Generating Parsers with JavaCC

Tom Copeland

Generating Parsers with JavaCC

Tom Copeland

Copyright 2009, Centennial Books. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher. Many of the names used by manufacturers and sellers to designate their products are claimed as trademarks. Where those names appear and we were aware of the trademark claim, the designations have been indicated with the trademark sign. Every attempt has been made to ensure this book's accuracy. No responsibility is assumed, however, for errors, omissions or damages resulting from the use of the information contained herein.

The cover of this book was produced by Microsoft Publisher. The interior was produced by the DocBook XSL-FO stylesheets, a custom Ruby/XSL layer, and AltSoft XML2PDF.

Library of Congress Control Number 2007929378

ISBN 0-9762214-3-8

Editing: Elizabeth Joyce

Printed in the United States of America

Alexandria, VA 22304

Table of Contents

List of Figures

List of Examples

Preface

JavaCC is one of the oldest and most widely used parser generators for the Java programming language, but it's sometimes considered hard to use. Some programmers haven't encountered the parser generator paradigm and thus are unfamiliar with the process of writing a specification and then generating a parser from it. Others have distant memories of using another parser generator (such as the venerable lex/yacc combination) and now need to ramp up on JavaCC's syntax and capabilities for specifying a grammar. Even programmers who have used JavaCC may be unsure about how certain advanced features work or how to effectively debug a particular JavaCC problem.

When I first began using JavaCC, a coworker set up the JavaCC grammar for me and wrote a few Ant targets to get things started. For the next year or so I occasionally would make a change to the grammar, close my eyes, run Ant and the unit tests and hope that everything still worked. Eventually I had a few epiphanies (tokenizing happens first! node descriptors can make a tree simpler!) and began to actually understand the large `.jjt` file that I had hitherto avoided as much as possible. After that I wrote more grammars, hung out on the JavaCC mailing lists, committed a few fixes to JavaCC's source code repository, and slowly got more and more familiar with both JavaCC and JJTree.

There's a certain amount of online documentation for JavaCC; typing "JavaCC" into your favorite search engine will bring up a slew[1] of results: articles, blog entries, the archives of the JavaCC mailing lists, documentation from the JavaCC site, and so forth. One of the more helpful of these results is the "JavaCC FAQ"[2], which is maintained by Dr. Theodore Norvell and contains lots of helpful information. I've reread this FAQ many times and always seem to pick up something new on each trip through it.

Despite this existing online documentation, JavaCC can still seem unapproachable and answers to more advanced questions can be elusive. This is a shame, since JavaCC is a versatile and powerful utility and can easily handle most parsing jobs. Better still, there are many existing JavaCC grammars, so if you need to process a particular language there's a good chance that a JavaCC grammar already exists for it. Understanding how JavaCC works and how to use it effectively can turn a month-long project into a problem that can be solved in a day.

With this in mind, making JavaCC more accessible is one of my prime motivators for writing this book. My intent is that a programmer who is working on a parsing job can pick up this book, flip to the table of contents or index, and quickly find a solution to almost any JavaCC problem. Whether you're writing a grammar from scratch or modifying an existing grammar, I hope you'll find answers to your questions here.

How This Book is Organized

This book is divided into eleven chapters and two appendices.

[1] Google reports 207,000 results, which is at least one slew
[2] http://www.engr.mun.ca/~theo/JavaCC-FAQ/

