

JavaCC, parse trees, and the XQuery grammar, Part 1

Using BNF and JavaCC to build custom parsers

[Howard Katz \(howardk@fatdog.com\)](#), Proprietor, Fatdog Software

Summary: After a brief discussion of grammars, parsers, and BNF, this article introduces JavaCC, a popular parser generator tool. You'll develop sample code that uses JavaCC to build a custom parser, starting from a BNF description of the grammar. Part 2 goes on to show how to use an auxiliary tool, JJTree, to build a parse tree representation of the same parse and how to walk that tree at runtime to recover its state information. The article concludes by developing sample code to build and walk a parse tree that you'll generate for a small portion of the XQuery grammar.

Date: 01 Dec 2002

Level: Intermediate

Also available in: [Japanese](#)

Activity: 17861 views

Comments: 0 ([View](#) | [Add comment](#) - Sign in)



Average rating (35 votes)

[Rate this article](#)

You don't need anything as complicated as an automated parser generator to do most simple, everyday parsing tasks. A programming exercise as straightforward as teasing apart the pieces of a CSV (Comma-Separated-Value) file, for example, requires an understanding of the structure of the file and maybe an understanding of how to use a Java `StringTokenizer`. Otherwise, a CSV exercise requires very little knowledge of parsing theory as such or the need to apply automated tools to the task.

Once a language and the grammar that formally describes it becomes complex, however, the number of valid expressions in the language grows rapidly. It becomes more and more difficult to be able to hand-roll the code you'd need to parse an arbitrary expression into its constituent pieces (which is more or less a very concise definition of parsing). Automated parser generators alleviate that difficulty. Other programmers might also be able to use the parser that you've generated for their own purposes.

Starting from the BNF

The grammars for complex languages are commonly described using BNF (Backus-Naur Form) notation or its close relative, EBNF (Extended BNF). Automated tools can use those descriptions (I'll use the generic term, *BNF*, to refer to both variations) or a close approximation thereto to generate your parsing code for you. This article describes one such parser-generator tool called JavaCC. I'll explore JavaCC basics briefly and end up spending some time with one of its auxiliary tools, JJTree, without getting sidetracked by too much theory along the way. This article attempts to demonstrate my belief that you don't need to know a lot about formal parsing theory to be able to parse!

Why JavaCC? A couple of reasons: I have a strong interest in XQuery, and JavaCC is used by the W3C's XML Query working group to build and test versions of the XQuery grammar, as well as the XPath grammar it shares with the XSL group. I also use JavaCC to provide the query-parsing code in XQEngine, my own open-source XQuery implementation (see [Resources](#)).

Last but not least, the price is entirely right: While JavaCC isn't open-source, it is 100% free. (See [Resources](#) for information on obtaining JavaCC).

Parsing 101

Before I get to some actual XQuery grammar, let's start with a much simpler BNF that describes a language consisting of only two arithmetic operators that operate only on integers. I'll call this language SimpleLang:

```
simpleLang    ::= integerLiteral ( ( "+" | "-" ) integerLiteral )?
integerLiteral ::= [ 0-9 ]+
```

Each of the rules in this grammar is a **production**, in which the term on the lefthand side -- the name of the production -- is described in terms of the other productions in the grammar. The topmost production, `simpleLang`, states that a valid (or legal) expression in this language consists of an integer literal, followed optionally by either a plus (+) or a minus (-) sign followed by another integer literal. In terms of this grammar, the single integer "42" is valid, as is the expression "42 + 1". The second production describes in regex-like fashion more specifically what an integer literal looks like: a contiguous sequence of one or more numeric digits.

The grammar describes the abstract relationship that exists between the two productions, `simpleLang` and `integerLiteral`. It also details in concrete terms the composition of the three **tokens** (a plus sign, a minus sign, and an integer) that the parser expects to encounter while scanning through the input stream. Not surprisingly, the part of the parser that's responsible for that task is called the **scanner** or **tokenizer**. `simpleLang` is an example of a *non-terminal* symbol in this language, one that references other productions; the rule for `integerLiteral` on the other hand describes a *terminal*: a symbol that can't be decomposed further into other productions.

