like *terminal* symbols represent leaf nodes in the AST (e.g., terminal symbol `Literal` and AST node class `Lit`).
- It looks like that the right-hand side of a production defines the children of the corresponding AST node (e.g., a `Mul` AST node, which is a kind of `Term`, has two children).

However, there is not really a 1:1 mapping between EBFN productions and AST node classes. The EBNF defines the **concrete** syntax of the language, with all the details you will see in the source code. The AST node classes define the **abstract** syntax of the language, focusing on the computationally relevant aspects.

One difference between the EBNF grammar and the AST node class hierarchy is that often you have one EBNF rule that corresponds to multiple different AST node classes.
For example, the rule for `TERM` in the EBNF is both about "*" and "/". It is about the two arithmetic operations that have higher precedence (compared to "+" and "-"). It relates not to one, but to *two* AST node classes: `Mul` and `Div`.

Another difference between the EBNF grammar and the AST node class hierarchy is that you usually will not see any information about parentheses, like what you see in 5-(2+3), in the AST node classes, but you will see information about parentheses in the EBNF.
For example, the `FACTOR` production in our grammar includes "(" and ")", but our ASTs do not retain any explicit information about parentheses. The precedence of which operation to compute first, for which one uses parentheses in the concrete grammar (and thus in the source code), is implicitly encoded in the AST through nesting. The same thing happens when drawing Expression Trees, which in fact are based on Abstract Syntax Trees.

Task A4

Class:	`lab.parser.ArithParser`

Task:	We will implement one `parseXXX` method for each production. To bootstrap our implementation, let's focus on `FACTOR` and assume that the other two productions just delegate all the work to the other methods. We will be implementing this grammar:
	`EXPRESSION ::= TERM`
	`TERM ::= EXPRESSION`
	`FACTOR ::= Literal`
	`| "(" EXPRESSION ")"`
	Implement
	• `parse` so that it calls `parseExpression` and checks that the last token is indeed an EOF, exactly like you did for the demo language
	• `parseExpression` so that it just calls `parseTerm`
	• `parseTerm` so that it just calls `parseFactor`
	Then, let us focus on `parseFactor`. **Implement** it so that it returns proper nodes for the two cases:
	• a literal, for which you need to produce a `Lit` node.
	• a parenthesized expression, for which you need to "consume" the open parenthesis, recursively call the proper method to parse an expression, and ensure that it is followed by a closed parenthesis.
	Use the already implemented helper method `currentlyAt(TokenType)` to check whether the lexer is at a specific token.
	Do not forget to advance the lexer after processing a token.

Run in JShell:	`new ArithParser("123").parse()`
Output:	`==> Some[value=Lit[value=123]]`
Run in JShell:	`new ArithParser("(1)").parse()`
Output:	`==> Some[value=Lit[value=1]]`
Run in JShell:	`new ArithParser("((123)").parse()`
Output:	`==> None[]`

Task A5

<table>
<tr><td>Class:</td><td>lab.parser.ArithParser</td></tr>
<tr><td>Task:</td><td><b>Implement</b> parseTerm according to the full grammar:

<b>TERM</b> ::= FACTOR { ("*" | "/") FACTOR }

A term can be just a factor, or it can be followed by an arbitrary number of multiplication/divisions with another factor.

We can parse a factor calling the parseFactor method we just implemented. We can then use a while loop until we keep seeing a STAR or a SLASH token.
When we encounter one of those two tokens, we can memorize whether it indicates a multiplication (as opposed to a division), move to the next token and then parse the other factor.
This other factor becomes the right operand of the new Mul or Div node we need to create.

<b>Use</b> currentlyAt(TokenType) in the condition of the loop.
<b>Keep</b> the tree you are building in this method in a mutable local variable. Initialize it with the result of parsing the first factor, and then update it inside the loop body every time you create a new node.
<b>Do not forget</b> to advance the lexer after processing a token.</td></tr>
<tr><td>Run in JShell:</td><td>new ArithParser("1*2").parse()</td></tr>
<tr><td>Output:</td><td>==> Some[value=Mul[left=Lit[value=1], right=Lit[value=2]]]</td></tr>
<tr><td>Run in JShell:</td><td>new ArithParser("1*2/3").parse()</td></tr>
<tr><td>Output:</td><td>==> Some[value=Div[left=Mul[left=Lit[value=1],
                              right=Lit[value=2]],
                     right=Lit[value=3]]]</td></tr>
<tr><td>Run in JShell:</td><td>new ArithParser("1*2*3").parse()</td></tr>
<tr><td>Output:</td><td>==> Some[value=Mul[left=Mul[left=Lit[value=1],
                              right=Lit[value=2]],
                     right=Lit[value=3]]]</td></tr>
<tr><td>Run in JShell:</td><td>new ArithParser("1*2*").parse()</td></tr>
<tr><td>Output:</td><td>==> None[]</td></tr>
</table>

Task A6

Class:	`lab.parser.ArithParser`
Task:	**Implement** `parseExpression` according to the full grammar: **`EXPRESSION`** `::= [ "+" | "-" ] TERM { ( "+" | "-" ) TERM }` If you ignore for a moment the first optional part, parsing an expression looks pretty much like parsing a term. You can follow the same guidelines of the previous task: use a loop and produce `Add` or `Sub` nodes. Once that is in place, we can think of the first part. If there is a plus token at the very beginning, that just acts as a unary plus. But `+TERM` is the same as `TERM`, so we can just move to the next token. If there is a minus token, that acts as a unary minus applied on the first term. Therefore, you can first call `parseTerm`, and then "wrap" its result in a `Neg` node.
Run in JShell:	`new ArithParser("1+2").parse()`
Output:	`==> Some[value=Add[left=Lit[value=1], right=Lit[value=2]]]`
Run in JShell:	`new ArithParser("+1+(2)").parse()`
Output:	`==> Some[value=Add[left=Lit[value=1], right=Lit[value=2]]]`
Run in JShell:	`new ArithParser("-1+2").parse()`

Output:	`==> Some[value=Add[left=Neg[expr=Lit[value=1]],`
	`right=Lit[value=2]]]`

Run in JShell:	`new ArithParser("(1/2)+10").parse()`

Output:	`==> Some[value=Add[left=Div[left=Lit[value=1],`
	`right=Lit[value=2]],`
	`right=Lit[value=10]]]`

Tests:	Our parser for the arithmetic language is now complete. All the **tests** in `ArithParserTest` should now pass.

Now that we have the entire toolchain implemented, we can write and execute arbitrarily complex arithmetic expressions!

Run in JShell:	`new ArithParser("12*(3+4)")`
	`.parse()`
	`.map(Node::evaluate)`

Output:	`==> Some[value=84]`

Run in JShell:	`new ArithParser("-1*2*3*4*5*6*7*8*9")`
	`.parse()`
	`.map(Node::evaluate)`

Output:	`==> Some[value=-362880]`