Photo by Marvin Meyer on Unsplash

Lab 6

AST • Bytecode • Tests

A. Abstract Syntax Tree

You're now going to start building a programming language called LemaLang. LemaLang is a simple programming language with few features. When building a programming language, there are many components involved. Some notable components that are relevant to this lab are the parser and the interpreter. A parser turns a string containing the source code of a program into an **Abstract Syntax Tree** (AST): a tree representation of the program (similar to the **Expression Tree**). In this exercise we will **not** build a parser, but instead we will manually build ASTs to

1. Evaluate expressions.
2. Unparse the AST back into source code (the opposite of what a parser does).
3. Analyze it to find all divisions by zero.

The goal of this first exercise is to implement a class hierarchy that consists of classes used to represent the Abstract Syntax Trees (ASTs) of LemaLang. We will model such trees by creating different classes for the different kinds of tree nodes.

We will support representing expressions consisting of (integer) literals and operations on them: addition, subtraction, multiplication, division (binary operators which apply the corresponding arithmetic operation on two integer values), and negation (the unary minus operator which flips the sign of an integer value).

Each node of this AST is represented by a class in the `lab.nodes` package that implements the `lab.nodes.Node` interface. This interface declares two methods:

- the `evaluate()` method, which returns an `int` value which corresponds to the result of the evaluation of the given node;
- the `unparse()` method, which returns a parenthesized `String` representation of the tree.

The LemaLang language represented by our AST allows to create expressions like `((6) / ((2) * ((1) + (2))))` as

```
new Div(
  new Lit(6),
  new Mul(
    new Lit(2),
    new Add(
      new Lit(1),
      new Lit(2)
    )
  )
)
```

Task A1

Class:	`lab.nodes.Lit`

Task:	Implement the `Lit` record class, which:
	• Has only one component, `value` of type `int`
	• Implements the `Node` interface
	  ○ The `evaluate()` method should simply return the value of the `value()` component of *this* record.
	  ○ The `unparse()` method should return a `String` containing the `value` within a pair of parentheses

Run in JShell:	`new Lit(1)`
Output:	`==> Lit[value=1]`
Run in JShell:	`new Lit(2).evaluate()`
Output:	`==> 2`
Run in JShell:	`new Lit(10).unparse()`
Output:	`==> "(10)"`

Task A2

Class:	`lab.nodes.Neg`

Task:	Implement the `Neg` record class, which:
	• Has only one component, `operand` of type `Node`
	• Implements the `Node` interface
	  ○ The `evaluate()` method should return the result of the *evaluation* of the `operand` with its sign inverted.
	  ○ The `unparse()` method should return a String containing the `unparse()` value of the `operand` prefixed with a negation sign, all enclosed within parentheses

Run in JShell:	`new Neg(new Lit(9))`
Output:	`==> Neg[operand=Lit[value=9]]`
Run in JShell:	`new Neg(new Lit(3)).evaluate()`
Output:	`==> -3`
Run in JShell:	`new Neg(new Neg(new Lit(12))).evaluate()`
Output:	`==> 12`
Run in JShell:	`new Neg(new Lit(4)).unparse()`
Output:	`==> "(-(4))"`

Task A3

Class:	`lab.nodes.Add`

Task:	Implement the `Add` record class, which:
	• Has 2 components, `left` and `right`, both of type `Node`
	• Implements the `Node` interface
	  ○ The `evaluate()` method should return the result of the *evaluation* of `left` summed with the result of the *evaluation* of `right`.
	  ○ The `unparse()` method should return a String representing the addition (parenthesized)

Run in JShell:	`new Add(new Lit(1), new Lit(2))`
Result:	`==> Add[left=Lit[value=1], right=Lit[value=2]]`
Run in JShell:	`new Add(new Lit(1), new Lit(2)).evaluate()`
Result:	`==> 3`
Run in JShell	`new Add(new Lit(4), new Neg(new Lit(3))).evaluate()`
Result:	`==> 1`
Run in JShell:	`new Add(new Lit(0), new Neg(new Lit(1))).unparse()`
Result:	`==> "((0) + (-(1)))"`