- Chapter 1 provides an introduction and overview of JavaCC and JJTree. It also shows you how to run JavaCC from the command line, using the Ant build tool, and using the Maven build tool.
- Chapter 2 discusses writing tokenizers with JavaCC. It covers regular expression productions, lexical actions, lexical states, and a variety of techniques for creating and modifying tokens.
- Chapter 3 covers the heart of JavaCC: writing syntactic specifications for parsers. We'll discuss basic grammar elements, syntactic actions, various types of lookahead, and ways to avoid common parsing pitfalls.
- Chapter 4 describes JavaCC's built-in tree builder, JJTree. This chapter looks at moving a grammar from JavaCC to JJTree, node creation and node descriptors, and writing programs that traverse an abstract syntax tree.
- Chapter 5 is a short chapter which covers JavaCC's documentation utility, JJDoc. We discuss the JJDoc functionality and review some recently added features.
- Chapter 6 covers processing Unicode data with JavaCC. We look at the facilities that JavaCC provides for handling Unicode escape sequences, some common character encoding issues and solutions, and the use of JavaCC's Unicode-related options.
- Chapter 7 discusses error handling strategies. This chapter covers the detection, reporting, and recovery of errors in the tokenizing, parsing, and tree building stages.
- Chapter 8 is a case study. In this chapter we go through one of the more complicated grammars you'll see—the JavaCC grammar itself. We look at the way this grammar solves a variety of parsing problems and along the way we take the time to discuss the structure of JavaCC grammars.
- Chapter 9 discusses strategies for testing JavaCC. We look at using the popular unit testing tool JUnit to write tests for tokenizers and parsers. We also explore a technique for using XPath expressions to test the abstract syntax tree produced by JJTree.
- Chapter 10 demonstrates the operation of the Eclipse JavaCC plugin. We discuss installation and configuration of the plugin, as well an overview of its many features and how it can save you time and effort in your JavaCC programming tasks.
- Chapter 11 wraps things up with a couple of small example grammars to illustrate various techniques and tie things together. This chapter shows that JavaCC is useful even if you're not writing a large and complex language grammar; JavaCC is good for small jobs too!
- Appendix A covers the JavaCC directory layout. We also discuss how to build JavaCC from source.
- Appendix B shows a listing of the JavaCC, JJTree, JJDoc, and JTB options. These options are covered in detail in the appropriate chapters, but they are presented here for quick reference.

Who This Book is For

I've targeted this book towards programmers who are working with JavaCC and JJTree to solve parsing problems. But, with any luck, this book will be helpful to a variety of folks. Here are some thoughts on what you'll get from this book from several perspectives:

- If you're a manager of a project that's using JavaCC this book can serve as insurance. It covers most if not all of the important features of JavaCC's tokenizer, parser, and tree builder, and it also covers tricky but important topics such as handling international character sets. If JavaCC has been one of the darker corners of the project it need be no more.

- If you're a student, this book will help you translate examples from your compiler theory course textbook and handouts into running Java code. JavaCC contains all the technical abilities that you'll see mentioned in your courses, such as EBNF support and LL(k) parsing. It also contains a bunch of extra features such as lexical actions and states, the ability to heavily customize the parse tree, and extensive debugging capabilities that can make solving parsing problems much easier. Finally, JavaCC is used extensively in both the corporate and the open source world, and getting familiar with it now can pay off when you get involved in programming outside of the classroom.

- Instructors will not find a great deal of new theoretical material in this book, but it should be helpful to your students as they try to work out the intricacies of left recursion and operator associativity. You may find that the sections on testing JavaCC parsers help keep students on track, and the Eclipse plugin can remove some of the command line drudgery. Many of the topics covered here were suggested by instructors, and additional feedback is certainly quite welcome.

- Finally, the main audience for this book is the programmer who is either creating a JavaCC grammar from scratch or working to improve, enhance, or debug an existing grammar. The current JavaCC documentation contains some good information, but reading it is like jumping into the deep end of the pool. Hopefully the many examples in this book will provide a slower introduction and perhaps fill in some of the gaps. A quick Google search turns up many JavaCC tutorials, but a five page online article can only go so deep. This book aims to provide more details on some of the darker corners of both JavaCC and JJTree.

  More generally, I think you'll find that studying and using JavaCC will make you a better programmer. Computer programs are written in human-readable text, and someone has to write a program to translate that source code into machine code. JavaCC covers the front end of that process: reading in the code, validating and parsing it, and producing a usable syntax tree. This is halfway to a complete compiler, and all that's left is writing the code generation back end. Once you get a good feel for the mechanics of this process you'll notice compiler technologies being used in all sorts of places, and you'll be able to contribute to such efforts far more effectively.

What's Not Included

This book is about a particular parser generator, JavaCC. It's meant to provide a good reference and numerous examples for that tool. But that means that this is not a book on compiler construction, nor is it a book on parsing theory. So you won't see discussions that would occur in those types of books; for example, there aren't any demonstrations of NFA to DFA conversions. Nor are there in-depth discussions of grammar formalisms such as the Chomsky hierarchy. I've included in the bibliography several references to books which cover those areas in great detail.

Of course, I'll occasionally dip into a bit of theory if I think it'll help you understand a particular JavaCC feature a bit more. But for the most part, this book stays solidly on the practical side.

Conventions

Code samples, program output, and script and program names are all written using a `monospace` font. Names of various JavaCC classes are usually presented in `monospace` as well, for example, "This generates a `TokenManager` interface." Sometimes if a class is referred to in a general sense it'll use a regular font, for example, "You may need to write your own TokenManager if you want full control."

