JavaCSS

March 27, 2015

Contents

1 JavaCSS

1.1 Introduction

JavaCSS is a toolset to simplify writing Java code.

Main benefits:

- Automation of the output style and conventions.
- Dependency management: import statement, pom.xml.

JavaCSS contains an ANTLR-based Java parser. It reads Java source code, and generates an Abstract Syntax Tree (AST). The parser and lexer are built by ANTLR from the Java8.g4 grammar already available in ANTLR's github repository.

Looking at the grammar itself, its main entry point is the "compilationUnit" rule:

```
compilationUnit
  :  packageDeclaration? importDeclaration* typeDeclaration* EOF
  ;
```

JavaCSS needs to parse whole Java files as well as certain incomplete Java snippets. Initially, the above rule seems to fit JavaCSS requirements nicely.

The whole process consists of:

- parsing Java code, and generating an AST
- AST processing

- serializing the final AST

JavaCSS uses StringTemplate as generator tool. However, it currently lacks a mechanism to bind or associate templates to parts of the AST. We'll refer to this feature as "template selectors".

1.2 Project setup

As with any other regular Java project, we'll start by investing some time in preparing the tool ecosystem:

- Create a new repository in github.
- Set up the folder structure expected by Maven.
- Write the initial Maven's pom.xml
- Create a new Jenkins job to listen to changes on the github repository.

1.3 Prototype

1.3.1 First test: Parsing an AST

The simplest test is simple: we want to verify the parser supports Java8 code and generates valid AST instances. Since we just use ANTLR-provided Java grammar, the purpose of this test is a simple verification of the correctness of the generated parser. We won't write many tests, since they don't help guiding us in the process of JavaCSS development.

Anyway, let's check if it is able to read the following Java code:

```
public interface Resolver extends Serializable {
public int resolve(String value); }
```

To write the test, we need to remember the API ANTLR provides for the generated parser. To build the parser instance, we provide the text to build a Java8Lexer. Then, we instantiate a CommonTokenStream with the lexer, and pass it to the Java8Parser constructor. Then, we call the method associated to the grammar rule we are interested in, and get a ParseTree instance in return. Such class represents an AST.

After adding the required imports and dependencies, the test should pass.

1.3.2 Second test: Count methods

What JavaCSS pursues is to aid in writing Java code, and one of such aids is freeing the developer from the task of managing which external classes

the code uses. That will eventually require us to deal with dependency management (which library/framework a class belongs to, and how to make sure it is available when compiling or at runtime), but for now we focus on browsing the AST to retrieve all declared types.

It's worth reviewing when such type declaration occurs in a Java source file:

- parameterized class/interface definitions
- static blocks
- instance/class attributes
- parameterized methods
- method returns
- method parameters
- local variables in methods
- local variables in lambdas

To start simple, and to allow us to get used to traversing ASTs, method returns seem a good starting point. But first we need to figure out how the AST itself looks like, how to distinguish a node from another, etc. It seems we tried to be too ambitious in our test. Let's change it: instead of retrieving the list of declared types, let's first count the methods.

The test means asking someone "how many methods are in this Java code?", but there's no one listening, yet. Even though we don't know if it'll be a wise decision, a MethodHelper class could be handy in this context:

new MethodHelper(ast).countMethods();

However, at this point we need to dig deeper into how an AST looks like. From the grammar, we can see that rule we are interested in is "methodDeclaration". But first, we need to learn more about ANTLR. In our context, we can work with ParseTree objects instead of AST nodes. They are meant to be a concrete, particularized representations. Besides that, we have three options:

1. Traverse the nodes recursively for each child, checking if the node corresponds to a method declaration.
2. Use a listener.

3. Use a visitor.

The first option is not recommended, since it adds no value and it's already implemented by ANTLR-generated classes. However, I followed it the first time, by implementing a method to check if the current node was a method (by checking the class of node.getPayload()), and calling recursively itself for each one of the children and incrementing the count.

However, ANTLR has anticipated our needs, and provides better options, and exported them as configuration settings in ANTLR's Maven plugin: add <listener>true</listener> for generating the listener API, and <visitor>true</visitor> for visitors.

For this specific test, a listener-based approach fits nicely: we don't need any parsing context besides the "methodDeclaration" rule's itself, and we don't need to tune the parsing process either.

The implementation is simple: extend Java8BaseListener to override exitMethodDeclaration(), which increments an internal counter. Then, to retrieve the number of methods, create a ParseTreeWalker instance, call its walk(listener, node) method, and retrieve the counter value inside the custom listener.

1.3.3 Third test: Retrieve the types the methods return

Now that we know how to count the methods, we can aim higher and find the return types of the methods. At this stage, it seems there's no real need to switch to a visitor approach. Eventually we'd probably rather skip processing certain nodes in the tree, which we know we are not going to deal with, but not now. Or so I thought.

The new test seems to be similar to the previous one, but we are adding some variety for the types of the methods: one iteration to build inputs with a number of methods ranging from 1 to 10, and another nested loop to provide the return types for each of the methods, choosing randomly from a list of predefined classes. Afterwards, we check whether the types found by our parser are the same as the original list.

The implementation is defined similarly to the previous use case: two overloaded methods. First, one that retrieves the AST/ParseTree after parsing the input. Second, another that takes a node and uses a listener to annotate each return type. But now, we find the first problem. Inside the exitMethodDeclarator() method, we can't retrieve the return type. We need to be in the exitMethodHeader rule. Well, in the "result" rule, but within the "methodHeader" context. And, if the return is not "void", within the

"unannType" rule, and either within "unannPrimitiveType" or "unannReferenceType". As you can see, this approach is going nowhere. What we do need is processing all terminal nodes which are descendant of the first "result" node, in all "methodHeader" contexts.

Before dealing with that problem, let's review other built-in capabilities of ANTLR. It supports XPath-like expressions, so we could try to find all terminal nodes matching "//methodHeader/result//*".

```
for (ParseTree node : XPath.findAll(tree, "//methodHeader/result//*", parser)) {
    if (node instanceof TerminalNode) {
        result.add(((TerminalNode) node).getText());
    }
}
```

It works perfectly for most cases, but if the type is a generic one, it contains one terminal node for the types and the '<', '>' and '?' symbols. Using the XPath expression "*/methodHeader/result//**!typeArguments" and calling "getText()" for any non-terminal nodes doesn't work either, since the grammar (correctly) builds different subtrees depending on the actual input and rules matched.

At this point, the only solution I see is to first ensure we are in the first occurrence of "result" within "methodHeader"; and second directly call getText() on the rule context, regardless of the subtree therein. The latter is easy, but the former is not. How can we ensure we are processing exactly the first "result" rule? ANTLR suggest to use labels in the grammar, but then we cannot use external, official grammars, verbatim.

Let's face it programatically. We know it's the first node once we're inside "methodHeader". There're no previous optional nodes to take care of. By using a walker to process the first "result", and implementing a listener for that specific rule, we are done, finally.

```
protected static class ReturnTypesOfMethodsListener
    extends Java8BaseListener {

    private final List<String> returnTypes = new ArrayList<String>();

    @Override
    public void exitResult(@NotNull final Java8Parser.ResultContext ctx)
    {
        returnTypes.add(ctx.getText());
```

```
        super.exitResult(ctx);
    }

    public List<String> getReturnTypesOfMethods() {
        return this.returnTypes;
    }
}

 public List<String> retrieveReturnTypesOfMethods(ParseTree tree, Java8Parser parser)
 {
     List<String> result = new ArrayList<>();

     for (ParseTree node : XPath.findAll(tree, "//methodHeader", parser))
     {
         ParseTreeWalker walker = new ParseTreeWalker();
         ReturnTypesOfMethodsListener listener = new ReturnTypesOfMethodsListener();
         walker.walk(listener, node.getChild(0));
         result.addAll(listener.getReturnTypesOfMethods());
    }

    return result;
}
```

1.3.4 Fourth test: adding imports to the AST

We're now one step closer towards the first requirement: automatic management of import statements. For our upcoming tests, we could use the logic we've just implemented, and perform some AST manipulations based on the return types of the methods. But that misses the point we pursue: invest the minimum time and effort before we get feedback and thus decide if the approach makes sense or not, as soon as possible.

So, in this particular context, what are we trying to do? Learn how to add specific new nodes to a ParseTree. And how can we verify it's working correctly? Well, we could generate code based on the AST and check whether the import statements are there. But again, we are nowhere near to that point. We haven't dealt with the generation phase yet. The simplest way to check in the new nodes are added correctly is to use ANTLR's XPath searches. To retrieve a ParseTree, we can parse the samples used for some of the already implemented tests.

Let's start by creating a new test ASTHelperTest, and a new test "$\text{add}_{\text{newASTnode}}()$". The first step then is to build a ParseTree instance, so let's copy our first test "$\text{can}_{\text{parseaninterfacewithextendsandasinglemethod}}()$" into a "buildAST()" helper method for the tests.

```
protected ParseTree buildAST()
    throws Exception {
    String input =
        "public interface Resolver\n"
        + "    extends Serializable {\n\n"

        + "    public int resolve(String value);\n"
        + "}\n";

    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(input));

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    Java8Parser parser = new Java8Parser(tokens);
    return parser.compilationUnit();
}
```

Similarly as we did before for retrieving the declared types for the methods, we can start with a simple helper class: "ASTHelper". Such class will add some logic in ParseTree we could use: "addImport(className)". But before that, we have to be confident we can detect whether the import nodes are added indeed. Let's add the XPath filters to the test first.

Damn it, we need the Parser instance for the XPath logic. Since Java don't allow methods returning tuples, we have two options: either split the buildAST() method in two (one for creating the parser, and the other for building the tree), or write an inner class representing a tuple. The simplest and cleanest option is the former.

```
protected Java8Parser buildParser()
    throws Exception {
    String input =
        "public interface Resolver\n"
        + "    extends Serializable {\n\n"

        + "    public int resolve(String value);\n"
```

```
        + "}\n";

    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(input));

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    return new Java8Parser(tokens);
}

protected ParseTree buildAST(Java8Parser parser)
    throws Exception {
    return parser.compilationUnit();
}
```

We only need now to verify the new import is contained in the XPath matches.

```
@Test
public void add_new_AST_node()
    throws Exception
{
    Java8Parser parser = buildParser();
    ParseTree tree = buildAST(parser);
    Assert.assertNotNull(tree);

    String myType = ASTHelperTest.class.getName();

    ASTHelper astHelper = new ASTHelper(tree);
    astHelper.addImport(myType);

    Collection<ParseTree> imports = XPath.findAll(tree, "//import", parser);
    Assert.assertNotNull(imports);
    boolean found = false;

    for (ParseTree node : imports) {
        if (node instanceof TerminalNode) {
            TerminalNode leaf = (TerminalNode) node;

            if (myType.equals(leaf.getText())) {
                found = true;
```

```
                break;
            }
        }
    }
    Assert.assertTrue(found);
}
```

Now that the test looks fine, we can proceed to defining the required skeleton and see if the test fails.

```
public class ASTHelper {
    private final ParseTree tree;

    public ASTHelper(ParseTree ast) {
        this.tree = ast;
    }

    public void addImport(final String myType) {
    }
}
```

