TOOLS

Introducing the JavaCC 21 parser generator for Java

This open source tool generates the code for embedding parsers into Java applications.

by Jonathan Revusky

August 6, 2021

Download a PDF of this article

This article introduces the JavaCC 21 parser generator by using a simple “Hello, World” example. JavaCC 21 is built on the JavaCC codebase that was open sourced by Sun Microsystems in mid-2003.

Not sure what a parser generator is or what you’d use one for? This fictitious conversation between Alan Zeichick (editor in chief of *Java Magazine*) and me (lead developer of the JavaCC 21 parser generator) will set the stage.

Zeichick: Brian Goetz, Oracle’s Java language architect, has said that every Java developer should have a tool like JavaCC in their toolkit. However, it seems that parser generators, such as yacc and Bison, and your own JavaCC 21, are rarely used by Java developers. Why?

Revusky: Several reasons. For one thing, parsers have developed a mystique about them. People think of them like, “This here is the realm of wizards, where mere mortals such as I dare not tread.”

Zeichick: Agreed; there is a huge intimidation factor. So, to rephrase the question, why are people so intimidated by parsers?

Revusky: To start with, the whole application space emerged from academia and is suffused with a kind of theoretical jargon. Consider the Wikipedia page on LL parsing. It begins with the following:

> In computer science, an LL parser (Left-to-right, Leftmost derivation) is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing Leftmost derivation of the sentence.

And it moves on to the following:

> “An LL parser is called LL-regular if it parses precisely the class of LL-regular languages[2][3][4] LLR grammars are a proper superset of LL(k) grammars for any k. For every LLR grammar there exists an LLR parser that parses the grammar in linear time.”

Zeichick: I can see why that’s intimidating. Is there a version of that page in plain English?

Revusky: That Wikipedia page has a version in 20 different languages, and I’m pretty certain that there is no plain-language explanation to be found in any of them. All the related pages are in the same vein, because that material emerges from an academic mindset.

Zeichick: How does JavaCC 21 help solve that problem?

Revusky: It’s not just that the whole field is so wrapped in a veil of jargon but also that the most common tools have, unfortunately, suffered from usability problems. The original JavaCC, written by a team led by Sreeni Viswanadha and Sriram Sankar at Sun Microsystems in the 1990s, was one of the most usable parser generator tools. However, JavaCC 21, which is now virtually a rewrite from the ground up of the original JavaCC, has had a key design goal of improving usability. I think this goal has largely been achieved.

Zeichick: That’s good, but why should the typical Java developer be interested in JavaCC 21, or any parser for that matter?

Revusky: Parsers are not only for people who want to design or implement an entire programming language, along the lines of Java or Python, say, although JavaCC 21 *does* include full, usable examples of grammar/parsers for both Java and Python out of the box.

Parsers are well suited for tasks of a much smaller scale that many working programmers have to deal with quite frequently. For example, defining (and implementing) some sort of little custom format—such as for configuration files or some custom data format.

These sorts of bread-and-butter tasks can be very laborious and error-prone when you have to code a parser by hand. But with a parser such as JavaCC 21 in your toolkit, such tasks are really a breeze. Basically, you write a high-level specification of your format and JavaCC 21 does just about everything for you. It generates the code to parse input in that format, as well as an API to manipulate the resulting AST.

Zeichick: What is AST?

Revusky: Oh, there, you see, I lapsed into some jargon myself. AST is an abstract syntax tree, analogous to how a DOM, a document object model, is an internal representation of an HTML or XML file.

JavaCC 21 takes your specified format and generates a program that can take input in that format and turn it into a similar kind of inverted tree structure. It’s really similar to a DOM processor. The point is that JavaCC 21 does most of the nitpicking work for you. I’ll show you with a “Hello, World” example.

Getting started: The prerequisites

Putting the conversation aside, I’ll now dive right into JavaCC 21. To follow along with the article, you should have

- A JDK (1.8 or higher) installed
- A UNIX or Linux shell in which to type commands

For Linux, UNIX, and macOS users, a suitable shell is included in the platform. Windows users can download and install the Git for Windows package. This gives you a command-line Git client for Windows, as well as Git for the Bash shell, along with other essential UNIX-like tools, including `grep` and `find`. With Git for Windows installed, you should be able to open a Bash terminal (called Git Bash) from the Windows Start menu.

Now, open the command shell and type the following to verify that you have JDK 1.8 or higher installed:

```
java -version
javac -version
```

Install the latest JavaCC 21 JAR file

You can download and install javacc-full.jar, a prebuilt JavaCC 21 JAR file, which has everything you need. Use the following commands:

```
mkdir hello
cd hello
curl -O https://javacc.com/download/javacc-full.ja
java -jar javacc-full.jar
```

Create the grammar file

The next step is to create a minimal “Hello, World” grammar file. In the empty `hello` directory, create a new text file called `HelloWorld.javacc`. If you use the nano text editor (which is installed on many systems and is included in Git for Windows), you could type

```
nano HelloWorld.javacc
```

And then populate the file with a single line.

```
HelloWorld : "Hello" "," "World")+  <EOF> ;
```

Save the file and exit back to the command line.