Task A4

Class:	`lab.nodes.Sub, lab.nodes.Mul, lab.nodes.Div`
Task:	Implement the remaining three binary operation record classes `Sub` (subtraction), `Mul` (multiplication) and `Div` (division). These are expected to work exactly like the `Add` class implemented in the previous task, except that they perform a different binary operation on the `left` and `right` values (-, * and / respectively). **Note:** each record class must: • Have 2 components, `left` and `right`, both of type `Node` • Implement the `Node` interface and its methods
Run in JShell:	`new Sub(new Lit(7), new Lit(5)).evaluate()`
Output:	`==> 2`
Run in JShell:	`new Mul(new Lit(111), new Lit(111)).evaluate()`
Output:	`==> 12321`
Run in JShell:	`new Div(new Lit(14), new Lit(5)).evaluate()`
Output:	`==> 2`
Run in JShell:	`new Div(new Lit(6), new Mul(new Lit(2),` `  new Add(new Lit(1), new Lit(2)))).unparse()`
Output:	`==> "((6) / ((2) * ((1) + (2))))"`
	Do you **feel the pain** of having to write almost the same code 4 times to produce the different operators? We will see in the coming weeks how to use the type hierarchy to reduce the code duplication.

Task A5

Class:	`lab.nodes.Node, lab.nodes.Add, lab.nodes.Sub, lab.nodes.Mul, lab.nodes.Div, lab.nodes.Lit, lab.nodes.Neg`
Task:	Add a new method to the `Node` interface: • `boolean containsDivisionByZero();` This method is used to check whether a given node contains a division by zero. Now write the appropriate implementation of this method in all the classes that implement the Node interface: • Lit • Neg • Add, Sub, Mul, Div Note that the check must be performed on the entire tree, which means that it's not enough to check whether the "current" node is a division by zero, but it's necessary to check whether any child node contains a division by zero.
Run in JShell:	`// Expr: 1 / 0` `new Div(new Lit(1), new Lit(0)).containsDivisionByZero()`
Output:	`==> true`
Run in JShell:	`// Expr: -(4 / 2)` `new Neg(new Div(new Lit(4), new Lit(2))).containsDivisionByZero()`
Output:	`==> false`

Run in JShell:	`// Expr: 2 / (1 - 1)`
	`new Div(new Lit(2),`
	`        new Sub(new Lit(1), new Lit(1))).containsDivisionByZero()`

Output:	`==> true`

Task A6

Class:	`TEST: lab.nodes.SubTest, lab.nodes.MulTest, lab.nodes.DivTest, lab.nodes.NegationTest`
Task:	Look at the `src/test/java/lab/nodes/AddTest.java` file in your project. This test class contains a 4 tests for the `lab.nodes.Add` record class: 1. `testUnparse()`: to test the implementation of the `unparse()` method. 2. `testEvaluate()`: to test the implementation of the `evaluate()` method. 3. `testContainsDivisionByZeroLeft()`: to test the implementation of the `containsDivisionByZero()` in a situation in which you expect the method to return true because there's a division by zero in the left sub–expression. 4. `testContainsDivisionByZeroRight()`: to test the implementation of the `containsDivisionByZero()` in a situation in which you expect the method to return true because there's a division by zero in the right sub–expression. 5. `testDoesNotContainDivisionByZero()`: to test the implementation of the `containsDivisionByZero()` in a situation in which you expect the method to return false (there's no division by zero in the expression) Write the test methods for the `lab.nodes.SubTest`, `lab.nodes.MulTest`, `lab.nodes.DivTest` and `lab.nodes.NegationTest` classes using the tests written in the `lab.nodes.AddTest` class as an example.
Run:	`All tests should pass`
Output:	

B. Family Tree

In this exercise you will model a family tree using subtyping polymorphism, writing the whole hierarchy from the ground up.

The family tree is modeled by having a `Person` which has two parents: a biological mother and a biological father, which may be known `Person`s or may be `Unknown`.
If the parents are known, then they themselves are `Person`s who have two parents (recursive case). Otherwise, if the parents are not known, they are `Unknown` (base case) and so are their parents.

Task B1

<table>
<tr><td>Class:</td><td><code>lab.family.FamilyTree, lab.family.Person, lab.family.Unknown</code></td></tr>
<tr><td>Task:</td><td>
1. Define an interface called <code>FamilyTree</code> in the <code>lab.family</code> package. This interface declares two methods, <code>mum()</code> and <code>dad()</code> that take no parameter and return a value of type <code>FamilyTree</code>.