Unfortunately, it fails with an unexpected exception:

```
java.lang.IllegalArgumentException: import at index 2 isn't a valid rule name
    at org.antlr.v4.runtime.tree.xpath.XPath.getXPathElement(XPath.java:175)
    at org.antlr.v4.runtime.tree.xpath.XPath.split(XPath.java:122)
```

Maybe we chose an invalid XPath selector. Yes, we did. The grammar rule is not "import", but "importDeclaration". Now the test fails as it should, which allows us to move forward. The idea is to implement a visitor for the rule where an "importDeclaration" occurs, and add the new subtree therein. Honestly, I didn't know how to do it, so I ended up adding a subtree which seemed good enough, but it was made up completely. It passed the test, though.

```
public void addImport(final String myType) {

    ImportAddOperation visitor = new ImportAddOperation(myType);

    visitor.visit(this.tree);
}
```

```
protected static class ImportAddOperation
    extends Java8BaseVisitor<CompilationUnitContext> {

    private final String importType;

    public ImportAddOperation(String newType) {
        importType = newType;
    }

    @Override
    public CompilationUnitContext visitCompilationUnit(CompilationUnitContext ctx) {
        ImportDeclarationContext newImport = new ImportDeclarationContext(ctx, ctx.inv
        newImport.addChild(new CommonToken(Java8Parser.IMPORT, "import"));
        newImport.addChild(new CommonToken(Java8Parser.Identifier, importType));
        ctx.addChild(newImport);
        return super.visitCompilationUnit(ctx);
    }
}
```

It was a start. But how to be sure our new tree is equivalent to a tree as if it was parsed by ANTLR? By looking at the grammar. In our current code, we are not respecting the grammar rules. Our import type must be represented by a tree of typeNameContext.

```
graph antlr_tree {
  label="ANTLR Parse Tree for Java8.g4"
  node [style=filled]

  importDeclaration [label="[importDeclarationContext]"];

  node [label="[packageOrTypeNameContext]"]
  packageOrTypeName1; packageOrTypeName2;

  typeName [label="[TypeNameContext]"];

  singleTypeImport [label="[singleTypeImportDeclarationContext]"];

  import [label="import"];
  java [label="java"];
```

```
  util [label="util"];
  list [label="List"];
  colon [label=";"];
  node [label="."]
  dot1; dot2;

  import -- singleTypeImport;
  singleTypeImport -- importDeclaration;
  typeName -- singleTypeImport;
  colon -- singleTypeImport;

  packageOrTypeName1 -- typeName;
  dot2 -- typeName;
  list -- typeName;

  packageOrTypeName2 -- packageOrTypeName1;
  dot1 -- packageOrTypeName1;
  util -- packageOrTypeName1;

  java -- packageOrTypeName2;
}
```

An easy way to review what our tree should look like is by adding a valid import statement to our test. It's pretty straightforward, but there's one more thing we have to take care of. We need to find out how to build a subtree of "packageOrTypeNameContext" from our type. But wait! Our grammar should handle that, we only need to parse our type, calling the "typeName" rule.

```
@Override
public CompilationUnitContext visitCompilationUnit(CompilationUnitContext ctx) {
    ImportDeclarationContext newImport = new ImportDeclarationContext(ctx, ctx.invoking
    SingleTypeImportDeclarationContext singleTypeImportDeclarationContext =
        new SingleTypeImportDeclarationContext(newImport, newImport.invokingState);
    newImport.addChild(singleTypeImportDeclarationContext);
    singleTypeImportDeclarationContext.addChild(new CommonToken(Java8Parser.IMPORT, "im
    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(this.importType));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    TypeNameContext typeNameContext = parser.typeName();
```

```
    singleTypeImportDeclarationContext.addChild(typeNameContext);
    newImport.addChild(new CommonToken(Java8Parser.COLON, ";"));

    ctx.addChild(newImport);
    return super.visitCompilationUnit(ctx);
}
```

The test now passes, but when debugging I saw something suspicious: an error message was logged in the console, and one node in the tree was referencing an exception. Then, reviewing the code, I decided it was much clearer if I let ANTLR do the whole parsing, not just part of it.

```
@Override
public CompilationUnitContext visitCompilationUnit(CompilationUnitContext ctx) {
    ImportDeclarationContext newImport = new ImportDeclarationContext(ctx, ctx.invoking
    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream("import " + this.importType
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    SingleTypeImportDeclarationContext singleTypeImportDeclaration = parser.singleTypeI
    newImport.addChild(singleTypeImportDeclaration);

    ctx.addChild(newImport);
    return super.visitCompilationUnit(ctx);
}
```

Now it's a little more readable, and it's parsing the import correctly with no complaints. But it still contains that redundant ImportDeclarationContext object that we've made up for no reason. ANTLR can handle it if we start parsing one level higher.

```
@Override
public CompilationUnitContext visitCompilationUnit(CompilationUnitContext ctx) {
    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream("import " + this.importType
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    ImportDeclarationContext newImport = parser.importDeclaration();

    ctx.addChild(newImport);
    return super.visitCompilationUnit(ctx);
}
```

Now it's much better. Let's hope it's not too expensive in terms of performance. Clearly, we should reuse the lexer and tokens from the initial parsing stage. We'll fix it when time is ready.

1.3.5 Fifth test: Generating code

So far we've got ourselves familiar with the first two steps in the process: reading source code, and manipulating it. Now it's time to work on generating code from an AST.

Needless to say, we'll use StringTemplate. It's the natural counterpart of ANTLR, it is easy to learn, and promotes good habits. In our situation, we are trying to answer the question "how can I generate Java sources?", but that's overly ambitious for a first test.

On the top of my mind, I dream of finding a way to somehow mirror a grammar automatically. Let's consider the following rules from our grammar:

```
packageDeclaration
      : packageModifier* 'package' Identifier ('.' Identifier)* ';'
      ;

packageModifier
      : annotation
      ;

annotation
      : normalAnnotation
      | markerAnnotation
      | singleElementAnnotation
      ;

normalAnnotation
      : '@' typeName '(' elementValuePairList? ')'
      ;

markerAnnotation
      : '@' typeName
      ;

singleElementAnnotation
      : '@' typeName '(' elementValue ')'
```

```
      ;

typeName
      : Identifier
      | packageOrTypeName '.' Identifier
      ;

packageOrTypeName
      : Identifier
      | packageOrTypeName '.' Identifier
      ;

elementValuePairList
      : elementValuePair (',' elementValuePair)*
      ;

elementValuePair
      : Identifier '=' elementValue
      ;

elementValue
      : conditionalExpression
      | elementValueArrayInitializer
      | annotation
      ;
```

We could think of analogous StringTemplate rules:

```
packageDeclaration(modifiers, identifier, extraIdentifiers) ::= <<
<modifiers:{ m | <packageModifier(mod=m)>}; separator=" "> package <identifier><extraI
>>

packageModifier(mod) ::= <<
<annotation(a=mod)>
>>

annotation(a) ::= <<
<if(a.normal)><
  normalAnnotation(a=a)><
else><
```

```
  if(a.marker)><
    markerAnnotation(a=a)><
  else><
    singleElementAnnotation(a=a)><
  endif><
endif>
>>

normalAnnotation(a) ::= <<
@<typeName(i=a)>(<if(a.elementValuePairList)><a.elementValuePairList:{ p |<elementValue
>>

markerAnnotation(a) ::= <<
@<typeName(i=a)>
>>

singleElementAnnotation(a) ::= <<
@<typeName(i=a)>(<a.elementValue>)
>>

typeName(i, p) ::= <<
<if(p)><p>.<i><else><i><endif>
>>

packageOrTypeName(i, p) ::= << <! it's the same as typeName !>
<if(p)><p>.<i><else><i><endif>
>>

elementValuePairList(pair) ::= <<
<pair:{ p |<elementValuePair(i=p.identifier, v=p.elementValue)>}; separator=",">
>>

elementValuePair(p, v) ::= <<
<i>=<elementValue(v=v)>
>>

elementValue(v) ::= <<
<if(v.conditionalExpression)><
  conditionalExpression(e=v)><
else><
```

```
  if(v.elementValueArrayInitializer)><
    elementValueArrayInitializer(i=v)><
  else><
    annotation(a=v)><
  endif><
endif>
>>
```

I hope you get the idea. There seems to exist an automatically-generated template set for a given ANTLR grammar, given the AST/ParseTree provides getters for each subtree, so StringTemplate can access them. But don't have that ANTLR->StringTemplate conversion, and still we want to generate code from a AST modelling a Java source file. I see two options: either build that tool ourselves, or build the StringTemplate templates we need for our particular purpose. Let's explore both options in detail.

- Option A: Build an ANTLR->StringTemplate translator

We're pretty confident that, for a clean (no semantic predicates, no embedded logic) ANTLR grammar, there exist a set of StringTemplate templates which can generate valid input for such grammar.

Such translator would involve: a) An ANTLR meta-parser, which reads an ANTLR grammar and generates the StringTemplate templates. b) An AST runtime decorator, which lets StringTemplate access the child nodes via getters.

- Option B: Hand-code the templates for Java8.g4

We've already felt the pain, above. Counting the parser rules gives us an astounding 271 rules. Of course, we could reduce that number to certain extent. But it's a lot of work indeed. Besides that, our work is not usable for other languages in the future, and forces us maintaining the generator manually.

So given this scenario, what would you decide? Each option has pros and cons. If we apply Lean philosophy, we should try to obtain feedback as soon as possible, regardless of the option. Under that perspective, let's review the actual hypothesis behind each option.

- Hypothesis for A: the automatic generation is feasible for any grammar, given it doesn't include logic or semantic predicates.

How could we possibly validate the hypothesis? If it doesn't hold, the whole point of building a generator is unclear. We can inspect some grammars already available for ANTLR, to check for some situations which were not anticipated.

- Hypothesis for B: A custom generator for Java8.g4 is doable for sure, but it'll take a lot of time, and we'll have to write tests for

lots of language constructs.

How long would it be? Hours? Days? We could implement just the templates above, for the "package" rule, and with that information try to estimate the whole grammar.

1.4 Pivoting the prototype

We could discuss each of the options endlessly, and still miss the important challenge we are actually facing. We want to implement a way to customize certain aspects and behavior of the generation templates, ortogonally to the templates themselves. It makes much more sense to focus on that particular problem, than whether we can automate default templates from grammars.

1.4.1 Background

Let's start with the same template to output Java package declarations:

packageDeclaration(modifiers, identifier, extraIdentifiers) ::= « <modifiers:{ m | <packageModifier(mod=m)>}; separator=" "> package <identifier><extraIdentifiers:{ e | .<e>}>; »

That template is saying:

- If there're any package modifiers, then run the "packageModifier" template for each of them, using a blank space

as separator.

- Append a blank space.
- Add the "package" word.
- Append a blank space.
- Append the identifier value.
- If there're any extra identifiers (the package is part of a tree), then append each part, preceded by a dot.

- Append a semicolon.
- Append a new line.

We'd like to be able to change how the template behaves:

- The separator used when calling "packageModifier" templates.
- The blank space.
- The "package" word.
- The identifier value.
- The separator used when calling the anonymous template.
- The semicolon.
- The new line.

