



Building and walking a custom parse tree using JJTree
Level: Intermediate

[Howard Katz \(howardk@fatdog.com\)](#)

Proprietor, Fatdog Software
December 2002

Part 1 of this article took an introductory look at grammars, parsers, and BNF. It then introduced JavaCC, a popular parser generator. Part 2 shows you how to modify the sample code in Part 1 so that you can use an additional tool, JJTree, to build a parse tree representation of the same parse. You'll explore the advantages of this approach and look at how to write Java code to walk the parse tree at runtime in order to recover its state information, and evaluate the expression being parsed. The article concludes by showing you how to develop a generalizable routine for walking and evaluating a parse tree that you'll generate from a small portion of the XQuery grammar.

Using the JavaCC parser generator has one major downside: Much or most of your client-side Java code needs to be embedded in the .jj grammar script that encodes your BNF (Backus-Naur Form). This means you lose many of the advantages that a proper Java IDE can provide you during the development cycle.

Enter JJTree, JavaCC's companion tool. JJTree is set up to emit a parser whose main job at runtime is not to execute embedded Java actions, but to build an independent parse-tree representation of the expression that's being parsed. This lets you capture the state of the parse session in a single tree that's easy to walk and interrogate at runtime, independent of the parsing code that produced it. Working with a parse tree representation also makes debugging easy and speeds development time. JJTree is distributed as part of the JavaCC distribution (see [Resources](#)).

I'll note before we go any further that the terms **parse tree** and **abstract syntax tree** (or AST) describe very similar grammatical structures. Strictly speaking, what I refer to as a parse tree in the following, language theoreticians would more precisely call an AST.

To work with JJTree, you need to be able to:

1. Create the .jjt script that JJTree takes as input
2. Write client-side code to walk and evaluate the parse tree that's produced at runtime

This article shows you how to do both. It doesn't cover everything, but it'll certainly get you started.

JJTree basics

JJTree is a preprocessor, and generating a parser for a particular BNF is an easy two-step process:

1. Run JJTree against a so-called .jjt file; this emits an intermediate .jj file
2. Compile that with JavaCC (this process was covered in [Part 1](#))

Fortunately, the structure of a .jjt file is a minor extension of the .jj format that I showed you in Part 1. The primary difference is that JJTree adds a new syntactic **node-constructor** construct which lets you specify where and under what conditions parse-tree nodes are to be generated during the parse. In other words, this governs the shape and content of the parse tree that's constructed by the parser.

Listing 1 shows a simple JavaCC .jj script that's similar to the ones you saw in Part 1. For simplicity, I'm showing productions only.

Contents:

- [JJTree basics](#)
- [Step-by-step through a JJTree grammar](#)
- [Working with the parse tree](#)
- [Aids to navigation](#)
- [Saving and restoring state](#)
- [Putting it all together: A BNF snippet for XQuery](#)
- [Walking the parse tree client-side](#)
- [Conclusion](#)
- [Resources](#)
- [About the author](#)
- [Rate this article](#)

Related content:

- [JavaCC, parse trees, and the XQuery grammar, Part](#)
- [An introduction to XQuery](#)
- [Subscribe to the developerWorks newsletter](#)

Also in the XML zone:

- [Tutorials](#)
- [Tools and products](#)
- [Code and components](#)
- [Articles](#)

Also in the Java zone:

- [Tutorials](#)
- [Tools and products](#)
- [Code and components](#)
- [Articles](#)

Listing 1. A JavaCC grammar for simpleLang

```
void simpleLang()      : {}      { addExpr() <EOF> }
void addExpr()        : {}      { integerLiteral() ( "+" integerLiteral() )? }
void integerLiteral() : {}      { <INT> }

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

This grammar states that a legal expression in this language consists of either:

1. a single integer literal, or
2. an integer literal, followed by a plus sign, followed by another integer literal.

The corresponding JJTree .jjt script (again, slightly abbreviated) might look something like the following:

Listing 2. The JJTree equivalent to the JavaCC grammar in Listing 1

```
SimpleNode simpleLang() : #Root      {} { addExpr() <EOF> { return jjtThis; }}
void addExpr()          :             {} { integerLiteral()
                                       ( "+" integerLiteral() #Add(2) )? }
void integerLiteral()   : #IntLiteral {} { <INT> }

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

This script adds several new syntactic features to those you've already seen. Let's just touch on the highlights for the moment. I'll fill in the details later.

Step-by-step through a JJTree grammar

Note first that JavaCC's procedural syntax for the topmost `simpleLang()` production now specifies a return type:

`SimpleNode`. This, together with the embedded Java action, `return jjtThis` (a bit of JJTree hand-waving), specifies that calling the parser's `simpleLang()` method from your application code returns the root of the parse tree, which is then available for treewalking.