If the parser finds anything *other* than these three tokens during its scan, the expression it's scanning is considered invalid. One of the parser's main jobs is to determine the validity of any expression you pass it and to let you know. Its second job, once an expression has been deemed valid, is to break the input stream into its component pieces and deliver those to you in some useful fashion.

From BNF to JavaCC

Let's look at how to use JavaCC to implement this grammar. JavaCC works with something called a .jj file. The description of the grammar in this file is written in a notation that's very similar to BNF, so that it's generally fairly easy to translate from one to the other. (The notation has a syntax of its own, making it expressible in JavaCC.)

The main difference between a JavaCC .jj file grammar and standard BNF is that, with the JavaCC version, you can embed actions in the grammar. These actions execute once those portions of the grammar have been successfully traversed. The actions are Java statements that become part of the Java source code for the parser that's emitted as part of the parser generation process.

(Note that other than a single Java `println()` statement, [Listing 1](#) doesn't contain the embedded Java code you'd need to actually evaluate an expression in this language. I'll explore that in more detail once you've looked at JJTree and its parse tree representations.)

Listing 1. The entire .jj script that encodes the SimpleLang grammar

```
PARSER_BEGIN( Parser_1 )
package examples.example_1;
public class Parser_1 {}
PARSER_END( Parser_1 )
```

```

void simpleLang() : {}           { integerLiteral()
                                   ( ( "+" | "-" ) integerLiteral() )? <EOF> }
void integerLiteral() : {Token t;} { t=<INT>
                                   { System.out.println("integer = "+t.image); }}

SKIP      : { " " | "\t" | "\n" | "\r" }
TOKEN     : { < INT : ( ["0" - "9"] )+ > }

```

Note the following things about this file:

- The `PARSER_BEGIN` and `PARSER_END` directives name the Java parser to be generated (`Parser_1.java`), and provide a place to insert Java statements into that class. In this case you're placing a Java `package` statement at the top of the file. The `package` statement is also placed at the top of the Java helper files that are emitted as part of the generation process (see [The JavaCC compilation process](#)). This is also a good place to declare instance variables that will be referenced by Java statements in your productions although I'm not doing that in this example. And if you like, you can even insert a Java `main()` procedure at this point and use it to build a standalone application to fire up and test the parser you're generating.
- The JavaCC syntax has a procedural look, with productions looking very much like method calls. This is no accident. The Java source that is emitted when JavaCC compiles this script contains methods that have identical names to these productions; these methods are executed at runtime in the order that they're called in the `.jj` script. I'll show you how that works in [Walking the parsing code](#). (The term "compile" here is also no accident -- parser generators are also commonly referred to as "compiler compilers".)
- Braces (`{` and `}`) delineate the bodies of the productions, and escape any Java actions you're embedding. Note the braces enclosing the `System.out.println()` statement in the `integerLiteral` rule. This action is emitted as part of the method `Parser_1.integerLiteral()`. It is executed every time the parser encounters an integer.
- The `SKIP` statement at the end of the file says that white space (blanks, tabs, carriage returns, and line feeds) can occur between tokens and is to be ignored.
- The `TOKEN` statement contains a regex-like expression that describes what integer tokens look like. References to these tokens in the preceding productions are enclosed in angle brackets.
- The second production, for `integerLiteral()`, declares a local variable `t` of type `Token`, a built-in class to JavaCC. This rule *fires* when an integer is encountered in the input stream, and the value of that integer (as text) is assigned to the instance variable, `t.image`. The other `Token` field, `t.kind`, is assigned an enum indicating that this particular token is an integer and not another kind of token that's known to the parser. Finally, the Java `System.out.println()` code that's generated in the parser is able to get at the internals of that token at parse time, using `t.image` to access and print out its textual value.