Let's try to define selectors for each one of the identified elements:

- .packageDeclaration .packageModifiers : To overwrite the "separator" directive.
- .packageDeclaration "package"::before : To tune the blank space before the "package".
- .packageDeclaration #identifier : to modify the way the identifier value is printed.
- .packageDeclaration #extraIdentifiers : again, needed to overwrite the separator.
- .packageDeclaration ";"::before : to optionally add text before the semicolon.
- .packageDeclaration ";" "s*" : to change whether there's a new line after the semicolon.

This is just an initial example, trying to adapt the standard CSS selectors to this scenario. What we are doing here is modeling the template itself as a DOM or AST, and filtering certain nodes or properties of such tree. But before worrying about that, we need to implement a DSL for the new CSS-like grammar. And that requires us to go on with our initial outline of what we'd like to build.

The next piece in the puzzle is defining the CSS-like properties to apply to the nodes matched by the selectors. If we wanted to use two spaces after the "package" word, and two new lines after the semicolon, we would write it as follows:

```
.packageDeclaration #identifier::before {
  content: "  ";
}

.packageDeclaration ";"::after {
  content: "[newline][newline]";
}
```

CSS Text defines certain properties we could reuse, but only to a certain extent, since their meaning and units are not compatible in some cases.

Anyway, let's try to implement such DSL, starting with a test.

1.4.2 First test: Parsing the CSS-like DSL

This first test consists of invoking logic on a new StringTemplateCSSParser class to parse the above examples. The output will consist of a list, and a map of maps. A list since the selectors are an ordered collection of items, with precedence semantics. A map since the properties are a flattened JSON-like structure of key-value tuples.

```
@Test
public void parses_a_simple_input() {
    String input =
        ".packageDeclaration #identifier::before {\n"
      + "    content: \"  \";\n"
      + "}\n";

    StringTemplateCSSLexer lexer =
        new StringTemplateCSSLexer(
            new ANTLRInputStream(input));

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    StringTemplateCSSParser parser =
        new StringTemplateCSSParser(tokens);
    ParseTree ast = parser.css();
```

```
    Assert.assertNotNull(ast);
}
```

As for the grammar itself, we can reuse the Java8 one, removing almost all the parser rules, and keeping the some of the lexer ones.

```
css
     : selector+ '{' property+ '}' EOF
     ;

selector
     : ('.' | '#') Identifier (':' ':' ('before' | 'after'))?
     ;

property
     : Identifier ':' StringLiteral ';'
     ;

StringLiteral
     : '"' StringCharacters? '"' | '\'' StringCharacters? '\''
     ;

fragment
StringCharacters
     : StringCharacter+
     ;

fragment
StringCharacter
     : ~["\\]
     | EscapeSequence
     ;

// §3.10.6 Escape Sequences for Character and String Literals

fragment
EscapeSequence
     : '\\' [btnfr"'\\]
     | OctalEscape
   |   UnicodeEscape // This is not in the spec but prevents having to preprocess the i
```

```
     ;

fragment
OctalEscape
     : '\\' OctalDigit
     | '\\' OctalDigit OctalDigit
     | '\\' ZeroToThree OctalDigit OctalDigit
     ;

fragment
OctalDigit
     : [0-7]
     ;

fragment
ZeroToThree
     : [0-3]
     ;

// This is not in the spec but prevents having to preprocess the input
fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

fragment
HexDigit
     : [0-9a-fA-F]
     ;

LBRACE : '{';
RBRACE : '}';
LBRACK : '[';
RBRACK : ']';
SEMI : ';';
COMMA : ',';
DOT : '.';

GT : '>';
LT : '<';
```

```
TILDE : '~';
COLON : ':';
MUL : '*';
COLONCOLON : '::';

// §3.8 Identifiers (must appear after all keywords in the grammar)

Identifier
     : JavaLetter JavaLetterOrDigit*
     ;

fragment
JavaLetter
     : [a-zA-Z$_] // these are the "java letters" below 0xFF
     | // covers all characters above 0xFF which are not a surrogate
         ~[\u0000-\u00FF\uD800-\uDBFF]
         {Character.isJavaIdentifierStart(_input.LA(-1))}?
     | // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
         [\uD800-\uDBFF] [\uDC00-\uDFFF]
         {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2), (c
     ;

fragment
JavaLetterOrDigit
     : [a-zA-Z0-9$_] // these are the "java letters or digits" below 0xFF
     | // covers all characters above 0xFF which are not a surrogate
         ~[\u0000-\u00FF\uD800-\uDBFF]
         {Character.isJavaIdentifierPart(_input.LA(-1))}?
     | // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
         [\uD800-\uDBFF] [\uDC00-\uDFFF]
         {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2), (cl
     ;

AT : '@';
ELLIPSIS : '...';

//
// Whitespace and comments
//
```

```
WS  :  [ \t\r\n\u000C]+ -> skip
    ;

COMMENT
    :   '/*' .*? '*/' -> skip
    ;

LINE_COMMENT
    :   '//' ~[\r\n]* -> skip
    ;
```

Our test succeeds. We can live with that "Java" rules in the lexer, since they probably hold true for the CSS specification we are trying to emulate as accurate as possible.

Let's check if the other input we wrote before is parsed correctly as well:

```
@Test
public void parses_another_simple_input() {
    String input =
          "  .packageDeclaration \";\"::after {\n"
        + "    content: \"\\n\\n\";\n"
        + "  }";

    StringTemplateCSSLexer lexer =
        new StringTemplateCSSLexer(
            new ANTLRInputStream(input));

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    StringTemplateCSSParser parser =
        new StringTemplateCSSParser(tokens);
    ParseTree ast = parser.css();
    Assert.assertNotNull(ast);
}
```

The test passes, but ANTLR complains in the console. There're two issues: first, the grammar is not parsing the input correctly; second, the test (and the previous one as well) doesn't detect when the parsing is failing.

```
Running org.acmsl.javacss.css.parser.StringTemplateCSSParserTest
```

```
line 1:31 extraneous input '::' expecting {'#', '{', '.'}
line 1:22 extraneous input '";"' expecting {'#', '{', '.'}
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 sec - in org.acms
```

For now, it's more important to fix the tests. Changing the ErrorHandler to a less permissive strategy is exactly what we look for.

```
parser.setErrorHandler(new BailErrorStrategy());
```

Now the tests fail as they should, we deal with why the grammar doesn't expect "::" in the first test. The second part of the selector, #identifier::before, is not a valid selector according to our grammar. The problem was that we described the consecutive colons as two tokens, whereas the lexer identified them as COLONCOLON:

```
selector
     : ('.' | '#') Identifier (COLONCOLON ('before' | 'after'))?
     ;
```

That fixes the first test. The second input is clearly not supported by our current grammar. We'll need to implement it, but it is not difficult.

```
selector
     : ('.' | '#')? (Identifier | StringLiteral) (COLONCOLON ('before' | 'after'))?
     ;
```

1.4.3 Second test: Listing selectors

Next, we want to define how to list all selectors defined. As before, we'll start with a simple helper which uses the ANTLR-generated parser and provides the two collections we need: the list of selectors, and a Map of Maps containing the properties for each selector. We'll envelop this in a StringTemplateCSSHelper class for now.

```
@Test
public void retrieves_selectors_for_a_simple_input()
{
    String input =
        ".packageDeclaration #identifier::before {\n"
        + "    content: \"  \";\n"
        + "}\n";
```

```
    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input);

    List<String> selectors = helper.getSelectors();

    Assert.assertNotNull(selectors);

    Assert.assertEquals(1, selectors.size());

    Assert.assertEquals(".packageDeclaration #identifier::before", selectors.get(0));
}
```

We can now proceed writing the skeleton of the class to ensure the code compiles.

```
public class StringTemplateCSSHelper
{
    private final String input;
    private List<String> selectors = new ArrayList<String>();

    public StringTemplateCSSHelper(final String input)
    {
        this.input = input;
    }

    public List<String> getSelectors()
    {
        return this.selectors;
    }
}
```

Now that we have our beloved red light, we can try to implement the logic. Notice I'm bypassing the "dumb" implementation here. Anyway, my first attempt wasn't much better either.

```
public List<String> getSelectors()
{
    if (this.selectors == null)
    {
        initialize(this.input);
```

```
    }

    return this.selectors;
}

protected void initialize(String input)
{
    StringTemplateCSSLexer lexer = new StringTemplateCSSLexer(new ANTLRInputStream(inpu

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    StringTemplateCSSParser parser = new StringTemplateCSSParser(tokens);

    ParseTree tree = parser.css();

    Collection<ParseTree> selectorEntries = XPath.findAll(tree, "//selector", parser);

    this.selectors = new ArrayList<String>(selectorEntries.size());

    for (ParseTree selectorEntry : selectorEntries)
    {
        this.selectors.add(selectorEntry.getText());
    }
}
```

The test doesn't pass because it expects just one selector, and the helper is returning two. And we're returning two because it's what the grammar dictates. For now, I feel more confortable with the idea that each block belongs to one selector, even though it's not really true. It's the combination of selectors (and some relationships among them) what allow us to match certain pieces of each template. But again, we'll leave that for later. Meanwhile, let's update the grammar to wrap all selectors into one.

```
css
     : selectorCombination '{' property+ '}' EOF
     ;

selectorCombination
     : selector+
     ;
```

Voilà! It did the trick, although the test is not passing yet, but this time is the test's fault.

```
retrieves_selectors_for_a_simple_input(org.acmsl.javacss.css.StringTemplateCSSHelperTes
org.junit.ComparisonFailure: expected:<.packageDeclaration[ ]#identifier::before> but w
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_selectors_for_a_
```

In our test, we are calling getText() on non-terminal nodes of our Parse-Tree. In our grammar the whitespace is not meaningful, so it's discarded by the lexer and omitted in the token stream and in the final tree. Therefore it's not returning the same input text, and we have to take it into account in our checks.

```
@Test
public void retrieves_selectors_for_a_simple_input()
{
    [..]

    Assert.assertEquals(".packageDeclaration#identifier::before", selectors.get(0));
}
```

We should extend this test to verify it is prepared for input containing more than one block.

