JLaTo Documentation

Release 0.1.0

Didier Villevalois

Sep 27, 2017

User Documentation

JLaTo is a Java Language Tools library.

CHAPTER 1

Getting Started

Parsing

Basics

You first have to create a `Parser` object:

```
final Parser parser = new Parser();
```

Then, parsing is just a matter of calling the `parse(...)` method with the appropriate reader, file or input-stream and encoding.

```
final File sourceFile = new File("<path-to-java-source-file>.java");
final CompilationUnit cu = parser.parse(sourceFile, "UTF-8");
```

Customization

A `Parser` can be instantiated with a `ParserConfiguration` object. Through this configuration object, you can enable whitespace preservation.

```
final Parser parser = new Parser(
        ParserConfiguration.Default.preserveWhitespaces(true));
```

The `Parser` class also provides a finer-grain `parse(...)` method that accepts a `ParseContext` object. For instance, you can parse an expression:

```
final String exprString = "x -> x + x";
final Expr expr = parser.parse(ParseContext.Expression, exprString);
```

Printing

Basics

You have to create a `Printer` object and call its `print(...)` method:

```
final CompilationUnit cu = // ...
PrintWriter writer = new PrintWriter(new FileWriter("out.java"));
Printer printer = new Printer();
printer.print(cu, writer);
```

The `Printer` class also propose static helper methods:

```
String formatted = Printer.printToString(tree);
```

Please refer to the JavaDoc of `Printer` for more variants of the static helper methods.

Customization

A `Printer` can be instantiated with an additional boolean flag to enable formatting of existing nodes and a `FormattingSetting` object to customize rendering:

```
// ...
Printer printer = new Printer(true, FormattingSettings.Default.withIndentation("
↪"));
printer.print(cu, writer);
```

Two base settings are provided (`FormattingSettings.Default` and `FormattingSettings.JavaParser`). Please refer to the JavaDoc of `FormattingSettings` for further customization.

Abstract Syntax Tree Manipulation

Every abstract syntax tree objects derive from the base `Tree` class and obey the same common design rules.

For each property `prop` of a `Tree` subclass is provided an accessor `prop()` and a mutator `withProp(...)` (or `isProp()` and `setProp(...)` for boolean properties). For instance, the `ImportDecl` class provide the following accessors and mutators for its properties:

```
public QualifiedName name();
public ImportDecl withName(QualifiedName name);
public boolean isStatic();
public ImportDecl setStatic(boolean isStatic);
public boolean isOnDemand();
public ImportDecl setOnDemand(boolean isOnDemand);
```

Thus you can modify an `ImportDecl` in the following way:

```
final ImportDecl decl = new ImportDecl(QualifiedName.of("org.jlato.tree"), false,␣
↪true);
Assert.assertEquals("import org.jlato.tree.*;", Printer.printToString(decl));

final ImportDecl newDecl = decl.withName(new QualifiedName(decl.name(), new Name("Tree
↪")))
```

```
                                .setOnDemand(false);
Assert.assertEquals("import org.jlato.tree.Tree;", Printer.printToString(newDecl));
```

Once you modified a tree node, you can go back to its parent tree or up to the root by calling the `parent()` or `root()` methods. Those will accordingly recreate the intermediate immutable tree nodes:

```
final CompilationUnit cu = // ...

final ImportDecl anImportDecl = cu.imports().get(0);

final ImportDecl newImportDecl = anImportDecl.withName(QualifiedName.of("com.acme.
↪MyClass"))
                                             .setStatic(false)
                                             .setOnDemand(false);

final CompilationUnit newCU = (CompilationUnit) newImportDecl.root();
Assert.assertEquals("import com.acme.MyClass;", Printer.printToString(newCU.imports().
↪get(0)));
```

Using patterns to match, filter and search

Every abstract syntax tree objects derive from the `TreeCombinators` interface which provides combinators to search, match and rewrite sub-trees. For instance, you can rewrite all names of a compilation unit and add a "42" suffix to those:

```
final CompilationUnit cu = // ...

final CompilationUnit rewrote = cu.forAll(
    new Matcher<Name>() {
        @Override
        public Substitution match(Object o, Substitution s) {
            return o instanceof Node && ((Node) o).kind() == Kind.Name ? s : null;
        }
    }, new MatchVisitor<Name>() {
        @Override
        public Name visit(Name name, Substitution s) {
            return name.withId(name.id() + "42");
        }
    });
```

In the previous example we implemented manually the implementations of `Matcher`. Fortunately, JLaTo provides quasi-quotations to do the exact same thing with Java code quote. For instance, one can rewrite all names:

```
final CompilationUnit cu = // ...

final CompilationUnit rewrote = cu.forAll(Quotes.names(),
    new MatchVisitor<Name>() {
      @Override
      public Name visit(Name name, Substitution s) {
          return name.withId(name.id() + "42");
      }
  });
```

or only all parameter names:

```
final CompilationUnit cu = // ...

final CompilationUnit rewrote = cu.forAll(
    Quotes.param("$t $n").or(Quotes.param("$t... $n")),
    new MatchVisitor<FormalParameter>() {
        @Override
        public FormalParameter visit(FormalParameter p, Substitution s) {
            return p.withId(p.id().withName(name(p.id().name().id() + "42")));
        }
    });
```

or only all public abstract classes:

```
final CompilationUnit cu = // ...

final CompilationUnit rewrote = cu.forAll(
    Quotes.typeDecl("public abstract class $t { ..$_ }"),
    new MatchVisitor<TypeDecl>() {
        @Override
        public TypeDecl visit(TypeDecl t, Substitution s) {
            ClassDecl c = (ClassDecl) t;
            return c.withName(c.name().withId(c.name().id() + "42"));
        }
    });
```

Look at the `TreeCombinators` interface for more information on the other methods to match, filter and search.

CHAPTER 2

State Classes

The so-called State classes are the bottom-up-constructed classes of the abstract syntax tree that back up the top-down-constructed Tree facade classes. They all inherit the `STreeState` interface and are wrapped inside `STree` instances which convey additional data for these nodes such as lexical information, semantic attributes, and user data.

They obey the following general contract:

General Contract

State objects contain child `STree` instances and additional primitive (or `String`) data.

In a fully operational state, State objects can't carry `null` values neither for their child trees nor for their data. Optional child trees must be wrapped into **option trees** (`STree<SNodeOptionState>`), lists of children trees must be wrapped into **list trees** (`STree<SNodeListState>`) and alternative child trees must be wrapped into **either trees** (`STree<SNodeEitherState>`).

State objects can thus be either pending initialization or fully initialized.

Pending Initialization State

When first instantiated, a State object may have null values for its non-compound child trees (those that are not options, lists or either trees). Their `equals(...)` and `hashCode()` methods may be called while pending initialization and, thus, must be null-proofed for those fields.

The compound child trees (those that are options, lists or either trees) must be fully initialized at construction time. The State constructor must then include individual nullity tests and instantiate new `STree<SNodeOptionState>`, `STree<SNodeListState>` or `STree<SNodeEitherState>` accordingly.

Fully Initialized State

The following actions will blindly query the State objects and therefore consider them to be fully initialized:

- Tree traversals
- Pattern matching and building
- Rendering with the help of the corresponding shape

The State objects must ensure non-nullity of their non-compound child trees as their first validation action in their `STreeState.validate()` implementation.