Walking the parser code

Let's look very briefly at the internals of the parser you've produced. Knowing a bit about the methods generated by a particular `.jj` grammar and their order of execution at runtime is useful for a couple of reasons:

- Sometimes (particularly when you're first starting out), the parser seems to return different results than what you think the `.jj` file instructs it to. You can step through the emitted parser code at runtime to see what it's actually doing and adjust the grammar file accordingly. I still do this frequently.
- If you know the order in which productions/methods are executed, you'll have a better understanding of where to embed Java actions in the script to obtain particular results. I'll come back to this in more detail when I talk about the JJTree tool and parse tree representations in [Part 2](#).

While it's beyond the scope of this article to delve too deeply into the details of the parser that's been produced, [Listing 2](#) shows the code that is generated for the method `Parser_1.integerLiteral()`. This might give you some idea of what the final code will look like. Note in particular the last statement in the method: `System.out.println("integer = "+t.image)`. This statement started life as the Java action that you embedded in the `.jj` script.

Listing 2. A generated method in Parser_1

```
static final public void integerLiteral() throws ParseException {
    Token t;
    t = jj_consume_token(INT);
    System.out.println( "integer = "+t.image);
}
```

Here's a high-level, blow-by-blow description of what this parser is doing:

1. The topmost method, `simpleLang()`, calls `integerLiteral()`.
2. `integerLiteral()` expects to immediately encounter an integer in the input stream, else the expression would be invalid. To verify this, it calls the tokenizer (`Tokenizer.java`) to return the next token in the input stream. The tokenizer moves through the stream, inspecting characters one at a time until it encounters either an integer or the end of file. If it's the former, it packages up the value in an `<INT>` token; if the latter, as an `<EOF>`; and returns the token to `integerLiteral()` for further processing. If the tokenizer encounters neither token, it returns a lexical error.
3. If the token returned by the tokenizer isn't an integer token or an `<EOF>`, `integerLiteral()` throws a `ParseException` and parsing is complete.
4. If it's an integer token, the expression is still potentially valid, and `integerLiteral()` calls the tokenizer once again to return the next token. If it returns an `<EOF>`, the entire expression consists of a single integer, is valid, and the parser returns control to the calling application.
5. If the tokenizer returns either a plus or a minus token, the expression is still valid, and `integerLiteral()` calls the tokenizer one last time, looking for another integer. If it encounters one, the expression is valid and the parser is done. If the next token isn't an integer, the parser throws an exception.

Note that if the parser fails, it throws either a `ParseException` or a `TokenMgrError`. Either one indicates that your expression was invalid.

A key point here is that any Java actions embedded in these two productions are only executed if the parser successfully traverses the portion of the production in which they're embedded. If this parser is passed the expression "42 + 1", the statement `integer = 42` is printed to the console, followed by `integer = 1`. If it's run against the invalid expression "42 + abc", the message `integer = 42` is emitted, followed by the catch-block message `a Token Manager error!`. In the latter case, the parser has only successfully traversed the first `integerLiteral()` term in the `simpleLang` production, and not its second:

```
void simpleLang() : {} { integerLiteral() ( ("+" | "-") integerLiteral() )? <EOF> }
```

In other words, the second `integerLiteral()` method has not been executed because the expected integer token wasn't encountered.

The JavaCC compilation process

When you run JavaCC against the `.jj` file, it generates a number of Java source files. One is the primary parsing code, `Parser_1.java`, which you'll invoke from your application when you have an expression to parse. JavaCC also creates six other auxiliary files that are used by the parser. In total, JavaCC generates the following seven Java files. The first three are specific to this particular grammar; the final four are generic helpers that are always generated no matter what the grammar looks like.

- `Parser_1.java`
- `Parser_1Constants.java`
- `Parser_1TokenManager.java`
- `ParseException.java`
- `SimpleCharStream.java`

- Token.java
- TokenMgrError.java

Once JavaCC has generated these seven Java sources, they can be compiled and linked into your Java application. You can then call the new parser from your application code, passing it an expression to evaluate. Here's a sample application that instantiates your parser and feeds it an expression that you hardwired in at the top of the app.