It's easy to make the test generic and don't assume a fixed number of blocks, at the cost of making the test non-deterministic.

```
protected void multipleBlockTests(int count)
{
    StringBuilder input =
        new StringBuilder(".packageDeclaration #identifier");

    for (int index = 0; index < count; index++)
    {
        input.append("::before {\n    content: \"  \";\n}\n");
    }

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input.toString());

    List<String> selectors = helper.getSelectors();
```

```
    Assert.assertNotNull(selectors);

    Assert.assertEquals(count, selectors.size());

    for (int index = 0; index < count; index++)
    {
        Assert.assertEquals(".packageDeclaration#identifier" + index + "::before", sele
    }
}

@Test
public void retrieves_selectors_for_an_input_with_several_blocks()
{
    multipleBlockTests((int) (Math.random() * 10));
}
```

However, the input we are building to test our helper is not valid. It generates texts like

```
.packageDeclaration #identifier::before {
    content: "  ";
}

::before {
    content: "  ";
}
```

I forgot the ".packageDeclaration #identifier" part, and we'd better off appending the count to make rules different from each another.

```
protected void multipleBlockTests(int count)
{
    StringBuilder input = new StringBuilder();

    for (int index = 0; index < count; index++)
    {
        input.append(".packageDeclaration #identifier");
        input.append(index);
        input.append("::before {\n    content: \"  \";\n}\n\n");
```

```
    }

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input.toString());

    List<String> selectors = helper.getSelectors();

    Assert.assertNotNull(selectors);

    Assert.assertEquals(count, selectors.size());

    for (int index = 0; index < count; index++)
    {
        Assert.assertEquals(".packageDeclaration#identifier" + index + "::before", sele
    }
}
```

Now some tests pass, some doesn't. For a count of 2 it doesn't work. Let's force that test to dig into the issue. The input we are parsing is:

```
.packageDeclaration #identifier0::before {
    content: "  ";
}

.packageDeclaration #identifier1::before {
    content: "  ";
}
```

Here's the mismatched input error:

Running org.acmsl.javacss.css.parser.StringTemplateCSSParserTest line 5:0 mismatched input '.' expecting <EOF>

If we review our grammar, the problem is obvious. The "css" rule is expecting just one block, so let's make it recursive.

```
css
       : selectorCombination '{' property+ '}' css* EOF
       ;
```

Now it works, so let's revert the test to a random number of blocks.

Our current "multipleBlockTests" method should be "multipleBlockSelectorTests" instead, so let's change that first.

```
protected void multipleBlockSelectorTests(int count)
{
    [..]
}
```

1.4.4 Third test: Reading properties

Next, we want to define how to retrieve the properties associated to a selector. A basic test would simply ask for all properties contained in a block, and verify its contents.

```
@Test
public void retrieves_properties_for_a_simple_input()
{
    String input =
        ".packageDeclaration #identifier::before {\n"
        + "    content: \"  \";\n"
        + "}\n";

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input);

    List<String> selectors = helper.getSelectors();

    Assert.assertNotNull(selectors);

    Assert.assertEquals(1, selectors.size());

    Map<String, String> properties = helper.getProperties(selectors.get(0));

    Assert.assertNotNull(properties);

    Assert.assertEquals(1, properties.size());

    Assert.assertTrue(properties.containsKey("content"));
    Assert.assertEquals("  ", properties.get("content"));
}
```

As always, we need to provide the skeleton to make it compile.

```
public Map<String, String> getProperties(String selector)
{
```

```
    return null;
}
```

As expected, the test fails.

```
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec <<< FAILURE!
retrieves_properties_for_a_simple_input(org.acmsl.javacss.css.StringTemplateCSSHelperTe
java.lang.AssertionError: null
        at org.junit.Assert.fail(Assert.java:86)
        at org.junit.Assert.assertTrue(Assert.java:41)
        at org.junit.Assert.assertNotNull(Assert.java:621)
        at org.junit.Assert.assertNotNull(Assert.java:631)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a
```

My first attempt to implement this was:

```
private Map<String, Map<String, String>> properties;

[..]

protected void initialize(String input)
{
    StringTemplateCSSLexer lexer = new StringTemplateCSSLexer(new ANTLRInputStream(inpu

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    StringTemplateCSSParser parser = new StringTemplateCSSParser(tokens);

    ParseTree tree = parser.css();

    Collection<ParseTree> selectorEntries = XPath.findAll(tree, "//selectorCombination"

    this.selectors = new ArrayList<String>(selectorEntries.size());
    this.properties = new HashMap<String, Map<String, String>>();

    for (ParseTree selectorEntry : selectorEntries)
    {
        String text = selectorEntry.getText();
        this.selectors.add(text);
        Map<String, String> block = retrieveProperties(selectorEntry, parser);
```

```
        this.properties.put(text, block);
    }
}

protected Map<String, String> retrieveProperties(ParseTree selectorEntry, StringTemplat
{
    Map<String, String> result;

    Collection<ParseTree> properties = XPath.findAll(selectorEntry, "//property", parse

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties)
    {
        String key = property.getChild(0).getText();
        String value = property.getChild(2).getText();
        result.put(key, value);
    }

    return result;
}

public Map<String, String> getProperties(String selector)
{
    if (this.properties == null)
    {
        initialize(this.input);
    }

    return this.properties.get(selector);
}
```

However, it didn't make the test pass.

```
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec <<< FAILURE!
retrieves_properties_for_a_simple_input(org.acmsl.javacss.css.StringTemplateCSSHelperTe
java.lang.AssertionError: expected:<1> but was:<0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
```

```
        at org.junit.Assert.assertEquals(Assert.java:118)
        at org.junit.Assert.assertEquals(Assert.java:555)
        at org.junit.Assert.assertEquals(Assert.java:542)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a
```

The reason is that our XPath expression, "//property", is not finding anything. If we review the code, we can figure out why: the first argument to the "retrieveProperties()" should be the "css" node, and we are passing the "selectorCombination" node. A simple change to passing "selectorEntry.getParent()" allows the block to be parsed successfully.

```
protected void initialize(String input)
{
    StringTemplateCSSLexer lexer = new StringTemplateCSSLexer(new ANTLRInputStream(inpu

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    StringTemplateCSSParser parser = new StringTemplateCSSParser(tokens);

    ParseTree tree = parser.css();

    Collection<ParseTree> selectorEntries = XPath.findAll(tree, "//selectorCombination"

    this.selectors = new ArrayList<String>(selectorEntries.size());
    this.properties = new HashMap<String, Map<String, String>>();

    for (ParseTree selectorEntry : selectorEntries)
    {
        String text = selectorEntry.getText();
        this.selectors.add(text);
        Map<String, String> block = retrieveProperties(selectorEntry.getParent(), parse

        this.properties.put(text, block);
    }
}

protected Map<String, String> retrieveProperties(ParseTree cssEntry, StringTemplateCSSH
{
    Map<String, String> result;
```

```
    Collection<ParseTree> properties = XPath.findAll(cssEntry, "//property", parser);

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties)
    {
        String key = property.getChild(0).getText();
        String value = property.getChild(2).getText();
        result.put(key, value);
    }

    return result;
}

public Map<String, String> getProperties(String selector)
{
    if (this.properties == null)
    {
        initialize(this.input);
    }

    return this.properties.get(selector);
}
```

But the test still fails.

```
retrieves_properties_for_a_simple_input(org.acmsl.javacss.css.StringTemplateCSSHelperTe
org.junit.ComparisonFailure: expected:<[  ]> but was:<["  "]>
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a
```

We have two options. Process the value of each property to remove any surrounding quotes, or leave them as they are, and update the test. Since we are not dealing with the nature of the property values, I can live with a test assuming the values can be quoted until it's time to focus on the property values.

```
Assert.assertEquals("\"  \"", properties.get("content"));
```

As we did previously, let's make sure the properties are parsed correctly if the input contains several blocks. Also, we're interested in verifying the properties are specific for each block.

```
protected void multipleBlockPropertyTests(int count)
{
    StringBuilder input = new StringBuilder();

    for (int index = 0; index < count; index++)
    {
        input.append(".packageDeclaration #identifier");
        input.append(index);
        input.append("::before {\n");
        input.append("    content: \"");
        input.append(index);
        input.append("\";\n");
        input.append("}\n");
    }

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input.toString());

    List<String> selectors = helper.getSelectors();

    Assert.assertNotNull(selectors);

    Assert.assertEquals(count, selectors.size());

    for (int index = 0; index < count; index++)
    {
        Map<String, String> properties = helper.getProperties(selectors.get(index));

        Assert.assertNotNull(properties);

        Assert.assertEquals(1, properties.size());

        Assert.assertTrue(properties.containsKey("content"));
        Assert.assertEquals("\"" + index + "\"", properties.get("content"));
    }
}
```

```
@Test
public void retrieves_properties_for_a_simple_input()
{
    multipleBlockPropertyTests(1);
}
```

This new test is equilavent to the previous one, and passes. Let's check what happens for random number of blocks.

```
@Test
public void retrieves_properties_for_an_input_with_multiple_blocks()
{
    multipleBlockPropertyTests((int) (Math.random() * 10));
}
```

It doesn't. It's a bit frustrating, but it also gives a great sense of progress.

```
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.007 sec <<< FAILURE!
retrieves_properties_for_an_input_with_multiple_blocks(org.acmsl.javacss.css.StringTemp
java.lang.AssertionError: expected:<0> but was:<1>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
        at org.junit.Assert.assertEquals(Assert.java:118)
        at org.junit.Assert.assertEquals(Assert.java:555)
        at org.junit.Assert.assertEquals(Assert.java:542)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.multipleBlockPropertyTests
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a


Results :

Failed tests:
  StringTemplateCSSHelperTest.retrieves_properties_for_an_input_with_multiple_blocks:15
```

To make debugging simpler, it's better to return to a deterministic testing context.

```
    @Test
    public void retrieves_properties_for_an_input_with_multiple_blocks()
    {
//        multipleBlockPropertyTests((int) (Math.random() * 10));
```

```
        multipleBlockPropertyTests(2);
    }
```

It still fails, so it's not depending on how many blocks, but only if there's more than one. We've been here already. Probably the input the test builds is incorrect, but ANTLR is not complaining. After a debugging session, I found out the issue. It's a bug that was not detected in previous tests: when we are retrieving all "property" nodes in the ParseTree, we use an XPath expresion "//property" on the "css" root node. However, that collection is the same, regardless which "css" or block we are processing. So at runtime, we iterate over the selectors, and then associate the properties. Due to this bug, and since in our test all keys are the same, all values get overriden in each iteration. We have to retrieve just the properties belonging to the specific selector, and therefore the XPath.findAll() call is not correct. The logic should be: for a "selectorCombination" node, its associated "properties" are the next-to-the-right siblings of type "property". We could write a specific test for that, but we'd probably need to use a mock framework to build the tree. On the other hand, we are already building such trees in our tests. Let's assume that new logic is indirectly tested already. Anyway, why don't we implement it using a Visitor pattern instead?

```
protected Map<String, String> retrieveProperties(ParseTree selectorEntry, StringTemplat
{
    Map<String, String> result;

    Collection<ParseTree> properties = findPropertyNodes(selectorEntry);

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties)
    {
        String key = property.getChild(0).getText();
        String value = property.getChild(2).getText();
        result.put(key, value);
    }

    return result;
}

protected Collection<ParseTree> findPropertyNodes(final ParseTree selectorEntry)
```

```
{
    ParseTree parent = selectorEntry.getParent();
    PropertyVisitor visitor = new PropertyVisitor();
    parent.accept(visitor);

    return visitor.properties;
}

public Map<String, String> getProperties(String selector)
{
    if (this.properties == null)
    {
        initialize(this.input);
    }

    return this.properties.get(selector);
}

protected static class PropertyVisitor
    extends StringTemplateCSSBaseVisitor<ParseTree>
{
    final List<ParseTree> properties = new ArrayList<ParseTree>();

    public PropertyVisitor(final ParseTree parent)
    {
        this.parent = parent;
    }

    @Override
    public ParseTree visitProperty(@NotNull final PropertyContext ctx)
    {
        this.properties.add(ctx);
        return super.visitProperty(ctx);
    }
}
```