The parser invocation that looked like this in JavaCC:

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
parser.simpleLang();
```

now looks like this:

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
SimpleNode rootNode = parser.simpleLang();
```

Note that the root node you're grabbing isn't simply of type `SimpleNode`. It's really of type `Root`, as indicated by the `#Root` directive in [Listing 2](#) (although you don't use that fact in your calling code above). `Root` is a `SimpleNode` descendant, as is every node constructed by a JJTree-generated parser. I'll show you several of `SimpleNode`'s built-in methods below.

The `#Add(2)` construct in the `addExpr()` production differs from the `#Root` directive above it in several ways:

- It's parameterized. The tree builder uses a node stack during tree construction; the default behavior for a node constructor with no parameters is to place itself at the top of the parse tree that's under construction, popping all the nodes off the node

stack that were created in the same *node scope* and hoisting itself into position as the parent of those nodes. The argument 2 tells the new parent (an `Add` node in this case) to adopt exactly *two* children, the two `IntLiteral` subnodes described in the [following bullet](#). The JJTree documentation describes this process in more detail. Walking through the parse tree at runtime with a good debugger is another invaluable adjunct to understanding how tree building works in JJTree.

- Of equal importance, the placement of the `#Root` directive outside the body of its production means that a `Root` node is generated *every time* this production is traversed (which admittedly only happens once in this particular case). However, the placement of the `#Add(2)` directive inside an optional "zero or one" term means that an `Add` node is only generated *conditionally* if the optional clause containing it is traversed during the parse -- in other words, if this production represents a true addition operation. When that happens, `integerLiteral()` is traversed twice, contributing a single `IntLiteral` node to the tree on each invocation. Both `IntLiteral` nodes become children of the `Add` node that invokes them. However, if the expression being parsed is a single integer, then the resulting `IntLiteral` node becomes a child of `Root` directly.

Pictures are worth a kiloword (to bring a old saw up to date). Here's a graphic representation of the two types of parse trees generated by the above grammar:

Figure 1: Parse tree for a single integer expression

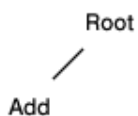
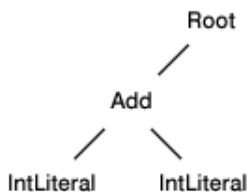


Figure 2: Parse tree for an addition operation



Let's look at the class hierarchy for `SimpleNode` in a bit more detail.

Working with the parse tree

Every node you declare in a `.jjt` script instructs the parser to generate a subclass of a JJTree `SimpleNode`. `SimpleNode`, in turn, implements a Java interface called `Node`. The source files for both of these classes are generated automatically by every JJTree script, along with a custom `.jj` file. [Listing 1](#) shows the current example of a custom `.jj` file. In the current example, JJTree also emits source files for your own classes -- `Root`, `Add`, and `IntLiteral` -- as well as several additional helper classes not looked at here.

All `SimpleNode` subclasses inherit useful behavior. The `SimpleNode` method `dump()` is a case in point. It also serves as an example of my earlier contention that using parse trees makes debugging easier and consequently speeds development time. The following three-line snippet of client-side code instantiates the parser, invokes it, grabs the parse tree that's returned, and dumps a straightforward, textual representation of the tree to the console:

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
SimpleNode rootNode = parser.simpleLang();
rootNode.dump();
```

The debug output for the tree in [Figure 2](#) would be:

```
Root
  Add
    IntLiteral
    IntLiteral
```

Aids to navigation

Another useful built-in `SimpleNode` method is `jjtGetChild(int)`. When you're navigating downward through the parse tree on the client side and encounter an `Add` node, you'll want to grab its `IntLiteral` children, extract the integer values they represent, and add them together -- that, after all, is the purpose of the exercise. Assuming that `addNode`, shown in the next bit of code, is a variable representing the `Add`-type node we're interested in, we can access `addNode`'s two children. (`lhs` and `rhs` are commonly used abbreviations for *left-hand side* and *right-hand side*, respectively.)

```
SimpleNode lhs = addNode.jjtGetChild( 0 );
SimpleNode rhs = addNode.jjtGetChild( 1 );
```

With everything you've done so far, however, you still don't have quite enough information to calculate the results of the arithmetic operation represented by this parse tree. Your current script has omitted one important detail: The two `IntLiteral` nodes in the tree don't actually contain the integers they purport to represent. That's because you didn't save their values into the tree when the tokenizer encountered them in the input stream; you need to modify your `integerLiteral()` production to do that. You also need to add a few simple accessor methods to `SimpleNode`.

Saving and restoring state

To store the value of scanned tokens into the appropriate nodes, add the following modifications to `SimpleNode`:

```
public class SimpleNode extends Node
{
    String m_text;

    public void    setText( String text ) { m_text = text; }
    public String getText()              { return m_text; }
    ...
}
```

Change the following production in your `JJTree` script:

