JavaCSS

March 2, 2015

Contents

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 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.

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.

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: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.

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.

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.

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><extraIdentifiers:{ e | .<e>}>; »

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.elementValuePair |<elementValuePairList(pair=p)>}><endif>) »

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 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 ";" "*" : 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: "\\"; }
```

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.

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.

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:

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.

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

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:

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.

1.4.3 Second test: Reading properties

Next, we want to define how to retrieve the properties associated to a selector. 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.

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

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.

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.

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

In our test, we are calling getText() on non-terminal nodes of our ParseTree. 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.