Unfortunately it still fails.

```
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec <<< FAILURE!
retrieves_properties_for_an_input_with_multiple_blocks(org.acmsl.javacss.css.StringTemp
```

```
org.junit.ComparisonFailure: expected:<"[0]"> but was:<"[1]">
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.multipleBlockPropertyTests
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a


Results :

Failed tests:
  StringTemplateCSSHelperTest.retrieves_properties_for_an_input_with_multiple_blocks:15
```

The problem now is that in our current grammar, each "css" node becomes the parent of the next "css" node to the right. And that layout makes our visitor end up visiting all "property" nodes. To fix this, there're two options: changing the node tree (the grammar), or ensuring we filter out all "property" nodes whose parent is not matching the selector's. The latter is simpler, but more expensive at runtime, though.

```
protected Map<String, String> retrieveProperties(ParseTree selectorEntry, StringTemplat
{
    Map<String, String> result;

    Collection<ParseTree> properties = findPropertyNodes(selectorEntry);

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties)
    {
        String key = property.getChild(0).getText();
        String value = property.getChild(2).getText();
        result.put(key, value);
    }

    return result;
}

protected Collection<ParseTree> findPropertyNodes(final ParseTree selectorEntry)
{
    ParseTree parent = selectorEntry.getParent();
```

```
    PropertyVisitor visitor = new PropertyVisitor(selectorEntry.getParent());
    parent.accept(visitor);

    return visitor.properties;
}

public Map<String, String> getProperties(String selector)
{
    if (this.properties == null)
    {
        initialize(this.input);
    }

    return this.properties.get(selector);
}

protected static class PropertyVisitor
    extends StringTemplateCSSBaseVisitor<ParseTree>
{
    final List<ParseTree> properties = new ArrayList<ParseTree>();
    final ParseTree parent;

    public PropertyVisitor(final ParseTree parent)
    {
        this.parent = parent;
    }

    @Override
    public ParseTree visitProperty(@NotNull final PropertyContext ctx)
    {
        if (ctx.getParent() == this.parent)
        {
            this.properties.add(ctx);
        }
        return super.visitProperty(ctx);
    }
}
```

We're now back into the green zone, but don't forget we're not calling "multipleBlockPropertyTests()" with random values. When switching back

to relying to input composed of a random number of blocks, the test fails again.

```
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.008 sec <<< FAILURE!
retrieves_properties_for_an_input_with_multiple_blocks(org.acmsl.javacss.css.StringTemp
java.lang.AssertionError: expected:<0> but was:<1>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
        at org.junit.Assert.assertEquals(Assert.java:118)
        at org.junit.Assert.assertEquals(Assert.java:555)
        at org.junit.Assert.assertEquals(Assert.java:542)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.multipleBlockPropertyTests
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.retrieves_properties_for_a


Results :

Failed tests:
  StringTemplateCSSHelperTest.retrieves_properties_for_an_input_with_multiple_blocks:15
```

This test is interesting, because it uncovers one aspect we are not dealing with: invalid inputs. The "multipleBlockPropertyTest()" method, when receiving a 0 as parameter, generates invalid input. Consequently, the parser complaints. Surprisingly, the XPath expression returns a match consisting of an empty string.

For now, let's annotate this for the next iteration: manage errors in ANTLR parsers / deal with invalid input. Meanwhile we have to ensure we are not testing empty or wrong inputs.

```
@Test
public void retrieves_properties_for_an_input_with_multiple_blocks()
{
    multipleBlockPropertyTests((int) (Math.random() * 10) + 1));
}
```

1.4.5 Fourth test: Matching selectors

The next logical step is to find all matching selectors, for a given AST. We'll use it in the context of StringTemplate templates, but need not to be constrained to any specific AST.

It's worth reviewing our selector syntax:

- a rule starting with a dot matches node classes.
- a rule starting with a hash identifies unique nodes.
- a rule with surrounding quotes selects nodes whose text matches the rule text.
- any optional pseudo-element (::before or ::after) do not customize the nodes themselves, but add new siblings.

What would be a good scenario for checking if a selector is matching the correct node in a given AST? As always, start by something simple, and increase in complexity gradually.

For our AST, a good fit could be the one produced by parsing this package declaration:

package com.foo.bar;

That is,

For the CSS, this would be simple enough: add a blank space before the semicolon.

```
.packageDeclaration ";"::before {
   content: " ";
}
```

Notice we're not testing whether the final output included the blank space before the semicolon. For now, we only want to implement the logic to find out which CSS block applies to each node in the AST. Or even simpler: testing every node has no matching selectors, but the semicolon. Additionally, since we have no immediate need for a better design, we'll implement the logic in our current helper class.

```
@Test
public void finds_the_matching_selector()
{
    String javaInput = "package com.foo.bar;";

    String cssInput =
          ".packageDeclaration \";\"::before {\n"
        + "   content: \" \";\n"
        + "}\n";

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(cssInput);
```

```
    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(javaInput));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    ParseTree ast = parser.compilationUnit();

    Collection<ParseTree> matches = XPath.findAll(ast, "//';'", parser);

    Assert.assertNotNull(matches);
    Assert.assertEquals(1, matches.size());

    ParseTree semiColon = matches.toArray(new ParseTree[1])[0];
    Assert.assertNotNull(semiColon);

    String matchedSelectors = helper.retrieveMatchingSelectors(semiColon);

    Assert.assertNotNull(matchedSelectors);
    Assert.assertEquals(".packageDeclaration \";\"::before", matchedSelectors);
}
```

This test is more involved, and of course it won't compile until we write the new method "retrieveMatchingSelectors()".

```
public String retrieveMatchingSelectors(ParseTree semiColon)
{
    return null;
}
```

And as expected, the null assertion makes the test fail.

```
Tests run: 5, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec <<< FAILURE!
finds_the_matching_selector(org.acmsl.javacss.css.StringTemplateCSSHelperTest)  Time el
java.lang.AssertionError: null
        at org.junit.Assert.fail(Assert.java:86)
        at org.junit.Assert.assertTrue(Assert.java:41)
        at org.junit.Assert.assertNotNull(Assert.java:621)
        at org.junit.Assert.assertNotNull(Assert.java:631)
        at org.acmsl.javacss.css.StringTemplateCSSHelperTest.finds_the_matching_selecto
```

```
Results :

Failed tests:
  StringTemplateCSSHelperTest.finds_the_matching_selector:187 null
```

Now we have to actually implement the "retrieveMatchingSelectors()" method, and it's not straight-forward. We have to find which selector defines a set which includes the input AST. Currently, we are storing the combination of selectors as one piece of text. Parsing it again makes no sense whatsoever, so we should invest some time fixing this first. Instead of a list of Strings, it makes more sense to describe it as a list of list of Strings.

```
public class StringTemplateCSSHelper
{
    private final String input;
    private List<List<String>> selectors;
    private Map<List<String>, Map<String, String>> properties;

    public StringTemplateCSSHelper(final String input)
    {
        this.input = input;
    }

    public List<List<String>> getSelectors()
    {
        if (this.selectors == null)
        {
            initialize(this.input);
        }

        return this.selectors;
    }

    protected void initialize(String input)
    {
        StringTemplateCSSLexer lexer = new StringTemplateCSSLexer(new ANTLRInputStream

        CommonTokenStream tokens = new CommonTokenStream(lexer);

        StringTemplateCSSParser parser = new StringTemplateCSSParser(tokens);
```

```
    ParseTree tree = parser.css();

    Collection<ParseTree> selectorCombinations = XPath.findAll(tree, "//selectorCo

    this.selectors = new ArrayList<List<String>>(selectorCombinations.size());
    this.properties = new HashMap<List<String>, Map<String, String>>();

    for (ParseTree selectorCombination : selectorCombinations)
    {
        List<String> currentSelectors = new ArrayList<String>(selectorCombination.g
        this.selectors.add(currentSelectors);

        for (int index = 0; index < selectorCombination.getChildCount(); index++)
        {
            String text = selectorCombination.getChild(index).getText();
            currentSelectors.add(text);
        }
        Map<String, String> block = retrieveProperties(selectorCombination, parser)

        this.properties.put(currentSelectors, block);
    }
}

protected Map<String, String> retrieveProperties(ParseTree selectorEntry, StringTe
{
    Map<String, String> result;

    Collection<ParseTree> properties = findPropertyNodes(selectorEntry);

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties)
    {
        String key = property.getChild(0).getText();
        String value = property.getChild(2).getText();
        result.put(key, value);
    }

    return result;
```

```
    }

    protected Collection<ParseTree> findPropertyNodes(final ParseTree selectorEntry)
    {
        ParseTree parent = selectorEntry.getParent();
        PropertyVisitor visitor = new PropertyVisitor(selectorEntry.getParent());
        parent.accept(visitor);

        return visitor.properties;
    }

    public Map<String, String> getProperties(List<String> selector)
    {
        if (this.properties == null)
        {
            initialize(this.input);
        }

        return this.properties.get(selector);
    }

    [..]
}
```

The current StringTemplateCSSHelperTest needs to be modified because of the API changes.

```
    @Test
    public void retrieves_selectors_for_a_simple_input()
    {
        String input =
            ".packageDeclaration #identifier::before {\n"
            + "    content: \"  \";\n"
            + "}\n";

        StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input);

        List<List<String>> selectorCombinations = helper.getSelectors();

        Assert.assertNotNull(selectorCombinations);
```

```
    Assert.assertEquals(1, selectorCombinations.size());

    List<String> selectors = selectorCombinations.get(0);
    Assert.assertEquals(2, selectors.size());

    Assert.assertEquals(".packageDeclaration", selectors.get(0));
    Assert.assertEquals("#identifier::before", selectors.get(1));
}

protected void multipleBlockSelectorTests(int count)
{
    StringBuilder input = new StringBuilder();

    for (int index = 0; index < count; index++)
    {
        input.append(".packageDeclaration #identifier");
        input.append(index);
        input.append("::before {\n    content: \"  \";\n}\n\n");
    }

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input.toString())

    List<List<String>> selectorCombinations = helper.getSelectors();

    Assert.assertNotNull(selectorCombinations);

    Assert.assertEquals(count, selectorCombinations.size());

    for (int index = 0; index < count; index++)
    {
        List<String> selectors = selectorCombinations.get(index);

        Assert.assertEquals(".packageDeclaration", selectors.get(0));
        Assert.assertEquals("#identifier" + index + "::before", selectors.get(1));
    }
}

@Test
public void retrieves_selectors_for_an_input_with_several_blocks()
```

```
{
    multipleBlockSelectorTests((int) (Math.random() * 10));
}

protected void multipleBlockPropertyTests(int count)
{
    StringBuilder input = new StringBuilder();

    for (int index = 0; index < count; index++)
    {
        input.append(".packageDeclaration #identifier");
        input.append(index);
        input.append("::before {\n");
        input.append("    content: \"");
        input.append(index);
        input.append("\";\n");
        input.append("}\n");
    }

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(input.toString())

    List<List<String>> selectors = helper.getSelectors();

    Assert.assertNotNull(selectors);

    Assert.assertEquals(count, selectors.size());

    for (int index = 0; index < count; index++)
    {
        Map<String, String> properties = helper.getProperties(selectors.get(index))

        Assert.assertNotNull(properties);

        Assert.assertEquals(1, properties.size());

        Assert.assertTrue(properties.containsKey("content"));
        Assert.assertEquals("\"" + index + "\"", properties.get("content"));
    }
}
```

```
    [..]
}
```