2. Define a record class called <code>Unknown</code> in the <code>lab.family</code> package. It is used to represent an unknown person in a family tree. This record class has no component and implements the <code>FamilyTree</code> interface. For the implementation of both the <code>mum()</code> and <code>dad()</code> methods needed to fulfill the contract of the <code>FamilyTree</code> interface, return a new <code>Unknown</code> instance.

3. Define a record class called <code>Person</code> in the <code>lab.family</code> package. It is used to represent a known person in a family tree. This record class has 3 components: <code>name</code> (of type String), <code>mum</code> and <code>dad</code> (both of type <code>FamilyTree</code>) and <code>Person</code> itself implements the <code>FamilyTree</code> interface.
</td></tr>
<tr><td>Run in JShell:</td><td><pre>new Person("A",
           new Person("B", new Unknown(), new Unknown()),
           new Person("C", new Unknown(), new Unknown()))</pre></td></tr>
<tr><td>Output:</td><td><pre>==> Person[name=A,
           mum=Person[name=B, mum=Unknown[], dad=Unknown[]],
           dad=Person[name=C, mum=Unknown[], dad=Unknown[]]]</pre></td></tr>
</table>

Task B2

<table>
<tr><td>Class:</td><td><code>lab.family.Person</code></td></tr>
<tr><td>Task:</td><td>Implement the following instance methods in the Person class that provide access to other members of the family tree:

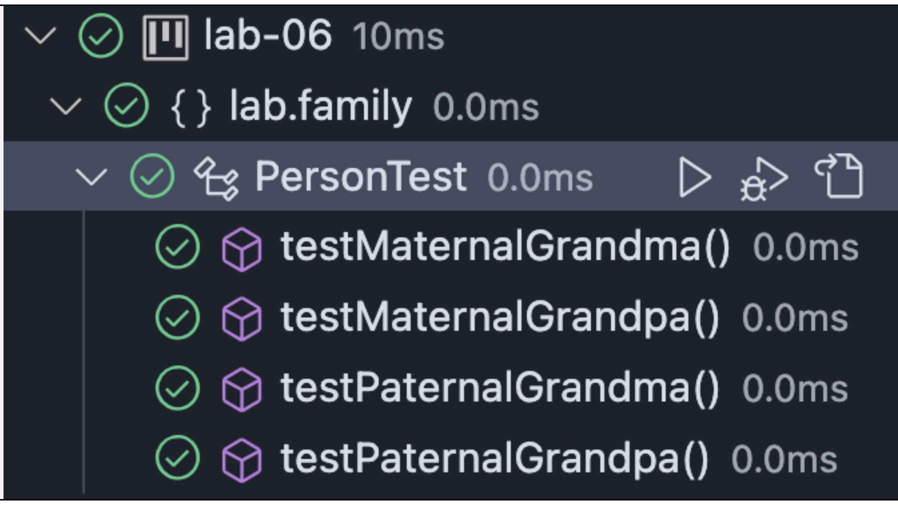
• <code>FamilyTree maternalGrandma()</code>
• <code>FamilyTree maternalGrandpa()</code>
• <code>FamilyTree paternalGrandma()</code>
• <code>FamilyTree paternalGrandpa()</code>
</td></tr>
</table>

<table>
<tr><td></td><td></td></tr>
<tr><td>Run in JShell:</td><td><pre>new Person("A",
          new Person("B",
                    new Person("C", new Unknown(), new Unknown()),
                    new Unknown()),
          new Person("D", new Unknown(), new Unknown())
).maternalGrandma()</pre></td></tr>
<tr><td>Output:</td><td><code>==> Person[name=C, mum=Unknown[], dad=Unknown[]]</code></td></tr>
</table>

Task B3

<table>
<tr><td>Class:</td><td><code>TEST: lab.family.PersonTest</code></td></tr>
<tr><td>Task:</td><td>Similarly to what you did in Task C1, write tests for the four methods you implemented in Task B2:

<code>testMaternalGrandma()</code>
<code>testMaternalGrandpa()</code>
<code>testPaternalGrandma()</code>
<code>testPaternalGrandpa()</code>

The tests should use <code>Assert.assertEquals</code> to compare two instances of Person.</td></tr>
<tr><td>Run:</td><td><code>All tests should pass</code></td></tr>
<tr><td>Output:</td><td>
</td></tr>
</table>