```
void integerLiteral() : #IntLiteral {} <INT> }
```

to this:

```
void integerLiteral() : #IntLiteral { Token t; }
                          { t=<INT> { jjtThis.setText( t.image ); } }
```

This production grabs the raw text value of the integer it's just encountered from `t.image` and uses your `setText()` setter to store that string in the current node. The client-side `eval()` code in [Listing 5](#) shows how to use the corresponding `getText()` getter.

You can easily modify `SimpleNode.dump()` to emit the value of `m_text` for any node that's stored it during a parse -- I'll leave that as a proverbial exercise for you. This gives an even better visualization of what the parse tree looks like for debugging purposes. For example, if you parsed "42 + 1", a slightly modified `dump()` routine can produce the following useful output:

```
Root
  Add
    IntLiteral[42]
    IntLiteral[1]
```

Putting it all together: A BNF snippet for XQuery

Let's put it all together and conclude by looking at a snippet of an actual, real-world grammar. I'll show you a very small subset of the BNF for XQuery, the W3C's specification for a query language for XML. (See [Resources](#).) Most of what I'm saying here applies to XPath as well, since the two share much of the same grammar in common. I'll also look briefly at the issue of operator precedence and generalize the treewalking code into a full-fledged recursive routine that's able to handle arbitrarily complex parse trees.

[Listing 3](#) shows the piece of the XQuery grammar that you'll work with. This snippet of BNF is from the November 15, 2002 working draft:

Listing 3: A portion of the XQuery grammar

```
[21] Query           ::= QueryProlog QueryBody
    ...
[23] QueryBody      ::= ExprSequence?
[24] ExprSequence   ::= Expr ( "," Expr ) *
[25] Expr           ::= OrExpr
    ...
[35] RangeExpr      ::= AdditiveExpr ( "to" AdditiveExpr ) *
[36] AdditiveExpr   ::= MultiplicativeExpr ( "+" | "-" ) MultiplicativeExpr *
[37] MultiplicativeExpr ::= UnionExpr ( "*" | "div" | "idiv" | "mod" ) UnaryExpr *
    ...
```

You're going to build just enough of a JJTree grammar script to handle the +, -, *, and div operators in productions [36] and [37], again with the simplifying assumption that the only datatype this grammar knows about is integers. This sample grammar is *extremely* small and does no justice to the richness of expression and data types supported by XQuery. However, it should give you a good head start in using JavaCC and JJTree, if you're going to build parsers for larger, more complex grammars.

[Listing 4](#) shows the .jjt script. Note the options { } block at the top of the file. These options (there are a number of other switches available as well) specify, among other things, that tree building in this case operates in *multi* mode, where node constructors are used to explicitly name the types of generated nodes. The alternative (not explored here) is for productions to contribute only SimpleNode nodes to the parse tree and not subclasses thereof. That option is useful if you want to avoid proliferating node classes.

Note also that the original XQuery BNF frequently multiplexes multiple operators into the same production. I've demultiplexed these into separate productions in the JJTree script in [Listing 4](#), since this makes the code on the client side more straightforward. To multiplex, simply store away the scanned operator's value, exactly as you did for integers.

Listing 4: The JJTree script for the XQuery grammar in Listing 3

```
options {
    MULTI=true;
    NODE_DEFAULT_VOID=true;
    NODE_PREFIX=" ";
}

PARSER_BEGIN( XQueryParser )
package examples.example_2;
public class XQueryParser{}
PARSER_END( XQueryParser )

SimpleNode query()      #Root      : {} { additiveExpr() <EOF> { return jjtThis; } }
void additiveExpr()    : {} { subtractiveExpr()
    ( "+" subtractiveExpr() #Add(2) ) * }
void subtractiveExpr() : {} { multiplicativeExpr()
    ( "-" multiplicativeExpr() #Subtract(2) ) * }
void multiplicativeExpr() : {} { divExpr() ( "*" divExpr() #Mult(2) ) * }
void divExpr()         : {} { integerLiteral()
    ( "div" integerLiteral() #Div(2) ) * }
void integerLiteral() #IntLiteral : { Token t; }
    { t=<INT> { jjtThis.setText(t.image); } }
```