The first time a technical term is used it's written in *italics*. Occasionally I'll repeat the italics if a term was first used in a high-level sense much earlier in the book.

Code Samples

For updates, errata, and example code, please see this book's web site:

http://generatingparserswithjavacc.com/
`

Second Edition Notes

This is the second edition, and as such it shares many similarities with the first edition; the book's overall structure and content are more or less the same. But JavaCC 4.1 was released in late summer 2008 and several bugfix releases have occurred since then. Thus, this edition covers the new features and bugfixes available in these new releases. I've tried to make these JavaCC version-related updates easy to locate by tagging them all with "new since JavaCC 4.0" in the index, so you can flip to the back and look up all the things that have changed. I've also brought the infrastructure and utilities that are mentioned throughout the book (Java, Ant, Maven, Eclipse) up to date so they'll more closely match what you're likely to find when you use JavaCC in your projects. Finally, a few small bugs have been found and fixed in the recent releases and I've mentioned those here as well.

Contacting Us

We've gone to great lengths to ensure the material in the book is accurate, but some errors may have crept in along the way. Please let us know about errors, confusing bits, or anything else you think we might be interested in fixing in future editions. You can contact Centennial Books at:

Centennial Books
1591 Chapel Hill Drive
Alexandria, VA 22304 USA
Phone: 703-751-6162
Fax: 703-751-2045
Email: <centennialbooks@comcast.net>

For technical issues, please feel free to contact the author directly at <tom@infoether.com>.

Credits

First and foremost, thanks to my wife, Alina, for her patience as I toiled away down in the office day after day working on this book. Thanks to our children (Maria, Tommy, Anna, Sarah, Steven, and James) for their many hugs throughout this time as well.

Thanks to my first edition technical reviewers: Dr. Kenneth Beesley, Sophie Quigley, and Rémi Koutchérawy. They filled in numerous gaps and without their excellent feedback this book would be much less readable. Any mistakes or omissions that remain are mine.

A huge thank-you to Dr. Sriram Sankar and Sreenivasa Viswanadha for writing and maintaining JavaCC. They did the hard work of implementing JavaCC, shaking out all the initial bugs, and optimizing both JavaCC itself and the code that JavaCC generates. They deserve all the credit for this superb utility that's been so useful to so many people. Thanks also to Paul Cager for his efforts in getting JavaCC 4.1 (and subsequent versions) released; he's done great work in implementing features, fixing bugs, and generally keeping the ball rolling.

Thanks to Dr. Michael Van De Vanter at Sun Microsystem for the work he did in coordinating the JavaCC open sourcing effort. Along the same lines, another thanks to Paul Cager for his work a few years ago in cleaning up the JavaCC licensing terms.

I've benefited greatly from the discussions on the `javacc-users` mailing list. Dr. Sankar and Sreenivasa Viswanadha have posted volumes of useful material there. Dr. Nathan Ryan's posts were quite helpful; at one point I went through the list archives and reread each one.

A second thanks to Dr. Kenneth Beesley for the excellent white papers he's written on several JavaCC topics; they were quite helpful for helping me understand various tricky bits. Thanks to Dr. Theodore Norvell for writing and maintaining the JavaCC FAQ, which served as one of my first sources of learning material. Thanks to Dr. Cur-

tis Dyreson for suggesting the inclusion of a chapter on testing grammars. Thanks to Matt Kirkey for doing an early review of the book and providing some helpful feedback.

Thanks to Dr. Nathan Ryan for his extensive feedback on the first edition. Thanks also to Nathan Ward and John Wilson for finding and reporting several typos in the first edition.

Chapter 1. Introduction to JavaCC

Courageously, and with a free desire; Attending but the signal to begin.--William Shakespeare, "King Richard the Second"

What is JavaCC?

Overview

Parsing data is part of every programmer's career. The parsing task can be simple, like splitting a comma-delimited list of words or a small configuration file that contains key-value pairs. On the other hand, it can be quite complex, like reading a series of Structured Query Language (SQL) statements to see which database tables are being queried the most, or reading a Java program and then writing out that program reformatted to meet your company's standards.