Even though we've had to change the tests, the changes were pretty simple, and the test passed nicely. Now we can go back to work: implementing retrieveMatchingSelectors(ParseTree).

```
public List<String> retrieveMatchingSelectors(ParseTree node) {
    List<String> result = null;

    for (List<String> selectors : this.selectors) {
        if (match(selectors, node)) {
            result = selectors;
            break;
        }
    }

    return result;
}
```

I noticed I actually missed to complete the previous refactoring. The matching selectors are actually a list, not just a string made of selectors. That requires adapting the test as well.

```
@Test
public void finds_the_matching_selector(){
    String javaInput = "package com.foo.bar;";

    String cssInput =
          ".packageDeclaration \";\"::before {\n"
        + "   content: \" \";\n"
        + "}\n";

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(cssInput);

    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(javaInput));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    ParseTree ast = parser.compilationUnit();
```

```
    Collection<ParseTree> matches = XPath.findAll(ast, "//';'", parser);

    Assert.assertNotNull(matches);
    Assert.assertEquals(1, matches.size());

    ParseTree semiColon = matches.toArray(new ParseTree[1])[0];
    Assert.assertNotNull(semiColon);

    List<String> matchedSelectors = helper.retrieveMatchingSelectors(semiColon);

    Assert.assertNotNull(matchedSelectors);
    Assert.assertEquals(2, matchedSelectors.size());
    Assert.assertEquals(".packageDeclaration", matchedSelectors.get(0));
    Assert.assertEquals("\";\"::before", matchedSelectors.get(1));
}
```

In the implementation I've started writing, I introduced a new method, match(List<String>, ParseTree), to isolate the logic to check whether the root node is part of the nodes described by the selectors. However, that's not really what we need. We cannot restrain ourselves to checking the root node. The check requires evaluating the node not as a root, but in context of the complete AST. We need to pass such complete AST as well.

```
[..]
List<String> matchedSelectors = helper.retrieveMatchingSelectors(semiColon, ast);
[..]

public List<String> retrieveMatchingSelectors(ParseTree node, ParseTree ast) {
    List<String> result = null;

    for (List<String> selectors : getSelectors())
    {
        if (match(selectors, node, ast))
        {
            result = selectors;
            break;
        }
    }

    return result;
```

```
}

protected boolean match(List<String> selectors, ParseTree node, ParseTree ast) {
    return false;
}
```

Now we have to implement the algorithm: Given a node in a tree, and a list of ordered filters describing node properties and inheritance relations among nodes, find out whether given node matches the filters. We should try to stop as soon as we know the node does not match any filter. Otherwise, it'd be more useful to retrieve all matching nodes for given selectors, and afterwards check whether our node is part of such list.

Let's do our first attempt to implement the algorithm. We'll be traversing two ordered structures: the selector list, and the tree, so we'll need two references. We start with the first selector, and search the tree for the first node for which the selector is true. If none is found, we are finished, otherwise we have to check if the matched node is our candidate. If it's our node, and there're no other selectors, we conclude there's a match. If it's our node, and there're more selectors, we just have to check if our node matches the remaining selectors. That check defines the outcome of the algorithm. If it's not our node, but it's an ancestor of our node, then we apply the algorithm to the subtree whose root is the recent match, providing the list of selectors not processed yet. If it's not our node, and our node doesn't descend from it, then we skip that node, try to find another match in the tree, and apply the same steps for the next match.

However, the simplest way to crawl a ParseTree is with a visitor. If we managed to implement the algorithm in a custom visitor, it would be simpler and shorter. For that, we need to find out whether we can stop the crawl on demand. Otherwise it'd be very innefficent. Fortunately, ANTLR visitor logic internally calls "parseTree.accept(visitor)", and such "accept" method calls "visitor.visitTerminal()" for a TerminalNode, "visitor.visitChildren()" for a "rule" node, or "visitor.visitErrorNode()" for an error node. So it's the visitor who manages when to stop the crawl.

This approach would then iterate through the selector list, and call a visitor-based tree traversal, finding the subtrees whose root node matches that selector. If our node is in that set, we add the selector to the list of matching selectors, and repeat the process with the next selector.

It's easier to start with a test, even if we are dealing with ParseTree objects, which add certain complexity to the test.

```
@Test
```

```
public void selector_found(){
    String javaInput = "package com.foo.bar;";

    String cssInput =
        ".packageDeclaration \";\"::before {\n"
        + "   content: \" \";\n"
        + "}\n";

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(cssInput);

    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(javaInput));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    ParseTree ast = parser.compilationUnit();

    Collection<ParseTree> matches = XPath.findAll(ast, "//';'", parser);

    Assert.assertNotNull(matches);
    Assert.assertEquals(1, matches.size());

    ParseTree semiColon = matches.toArray(new ParseTree[1])[0];
    Assert.assertNotNull(semiColon);

    matches = XPath.findAll(ast, "//'package'", parser);

    Assert.assertNotNull(matches);
    Assert.assertEquals(1, matches.size());

    ParseTree packageNode = matches.toArray(new ParseTree[1])[0];
    Assert.assertNotNull(packageNode);

    List<String> selectors = Arrays.asList(".packageDeclaration", "\";\"::before");

    Assert.assertTrue(helper.match(selectors, semiColon, ast));
    Assert.assertFalse(helper.match(selectors, packageNode, ast));
}
```

What we're testing here is that, given two selectors (.packageDeclaration and ";"::before) and the AST resulting from parsing "package com.foo.bar;", the node associated to the semicolon matches the selectors, whereas the node

associated to the "package" keyword, does not.

```
protected boolean match(List<String> selectors, ParseTree node, ParseTree ast) {
    SelectorMatchVisitor visitor = new SelectorMatchVisitor(selectors, node);

    visitor.visit(ast);

    return visitor.matchFound();
}
```

We'll isolate everything in the SelectorMatchVisitor class. Before we think about the algorith itself, we know for sure we'll need to implement the logic to check whether a selector matches a given node in an AST.

```
/**
 * An ANTLR visitor to find whether some selectors match certain subtrees of an AST.
 * @author <a href="mailto:queryj@acm-sl.org">Jose San Leandro</a>
 * @since 3.0
 * Created: 2015/03/20 10:35
 */
@ThreadSafe
public class SelectorMatchVisitor
    extends Java8BaseVisitor<ParseTree> {

    boolean match = false;
    final ParseTree focusNode;
    final List<String> selectors;

    public SelectorMatchVisitor(List<String> selectors, ParseTree focusNode) {
        this.selectors = selectors;
        this.focusNode = focusNode;
    }

    public boolean matchFound() {
        return this.match;
    }

    protected boolean matches(final ParseTree node, final String currentSelector) {
        boolean result = false;
```

```
        return result;
    }
}
```

Let's write our test then.

```
@RunWith(JUnit4.class)
public class SelectorMatchVisitorTest {

    @Test
    public void compares_selectors_correctly() {
        String javaInput = "package com.foo.bar;";

        Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(javaInput));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        Java8Parser parser = new Java8Parser(tokens);
        ParseTree ast = parser.compilationUnit();

        Collection<ParseTree> matches = XPath.findAll(ast, "//';'", parser);

        Assert.assertNotNull(matches);
        Assert.assertEquals(1, matches.size());

        ParseTree semiColon = matches.toArray(new ParseTree[1])[0];
        Assert.assertNotNull(semiColon);

        matches = XPath.findAll(ast, "//'package'", parser);

        Assert.assertNotNull(matches);
        Assert.assertEquals(1, matches.size());

        ParseTree packageNode = matches.toArray(new ParseTree[1])[0];
        Assert.assertNotNull(packageNode);

        List<String> selectors = Arrays.asList(".packageDeclaration", "\";\"::before")

        SelectorMatchVisitor visitor = new SelectorMatchVisitor(selectors, ast);

        Assert.assertTrue(visitor.matches(ast.getChild(0), ".packageDeclaration"));
        Assert.assertTrue(visitor.matches(semiColon, "\";\"::before"));
```

```
    }
}
```

As we defined above, if the selector starts with a colon, we'll match those nodes whose "type" is precisely that selector.

No surprise the test fails.

```
java.lang.AssertionError
        at org.junit.Assert.fail(Assert.java:86)
        at org.junit.Assert.assertTrue(Assert.java:41)
        at org.junit.Assert.assertTrue(Assert.java:52)
        at org.acmsl.javacss.css.SelectorMatchVisitorTest.compares_ByClass_selectors_co
```

The only problem is to figure out how can we retrieve the type of the node. The "payload" attribute of the ParseTree is just an object, but after debugging it turns out it can be either a CommonToken, or a Context. For our first assert to be true, we rely on ANTLR's internal convention of naming the Context object after the name of the rule that creates it.

```
protected boolean matches(final ParseTree node, final String currentSelector) {
    boolean result = false;

    if (currentSelector.startsWith(".")) {
        // class selector
        String className = node.getPayload().getClass().getSimpleName();

        // remove any container class, if it's anonymous
        if (className.contains("$")) {
            className = className.substring(className.lastIndexOf("$"));
        }
        // uncapitalize the first letter
        if (className.length() > 1) {
            className = className.substring(0, 1).toLowerCase(Locale.getDefault()) + cl
        }
        // remove the trailing "Context".
        className = className.substring(0, className.lastIndexOf("Context"));

        result = currentSelector.equals("." + className);
    } else if (currentSelector.startsWith("\"")) {
        result = node.getPayload().toString().equals(currentSelector.substring(1, curre
```

```
    }

    return result;
}
```

Now our first assert passes, so the by-class selector should work just fine. The other selector we need to support right now is the one matching node texts. In these cases, the payload of the node must be a CommonToken, and we can compare its value.

```
if (currentSelector.startsWith(".")) {
    [..]
} else if (currentSelector.startsWith("\"")) {
    Object payload = node.getPayload();

    if (payload instanceof CommonToken) {

        String value = ((CommonToken) payload).getText();

        String selectorPart = currentSelector.substring(1);
        selectorPart = selectorPart.substring(0, selectorPart.indexOf("\""));

        result = value.equals(selectorPart);
    }
}
```

We can implement the algorithm now. ANTLR generates visitor methods for each rule. It'd be very inconvenient to implement all of them, and it'd make the visitor very fragile regarding changes on the Java grammar. Fortunately, we can influence the visitor's crawling process just by overriding three methods: visitChildren(), visitTerminal(), and visit().