```

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

```

This .jgt file introduces several new features. For one, the arithmetic productions in this grammar are now *iterative*: Their optional second terms are expressed using a * (zero or more) occurrence indicator, as opposed to the ? (zero or one) notation in [Listing 2](#). The parser emitted by this script can parse arbitrarily long expressions such as "1 + 2 * 3 div 4 + 5".

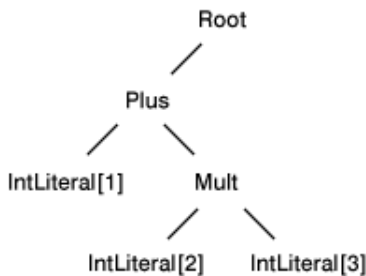
Implementing precedence

This grammar also knows about *operator precedence*. You would expect multiplication, for example, to have a higher precedence than addition. In practical terms, this means an expression like "1 + 2 * 3" would be evaluated as "1 + (2 * 3)" and not "(1 + 2) * 3".

Precedence is achieved using the cascading style, where each production calls the higher-precedent production immediately following it. That, as well as the placement and format of the node constructors, guarantees that the parse tree is generated in the proper conformation so that treewalking does the right thing. It might be easier to grasp this with some visual aids.

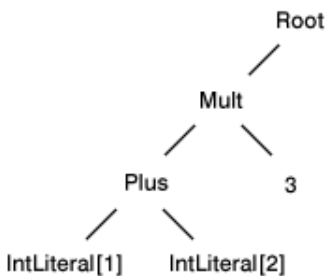
[Figure 3](#) shows the parse tree generated by this grammar that correctly enables you to evaluate "1 + 2 * 3" as "1 + (2 * 3)". Note that the `Mult` operator binds its terms tighter than does `Plus`, which is exactly what you want:

Figure 3. A properly formed tree



And, [Figure 4](#) shows a tree (which this grammar *doesn't* generate) indicating that you (incorrectly) wanted to evaluate this as "(1 + 2) * 3".

Figure 4. An incorrectly formed tree



Walking the parse tree client-side

I conclude as promised with a listing for the client-side code that invokes this parser and walks the parse tree it generates, using a simple but powerful, recursive `eval()` function to do the right thing with each node it encounters during the treewalk. Comments in [Listing 5](#) provide additional detail on internal JJTree workings.

Listing 5. An easily generalizable `eval()` routine

```

// return the arithmetic result of evaluating 'query'
public int parse( String query )
//-----
{
    SimpleNode root = null;

    // instantiate the parser
    XQueryParser parser = new XQueryParser( new StringReader( query ) );

    try
    {
        // invoke it via its topmost production
        // and get a parse tree back

        root = parser.query();
        root.dump("");
    }
    catch( ParseException pe ) {
        System.out.println( "parse(): an invalid expression!" );
    }
    catch( TokenMgrError e ) {
        System.out.println( "a Token Manager error!" );
    }

    // the topmost root node is just a placeholder; ignore it.

    return eval( (SimpleNode) root.jjtGetChild(0) );
}

int eval( SimpleNode node )
//-----
{
    // each node contains an id field identifying its type.
    // we switch on these. we could use instanceof, but that's less efficient

    // enum values such as JJTINTLITERAL come from the interface file
    // SimpleParserTreeConstants, which SimpleParser implements.
    // This interface file is one of several auxilliary Java sources
    // generated by JJTree. JavaCC contributes several others.

    int id = node.id;

    // eventually the buck stops here and we unwind the stack
    if ( node.id == JJTINTLITERAL )
        return Integer.parseInt( node.getText() );

    SimpleNode lhs = (SimpleNode) node.jjtGetChild(0);
    SimpleNode rhs = (SimpleNode) node.jjtGetChild(1);

    switch( id )
    {
        case JJTADD :      return eval( lhs ) + eval( rhs );
        case JJTSUBTRACT : return eval( lhs ) - eval( rhs );
        case JJTMULT :     return eval( lhs ) * eval( rhs );
        case JJTDIV :      return eval( lhs ) / eval( rhs );

        default :

            throw new java.lang.IllegalArgumentException(
                "eval(): invalid operator!" );
    }
}

```

Conclusion

If you want to see a much beefier version of an `eval()` function that handles much of the real XQuery grammar, feel free to download a copy of my open-source XQuery implementation, XQEngine. (See [Resources](#).) Its `TreeWalker.eval()` routine *cases* on some 30-odd XQuery node types. A `.jjt` script is provided.

Resources

- Participate in the [discussion forum](#) on this article. (You can also click **Discuss** at the top or bottom of the article to access the forum.)
- Review [Part 1](#) of this article for a brief discussion of grammars, parsers, and BNF, and an introduction to JavaCC. You'll also find sample code that uses JavaCC to build a custom parser, starting from a BNF description of the grammar.
- 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.
- Want to find out more about BNF? Check out [Wikipedia.org](#).
- Find more information on the technologies covered in this article at the developerWorks [XML](#) and [Java technology](#) zones.
- [IBM WebSphere Studio](#) provides a suite of tools that automate XML development, both in Java and in other languages. It is closely integrated with the [WebSphere Application Server](#), but can also be used with other J2EE servers.
- 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.

 [Discuss](#)  [e-mail it!](#)

What do you think of this document?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Send us your comments or click **Discuss** to share your comments with others.