

JavaCSS

February 27, 2015

Contents

1	JavaCSS	1
1.1	Introduction	1
1.2	Project setup	2
1.3	Prototype	2
1.3.1	First test: Parsing an AST	2
1.3.2	Second test: Count methods	3
1.3.3	Third test: Retrieve the types the methods return	4
1.3.4	Fourth test: adding imports to the AST	5

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//*`.

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 programmatically. 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.

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 `"add_newASTnode()"`. The first step then is to build a `ParseTree` instance, so let's copy our first test `"can_parseaninterfacewithextendsandasinglemethod()"` into a `"buildAST()"` helper method for the tests.

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.

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

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

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: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.

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 by

represented by a tree of `typeNameContext`.

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.

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.

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.

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.