The algorithm works as follows: the AST is processed by ANTLR's visitor implementation. For each node, our visitor checks whether such node is matched by the current selector, taken from the list. If it matches, we "consume" the selector and let the tree traversal process continue, with the next selector in the list, unless there's no other selector remaining, in which case we check if the node being processed matches our original focus node. In such case, we have a match, and should stop the process. If there current node doesn't match the selector, we try to process its children.

```
public class SelectorMatchVisitor
```

```
extends Java8BaseVisitor<ParseTree> {
boolean match = false;

final ParseTree focusNode;
final List<String> selectors;
final Iterator<String> iterator;
String currentSelector = null;

[..]

@Override
public ParseTree visitChildren(RuleNode node) {
    ParseTree result;

    if (!this.match) {
        result = visit(node);
    } else {
        result = null;
    }

    return result;
}

@Override
public ParseTree visitTerminal(TerminalNode node) {
    ParseTree result;

    if (!this.match) {
        result = visit(node);
    } else {
        result = null;
    }

    return result;
}

@Override
public ParseTree visit(ParseTree node) {
    ParseTree result = null;
```

```
        if (!this.match) {
            if (matches(node, this.currentSelector)) {
                if (this.iterator.hasNext()) {
                    this.currentSelector = this.iterator.next();
                    result = super.visit(node);
                } else if (focusNode.equals(node)) {
                    match = true;
                }
            } else if (node.getChildCount() > 0) {
                if (!this.match) {
                    result = super.visitChildren((RuleNode) node);
                }
            }
        }
        return result;
    }

    [..]
}
```

This algorithm has limitations and will be changed later, at least whenever we need to implement selector operators. But meanwhile, it passes our tests.

1. An unexpected bug

   However, I found some other failing test, which was unexpected, and seems to happen randomly:

   ```
   Tests run: 6, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.075 sec <<< FAIL
   retrieves_selectors_for_an_input_with_several_blocks(org.acmsl.javacss.css.StringT
   java.lang.AssertionError: expected:<0> but was:<1>
           at org.junit.Assert.fail(Assert.java:88)
           at org.junit.Assert.failNotEquals(Assert.java:743)
           at org.junit.Assert.assertEquals(Assert.java:118)
           at org.junit.Assert.assertEquals(Assert.java:555)
           at org.junit.Assert.assertEquals(Assert.java:542)
           at org.acmsl.javacss.css.StringTemplateCSSHelperTest.multipleBlockSelector
   ```

   The problem is that, if the test randomly chooses 0 as the number of CSS blocks to parse, ANTLR cannot parse it, and the logic misbehaves. But **StringTemplateCssHelper** cannot return any selectors

for an empty CSS. After debugging, we can figure out why: ANTLR's *XPath.findAll()* returns a single node representing the error, upon a wrong input. In this case, we prefer ANTLR not to try to recover, or swallow invalid input silently, but throw a runtime exception instead.

To reproduce the error, we can write a test for it:

```
public class StringTemplateCSSHelperTest {
    [..]
    @Test
    public void throws_a_runtime_exception_on_empty_input() {
        try {
            multipleBlockSelectorTests(0);
            Assert.fail("Should throw an exception when parsing empty input");
        } catch (RuntimeException parsingCancelled) {
            Assert.assertTrue(parsingCancelled instanceof ParseCancellationExcepti
        }
    }
    [..]
}
```

The test fails as we were hoping. Lucky us, ANTLR supports that behavior out of the box, just by using a different error handling strategy:

```
public class StringTemplateCSSHelper {
    [..]
    protected void initialize(String input) {
        [..]
        StringTemplateCSSParser parser = new StringTemplateCSSParser(tokens);

        parser.setErrorHandler(new BailErrorStrategy());

        ParseTree tree = parser.css();

        [..]
    }

    [..]
}
```

And now we're green again.

1.4.6 Fifth test: Implementing ::before pseudo-class

For each node in the parse tree, we now can retrieve its selectors. It makes sense then to implement the first one. But we have a minor refactoring to do first. The *retrieveMatchingSelectors()* method currently retrieves a list of Strings, but we need to return a list of *css* entries, as defined in the CSS grammar.

Let's change our test accordingly:

```
@Test
public void finds_the_matching_css() {
    String javaInput = "package com.foo.bar;";

    String cssInput =
          ".packageDeclaration \";\"::before {\n"
        + "   content: \" \";\n"
        + "}\n";

    StringTemplateCSSHelper helper = new StringTemplateCSSHelper(cssInput);

    Java8Lexer lexer = new Java8Lexer(new ANTLRInputStream(javaInput));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Java8Parser parser = new Java8Parser(tokens);
    ParseTree ast = parser.compilationUnit();

    Collection<ParseTree> matches = XPath.findAll(ast, "//';'", parser);

    Assert.assertNotNull(matches);
    Assert.assertEquals(1, matches.size());

    ParseTree semiColon = matches.toArray(new ParseTree[1])[0];
    Assert.assertNotNull(semiColon);

    List<Css> matchedCss = helper.retrieveMatchingCss(semiColon, ast);

    Assert.assertNotNull(matchedCss);
    Assert.assertEquals(1, matchedCss.size());
    Css css = matchedCss.get(0);
    Assert.assertNotNull(css);

    List<String> matchedSelectors = matchedCss.getSelectors();
```

```
    Assert.assertNotNull(matchedSelectors);
    Assert.assertEquals(2, matchedSelectors.size());
    Assert.assertEquals(".packageDeclaration", matchedSelectors.get(0));
    Assert.assertEquals("\";\"::before", matchedSelectors.get(1));
}
```

The changes were pretty simple. They include renaming *retrieveMatchingSelectors()* to *retrieveMatchingCss()*.

Our initial **Css** class is just a bean:

```
public class Css {
    /**
     * The selectors.
     */
    private List<String> selectors = new ArrayList<>();

    /**
     * Adds a new selector.
     * @param selector the selector.
     */
    public void addSelector(String selector) {
        this.selectors.add(selector);
    }

    /**
     * Retrieves the selectors.
     * @return such information.
     */
    public List<String> getSelectors() {
        return selectors;
    }
}
```

We now have to refactor *retrieveMatchingCss()* method, which now has to return a list of **Css** instances.

```
public List<Css> retrieveMatchingCss(ParseTree node, ParseTree ast) {
    List<Css> result = new ArrayList<Css>();
```

```
    for (List<String> selectors : getSelectors()) {
        if (match(selectors, node, ast)) {
            Css css = new Css();
            for (String selector : selectors) {
                css.addSelector(selector);
            }
            result.add(css);
            break;
        }
    }

    return result;
}
```

The tests still pass. Great! Now, we need to test whether the CSS properties are available as well.

```
    @Test
    public void finds_the_matching_css() {
        [..]
        List<Property<?>> properties = css.getProperties();
        Assert.assertNotNull(properties);
        Assert.assertEquals(1, properties.size());
        Property<String> content = (Property<String>) properties.get(0);
        Assert.assertNotNull(content);
        Assert.assertEquals("content", content.getKey());
        Assert.assertEquals(" ", content.getValue());
}
```

That requires us to add the *Property* collection the **Css** class, and implement the **Property** class as well. We'll use a parameter to represent the type of the value of each property.

```
public class Css {
    [..]

    /**
     * The CSS properties.
     */
    private List<Property<?>> properties = new ArrayList<>();
```

```
    [..]

    /**
     * Retrieves the properties.
     * @return such information.
     */
    public List<Property<?>> getProperties() {
        return properties;
    }
}
```

The first version of the **Property** class could be as follows.

```
public class Property<T> {
    private final String key;
    private final T value;

    public Property(String key, T value) {
        this.key = key;
        this.value = value;
    }

    public String getKey() {
        return key;
    }

    public T getValue() {
        return value;
    }
}
```

The test now fails, as expected, when checking how many properties are found.

```
java.lang.AssertionError:
Expected :1
Actual   :0
```

It shouldn't be too difficult to annotate the properties when building **Css** instances.

```
public List<Css> retrieveMatchingCss(ParseTree node, ParseTree ast) {
    [..]
            for (Map.Entry<String, String> entry : this.properties.get(selectors).entry
                css.addProperty(new Property<String>(entry.getKey(), entry.getValue()))
            }
            result.add(css);
    [..]
}
```

We have to remember later to write tests for supporting different types of properties: integer, arrays, etc.

```
public class Css {
    [..]
    public void addProperty(final Property<?> property) {
        this.properties.add(property);
    }
}
```

When running the tests, one assertion fails, and it has to do with a past decision of including the quotes in property values. It makes sense to fix that now.

```
protected Map<String, String> retrieveProperties(ParseTree selectorEntry, StringUtils s
{
    Map<String, String> result;

    Collection<ParseTree> properties = findPropertyNodes(selectorEntry);

    result = new HashMap<String, String>(properties.size());

    for (ParseTree property : properties) {
        String key = property.getChild(0).getText();
        String value = stringUtls.unquote(stringUtls.unquote(property.getChild(2).getTe
        result.put(key, value);
    }

    return result;
}
```

We added a the reference to **StringUtils**, since it's being used in a loop.

```
protected void initialize(String input)
{
    [..]

    StringUtils stringUtils = StringUtils.getInstance();

    for (ParseTree selectorCombination : selectorCombinations) {
        [..]

        Map<String, String> block = retrieveProperties(selectorCombination, stringUtils
        [..]
    }
}
```

However, this change is incompatible with our test block *multipleBlock-PropertyTests(int)*, so we have to update that as well.

```
for (int index = 0; index < count; index++) {
    [..]
    Assert.assertEquals(String.valueOf(index), properties.get("content"));
}
```

Now we have a way to retrieve the properties to apply for any given node in the AST. Next step is to interpret the semantics of each CSS block, and perform the actions each one defines, if any. To keep this as simple as possible at the beginning, we'll use a *Factory method* pattern, to be able to instantiate the correct entities representing the actions to perform, for each CSS block.

Let's call this factory **CssActionFactory**. Our initial test is simple: if the CSS block contains any *::before/* pseudo selector, then we instantiate our first **CssAction** class. Otherwise a **NullCssAction** will be returned. If we need it, we'll probably switch to a chain-of-responsibility approach, since it would simplify how to process several actions for a given CSS block.

```
@RunWith(JUnit4.class)
public class CssActionFactoryTest {
    @Test
    public void when_there_is_nothing_to_do_createAction_returns_NullCssAction() {
        CssActionFactory factory = new CssActionFactory();
```

```
        Css css = new Css();
        css.addSelector(".rule");
        css.addProperty(new Property<String>("content", "value"));

        CssAction action = factory.createAction(css);
        Assert.assertNotNull(action);
        Assert.assertTrue(action instanceof NullCssAction);
    }
}
```

TODO Note: research tools to visually design ideas, diagrams, etc.

The **CssActionFactory**, as expected by our test, can be simply:

```
public class CssActionFactory {
    public CssAction createAction(Css css) {
        return null;
    }
}
```

We need the **CssAction** interface as well, but we can leave it empty for now.

```
public interface CssAction {
}
```

Last, our test demands a *null object* implementation for that interface.

```
public class NullCssAction
    implements CssAction {
}
```

Finally, our test compiles. We expect it to fail, since we're not using **NullCssAction** yet.

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec <<< FAILURE!
when_there_is_nothing_to_do_createAction_returns_NullCssAction(org.acmsl.javacss.css.Cs
java.lang.AssertionError: null
        at org.junit.Assert.fail(Assert.java:86)
```

The fix is simple, and allows us to move forward in our TDD iteration.