Listing 3: Invoking your first parser

```
package examples.example_1;

import examples.example_1.Parser_1;
import examples.example_1.ParseException;
import java.io.StringReader;

public class Example_1
{
    static String expression = "1 + 42";

    public static void main( String[] args )
    //-----
    {
        new Example_1().parse( expression );
    }

    void parse( String expression )
    //-----
    {
        Parser_1 parser = new Parser_1( new StringReader( expression ) );
        try
        {
            parser.simpleLang();
        }
        catch( ParseException pe ) {
            System.out.println( "not a valid expression" );
        }
        catch( TokenMgrError e ) {
            System.out.println( "a Token Manager error!" );
        }
    }
}
```

What's noteworthy here?

1. To invoke the parsing code in `Parser_1`, you invoke the method `simpleLang()` in that class. The order of the productions in a `.jj` file is generally irrelevant except in this case, where the name of the topmost production in the grammar is used to invoke the parser.
2. If the expression you're passing in to the parser code can't be legally constructed according to the grammar, either a `ParseException` or a `LexicalError` is thrown.
3. If the expression is valid, any Java actions embedded in the parts of the grammar that have been successfully traversed are executed, as described at the end of [Walking the parser code](#).

Conclusion

Part 2 of this article continues where this one leaves off. You'll start with similar example code and learn how to use JavaCC's companion tool, JJTree, to create a parser that builds a parse-tree representation of the parse at runtime, rather than executing actions that you've embedded in the `.jj` script. This has a number of advantages, as you'll see.

Resources

- [Participate in the discussion forum.](#)
- Continue with [Part 2](#) to learn how to use JavaCC's companion tool, JJTree, to create a parser that builds a parse-tree representation of the parse at runtime.
- Want to find out more about BNF? Check out [Wikipedia.org](#).
- Check out the free (though not open source) [distribution for JavaCC and JJTree](#).
- Find out more about the W3C's XQuery and XPath specifications at the [XML Query home page](#).
- [XQEngine](#) is the author's Java-based open-source implementation of an XQuery engine.
- Find more information on the technologies covered in this article at the developerWorks [XML](#) and [Java technology](#) zones.
- [IBM trial software for product evaluation](#): Build your next project with trial software available for download directly from developerWorks, including application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- [IBM XML certification](#): Find out how you can become an IBM-Certified Developer in XML and related technologies.

About the author

Howard Katz lives in Vancouver, Canada, where he is the sole proprietor of Fatdog Software, a company that specializes in software for searching XML documents. He's been an active programmer for nearly 35 years (with time off for good behavior) and is a long-time contributor of technical articles to the computer trade press. Howard cohosts the Vancouver XML Developer's Association and is the editor of an upcoming book from Addison Wesley, *The Experts on XQuery*, a compendium of technical perspectives on XQuery by members of the W3C's Query working group. He and his wife do ocean kayaking in the summer and backcountry skiing in the winter. You can contact Howard at howardk@fatdog.com.

[Close \[x\]](#)

developerWorks: Sign in

IBM ID:

[Need an IBM ID?](#)

[Forgot your IBM ID?](#)

Password:

[Forgot your password?](#)

[Change your password](#)

☐ Keep me signed in.

By clicking **Submit**, you agree to the [developerWorks terms of use](#).

The first time you sign into developerWorks, a **profile** is created for you. **Select information in your developerWorks profile is displayed to the public, but you may edit the information at any time.** Your first name, last name (unless you choose to hide them), and display name will accompany the content that you post.

All information submitted is secure.

[Close \[x\]](#)

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.























Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the [developerWorks terms of use](#).

All information submitted is secure.

 Average rating (35 votes)

- ☐ 1 star      1 star
- ☐ 2 stars      2 stars
- ☐ 3 stars      3 stars
- ☐ 4 stars      4 stars
- ☐ 5 stars      5 stars

Add comment:

[Sign in](#) or [register](#) to leave a comment.

Note: HTML elements are not supported within comments.

☐ Notify me when a comment is added1000 characters left

Be the first to add a comment

[Print this page](#)

[Share this page](#)

[Follow developerWorks](#)

[About](#)

[Feeds and apps](#)

[Report abuse](#)

[Faculty](#)

[Help](#)

[Newsletters](#)

[Terms of use](#)

[Students](#)

Contact us

Submit content

IBM privacy

IBM accessibility

Business Partners