This `HelloWorld.javacc` is a minimal JavaCC 21 grammar that describes a little language. It’s not a very useful or interesting language, but it’s a language all the same. It accepts the following strings, and so on:

```
Hello,World
 Hello,World,World,
 Hello,World,World,World
```

Thus, the parser I will build expects input comprising `Hello` followed by a comma and the word `World`. The `,World` can be repeated an arbitrary number of times, but it must occur at least once. That, by the way, is what the `+` after (`"," "World"`) means: one or more repetitions.

Generate the parser code and add main()

You can generate the parser source for the “Hello, World” language with the following shell command:

```
java -jar javacc-full.jar HelloWorld.javacc
```

This should mean you have many Java source files in the `hello` directory. Compile the code with the following command:

```
javac *.java
```

Oh, wait; I forgot the `main()` method, which is required as an entry point. Open the grammar file again and add the following lines to either the top or bottom of the file:

```
nano HelloWorld.javacc

INJECT HelloWorldParser : {
    static public void main(String[] args) {
       if (args.length == 0) {
           System.out.println("Usage: java HelloWo
           System.exit(-1);
       }
       String input = "";
       for (String arg : args) input += arg;
       HelloWorldParser parser = new HelloWorldPar
       parser.HelloWorld();
       parser.rootNode().dump();
    }
}
```

Save the file and close the editor to return to the command line. Generate the source code, as follows:

```
java -jar javacc.jar HelloWorld.javacc
```

The `INJECT` statement above injected the necessary `main()` method into the `HelloWorldParser.java` file that was generated. And now recompile, as follows:

```
javac *.java
```

And finally, test the application.

```
java HelloWorldParser Hello, World
```

The application read in `Hello,World` as input, parsed the input according to the mini-language you defined, built a parse tree, and then dumped the tree in a human-readable format, providing the following output:

```
HelloWorld
Hello
,
World
EOF
```

How it works

The interesting lines in the `main()` method are the last three.

- Those lines created an instance of `HelloWorldParser` and passed it the input as a parameter, in this case `Hello,World`.
- The `parser.HelloWorld();` line had the parser match the input to the HelloWorld production or grammar rule. (Note that HelloWorld is, in fact, the only grammar rule at the moment.)
- The `parser.rootNode().dump()` line used a utility method to dump a representation of the tree.

In this case, the tree has a root node of type `HelloWorld` and it has four child nodes that are the tokens: `Hello` , `World`, and `EOF`. The `EOF` is generated automatically when the program reaches the end of the input.

The matter of spaces

You may have noticed that the command-line phrase that launched the program was

```
java HelloWorldParser Hello, World
```

with a space after the comma. However, the parser you wrote received the input `Hello,World` *without* the space. That's because of how the command line's own parser works. The command line received two arguments: `Hello,` and `World`. The test harness concatenated them together to produce the spaceless input `Hello,World`.

Because the test grammar you wrote never specified how to handle spaces, the input string `Hello, World` is not actually valid input to the parser. You can see that by trying the following:

```
java HelloWorldParser "Hello, World"
```

The line above passes in a single command-line argument, `Hello, World` (with the space), and, as you can see, the parser does not know what to do with the space. It gets scanned as a token type called `INVALID`, which represents invalid input.

I suggest that you play around a bit. You could try the test harness with other valid input.

```
java HelloWorldParser Hello,World,World,World,Worl
```

And you could also try other invalid inputs to see what happens.

```
java HelloWorldParser Hello,
 java HelloWorldParser Hello,World,World,
 java HelloWorldParser Bye, World
```

Next steps

If you want to build the `javacc-full.jar` file from source, you need the Apache Ant build library. After installing Ant, use the following commands:

```
git https://github.com/javacc21/javacc21.git
cd javacc21
ant full-jar
```

Important note: This tutorial applies only to JavaCC 21, which is built on the original JavaCC tool, which was originally open sourced by Sun Microsystems in 2003. What are the differences? One is that the `INJECT` statement does not exist in the original JavaCC. Also, the automatic building of a parse tree can be done with JavaCC but involves a complex build process using a separate preprocessing utility named JJTree.

In short, a similar sort of “Hello, World” introductory example with the original JavaCC is possible, but it would be more cumbersome and verbose.

To learn more about JavaCC 21 development, follow the blog.

Dig deeper

- jsoup HTML parsing library
- Designing and implementing a library
- JavaCC 21
- JavaCC (the original)

Jonathan Revusky

Jonathan Revusky is an independent software developer located in Spain, the project lead of the FreeMarker 2.0 template engine, and lead developer of the JavaCC 21 project. His work on JavaCC emerged when he became very interested in the myriad possibilities that a template engine such as FreeMarker offers for source code generation.

Share this Page