```
public class CssActionFactory {
    public CssAction createAction(Css css) {
        return new NullCssAction();
    }
}
```

Now that the test passes, and there's not much refactoring to do until we evolve our **CssAction** design, we have to write another test. This time, we want to describe what it would be to have a real **CssAction** for a *::before* pseudo selector.

```
public class CssActionFactoryTest {
    [..]
    @Test
    public void for_a_before_pseudo_selector_createAction_returns_InsertBeforeCssActio
        CssActionFactory factory = new CssActionFactory();
        Css css = new Css();
        css.addSelector(".rule::before");
        css.addProperty(new Property<String>("content", "value"));
        CssAction action = factory.createAction(css);
        Assert.assertNotNull(action);
        Assert.assertTrue(action instanceof InsertBeforeCssAction);
    }
}
```

For the sake of the TDD cycle, let's write an empty **InsertBeforeCssAction**.

```
public class InsertBeforeCssAction
    implements CssAction {
}
```

The new test fails, of course.

```
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.009 sec <<< FAILURE!
for_a_before_pseudo_selector_createAction_returns_InsertBeforeCssAction(org.acmsl.javac
java.lang.AssertionError: null
        at org.junit.Assert.fail(Assert.java:86)
```

We can implement now the logic to map CSS selectors, to **CssAction** instances. To save our time, let's short-circuit the naive implementation of hard-coding the outcome expected by the test, but without worrying too much about it for now.

```
public class CssActionFactory {
    public CssAction createAction(Css css) {
        CssAction result = null;

        for (String selector : css.getSelectors()) {
            if (selector.contains("::before")) {
                result = new InsertBeforeCssAction();
                break;
            }
        }

        if (result == null) {
            result = new NullCssAction();
        }

        return result;
    }
}
```

The test passes. We can now write another test to allow us to implement the actual logic of the **::before** pseudo-selector. In our approach, CSS semantics apply to text, not AST or parse tree nodes. That makes our test much clearer.

```
public class InsertBeforeCssActionTest {
    @Test
    public void execute_inserts_the_content_before_correctly() {
        Css css = new Css();
        css.addSelector(".rule::before");
        css.addProperty(new Property<String>("content", "css-prefix>"));
        CssActionFactory factory = new CssActionFactory();
        CssAction action = factory.createAction(css);
        Assert.assertNotNull(action);
        String newText = action.execute("my text");
        Assert.assertNotNull(newText);
        Assert.assertEquals("css-prefix>my text", newText);
    }
}
```

We haven't declared any *execute(String)* method yet in **CssAction** yet, but now we actually need it.

```
public interface CssAction {
    /**
     * Applies the action to given text.
     * @param text the text.
     * @return the outcome of applying the action.
     */
    String execute(String text);
}
```

Updating the **NullCssAction** is straightforward.

```
public class NullCssAction
    implements CssAction {
    @Override
    public String execute(String text) {
        return text;
    }
}
```

Which is identical to the implementation for **InsertBeforeCssAction** at this stage of the TDD lifecycle.

```
public class InsertBeforeCssAction
    implements CssAction {
    @Override
    public String execute(String text) {
        return text;
    }
}
```

The test must fail since we're not modifying the original text.

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec <<< FAILURE!
execute_inserts_the_content_before_correctly(org.acmsl.javacss.css.InsertBeforeCssActio
org.junit.ComparisonFailure: expected:<[css-prefix>]my text> but was:<[]my text>
```

The simplest implementation would be:

```
public class InsertBeforeCssAction
    implements CssAction {
```

```
    /**
     * The associated {@link Css} block.
     */
    private final Css css;

    /**
     * Creates a new {@code InsertBeforeCssAction} instance.
     * @param css the {@link Css} block.
     */
    public InsertBeforeCssAction(Css css) {
        this.css = css;
    }

    @Override
    public String execute(String text) {
        String result = text;

        for (Property property : css.getProperties()) {
            if ("content".equals(property.getKey())) {
                result = property.getValue() + text;
                break;
            }
        }

        return result;
    }
}
```

That forces us to modify **CssActionFactory** to provide the original **Css** when creating **InsertBeforeCssAction** instances.

```
public class InsertBeforeCssAction
    implements CssAction {

    /**
     * The associated {@link Css} block.
     */
    private final Css css;

    /**
```

```
     * Creates a new {@code InsertBeforeCssAction} instance.
     * @param css the {@link Css} block.
     */
    public InsertBeforeCssAction(Css css) {
        this.css = css;
    }

    @Override
    public String execute(String text) {
        String result = text;

        for (Property property : css.getProperties()) {
            if ("content".equals(property.getKey())) {
                result = property.getValue() + text;
                break;
            }
        }

        return result;
    }
}
```

And that is enough to make the test pass. However, the code is not just right. At least, we are using the String literal "content" in several places. We should use a constant to ensure a typo cannot be the cause of a future bug.

```
public class Property<T> {
    /**
     * The "content" property.
     */
    public static final String CONTENT = "content";
    [..]
}
```

Although the temptation to refactor the method deeper is high, let's fight it and implement another easy **CssAction**: the **InsertAfterCssAction**.

```
public class CssActionFactoryTest {
    [..]
```

```
    @Test
    public void for_an_after_pseudo_selector_createAction_returns_InsertAfterCssAction(
        CssActionFactory factory = new CssActionFactory();
        Css css = new Css();
        css.addSelector(".rule::after");
        css.addProperty(new Property<String>("content", "value"));
        CssAction action = factory.createAction(css);
        Assert.assertNotNull(action);
        Assert.assertTrue(action instanceof InsertAfterCssAction);
    }
}
```

Next, we need to implement the bare minimum version of **InsertAfterCssAction** just to make the compiler happy at least.

```
public class InsertAfterCssAction
    implements CssAction {
    @Override
    public String execute(final String text) {
        return text;
    }
}
```

And we're back into the red zone. Our **CssActionFactory** is not aware of the new class yet.

```
public class CssActionFactory {
    public CssAction createAction(Css css) {
        CssAction result = null;

        for (String selector : css.getSelectors()) {
            if (selector.contains("::before")) {
                result = new InsertBeforeCssAction(css);
                break;
            } else if (selector.contains("::after")) {
                result = new InsertAfterCssAction();
                break;
            }
        }
```

```
        if (result == null) {
            result = new NullCssAction();
        }

        return result;
    }
}
```

It's not very elegant, but we are just trying to recover the green light. And it does that job.

Similarly to the **InsertBeforeCssActionTest**, we can anticipate what we want our new CSS action class to do.

```
public class InsertAfterCssActionTest {
    @Test
    public void execute_inserts_the_content_after_correctly() {
        Css css = new Css();
        css.addSelector(".rule::after");
        css.addProperty(new Property<String>("content", "<css-suffix"));
        CssActionFactory factory = new CssActionFactory();
        CssAction action = factory.createAction(css);
        Assert.assertNotNull(action);
        String newText = action.execute("my text");
        Assert.assertNotNull(newText);
        Assert.assertEquals("my text<css-suffix", newText);
    }
}
```

We check it fails:

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec <<< FAILURE!
execute_inserts_the_content_after_correctly(org.acmsl.javacss.css.InsertAfterCssAction
org.junit.ComparisonFailure: expected:<my text[<css-suffix]> but was:<my text[]>
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at org.acmsl.javacss.css.InsertAfterCssActionTest.execute_inserts_the_content_a
```

To fix it, we can follow the same steps as we did previously for **InsertBeforeCssAction**: add the **Css** class as an attribute, expect it as a parameter of the constructor,

```
public class InsertAfterCssAction
    implements CssAction {

    /**
     * The associated {@link Css} block.
     */
    private final Css css;

    /**
     * Creates a new {@code InsertAfterCssAction} instance.
     * @param css the {@link Css} block.
     */
    public InsertAfterCssAction(final Css css) {
        this.css = css;
    }

    @Override
    public String execute(final String text) {
        String result = text;

        for (Property property : css.getProperties()) {
            if (Property.CONTENT.equals(property.getKey())) {
                result = text + property.getValue();
                break;
            }
        }

        return result;
    }
}
```

and make sure **CssActionFactory** pass the **Css** instance when creating **InsertAfterCssAction** instances.

```
public class CssActionFactory {
    [..]
    public CssAction createAction(Css css) {
        [..]
        for (String selector : css.getSelectors()) {
            [..]
```

```
            } else if (selector.contains("::after")) {
                result = new InsertAfterCssAction(css);
                [..]
            }
        }
        [..]
    }
}
```

Now that our tests pass again, we are in need of a refactoring. In our current **CssAction** implementations we are searching the "content" property to retrieve the text to modify. We're duplicating code, so let's extract that logic to a common ancestor.

First, the base class.

```
public abstract class AbstractCssAction
    implements CssAction {

    /**
     * The associated {@link Css} block.
     */
    private final Css css;

    /**
     * Creates a new {@code InsertAfterCssAction} instance.
     * @param css the {@link Css} block.
     */
    protected AbstractCssAction(Css css) {
        this.css = css;
    }

    /**
     * Retrieves the {@link Css} block.
     * @return such information.
     */
    public Css getCss() {
        return this.css;
    }

    /**
```

```
     * Retrieves the content property from given {@link Css}.
     * @param css the CSS block.
     * @return the "content" property.
     */
    protected Property getContentProperty(Css css) {
        Property result = null;

        for (Property property : css.getProperties()) {
            if (Property.CONTENT.equals(property.getKey())) {
                result = property;
                break;
            }
        }

        return result;
    }
}
```

Now we can refactor **InsertBeforeCssAction**.

```
public class InsertBeforeCssAction
    extends AbstractCssAction {

    /**
     * Creates a new {@code InsertBeforeCssAction} instance.
     * @param css the {@link Css} block.
     */
    public InsertBeforeCssAction(final Css css) {
        super(css);
    }

    @Override
    public String execute(String text) {
        final String result;

        Property property = getContentProperty(getCss());

        if (property != null) {
            result = property.getValue() + text;
        } else {
```

```
            result = text;
        }

        return result;
    }
}
```

And similarly for **InsertAfterCssAction**.

```
public class InsertAfterCssAction
    extends AbstractCssAction {

    /**
     * Creates a new {@code InsertAfterCssAction} instance.
     * @param css the {@link Css} block.
     */
    public InsertAfterCssAction(final Css css) {
        super(css);
    }

    @Override
    public String execute(String text) {
        final String result;

        Property property = getContentProperty(getCss());

        if (property != null) {
            result = text + property.getValue();
        } else {
            result = text;
        }

        return result;
    }
}
```

The feedback of our current tests give us confidence we haven't broke anything. However, there's still some duplication in the *execute(String)* methods of both **InsertBeforeCssAction** and **InsertAfterCssAction**. Shouldn't we extract the shared logic to the base class?

Looking at the body of the methods, they're pretty obvious: retrieve the "content" property, and use it to generate the new text. Moving that logic up, and use a Strategy pattern, would not add real value: the redundancy we are removing makes the classes more complex somehow, and introduces some assumptions in the base class.

If we add other pseudo-classes that deal with text transformations (which are not currently part of CSS), then it would be worth reviewing.

1.4.7 Sixth test: Process the first AST

Now we can fit these new components together, so, given an AST and a CSS document, we can obtain another AST.