If the data stream is simple, as with a comma-delimited list, it's easy to meet the need using the built-in parsing functionality of most languages. Java, for example, provides built-in packages like `java.util.regex` that handle many cases. But if the data stream is complicated, you'll need to write the parser, and that can become a major time-consumer. Even if it's simple to begin with, the parser code gets more and more tangled as you try to meet all the edge cases. As the parser becomes more tangled, it gets slower and, equally important, harder to maintain. After a while the programming team responsible for the parser starts to resist any changes in the input format—if only to avoid revisiting the parser code.

Fortunately, there is a better way. Instead of writing code to parse a data stream, you can use a *parser generator* to write the parser for you. Instead of burying the data format in the code—or, worse yet, documenting it once and then allowing that document to get out of sync with the code—you can write a concise description of the data format in a *grammar*. The parser generator can then read the grammar file and generate the source code necessary to read and validate data in the format you described.

The Java Compiler Compiler (JavaCC) is one such parser generator. The JavaCC project started in 1996 when Dr. Sriram Sankar wrote the initial version ("Jack") as part of a Sun Microsystems internal project. Dr. Sankar was then joined by Sreenivasa Viswanadha, who wrote the lexical analyzer generator. JavaCC first bootstrapped itself in June, 1996. Soon afterwards, Robert Duncan joined the team and put JavaCC to its first real test: generating a Prolog parser. In "The JavaCC Story", Dr Sankar relates that the first Prolog parser took over five minutes to parse a test file. Within months the team had reduced that time to just 27 seconds. JavaCC, version 0.5, was first released to the public in October, 1996.

Later in 1996, Robert simplified the tree building task by building a JavaCC preprocessor that allowed the tree to be easily customized. This preprocessor, originally named "Beanstalk", is now known as JJTree. Robert also built the JJDoc utility for documenting JavaCC grammars.

In 1997 Dr. Sankar founded a new company, Metamata, to develop cross platform Java development tools. Since Metamata had licensed JavaCC from Sun, work on JavaCC continued. In April 2001, Metamata was acquired by Webgain. After Webgain closed down in August 2002, JavaCC was hosted on Sun's "Experimental Stuff" project site. In June, 2003, thanks to efforts spearheaded by Dr. Michael Van De Vanter, it was released as a BSD-licensed open source project on Sun's Java community site, java.net. At this printing, it continues to reside and thrive on java.net, with version 4.2 being released in November 2008.

JavaCC generates source code for three main components: a tokenizer, a parser, and a tree builder. JavaCC generates both the *tokenizer* (also known as a *lexical analyzer*, or *lexer*) and the syntax analyzer, also known as a *parser* as part of almost every parsing job. The tokenizer transforms your input data from a stream of characters into a stream of *tokens* that comprise the meaningful units. The parser establishes the relationships between the tokens. JavaCC also comes with JJTree, a *tree builder*, that produces a modified grammar and source code that can build a tree structure from the token relationships.

In the next sections we'll take a closer look at these three major JavaCC components—the tokenizer, the parser, and the tree builder.

The Tokenizer

To understand the tokenizer's role, imagine you want to parse a tiny program written in the Ruby [1] programming language. Here's a Ruby program that declares a class named `Hello`:

```
# examples/introduction/hello_class.rb

1    class Hello
2    end
```

If you look at the code above character by character—as a computer must do—you'll see `c`, `l`, `a`, `s` and so forth. But that's not what we want— instead, we want to see the Ruby keyword `class`, followed by the class name `Hello`, followed by the end of class keyword `end`. Combining characters to form those "words" is the tokenizer's job. To do this, the tokenizer uses the *regular expressions* and *lexical states* that you have defined using the grammar.

The Parser

Imagine the tokenizer has processed the input data and has produced the three tokens: `class`, `Hello`, and `end`. That doesn't tell us much about the Ruby language construct that is formed by these three words. Figuring that out is the job of the next link in the chain, the parser. The parser recognizes that it's been handed a series of tokens that conform to the syntax of a Ruby class declaration. For the parser to combine the three tokens into a syntactical unit, we had to specify this syntax using a grammar that itself uses Extended Backus-Naur Format (EBNF, named for the creators of the BNF specification, John Backus and Peter Naur). In our example, the syntactic unit would be a `ClassDeclaration`, and we could add code to print a message each time a `ClassDeclaration` was encountered.

[1] http://ruby-lang.org/

Specifying the syntax in JavaCC enables the parser to combine the tokens into syntactic units, and it also ensures that token sequences that don't match one of the expected syntactic units are rejected. So, the tokenizer ensures that each "word" of the input is valid, and the parser ensures that the "sentences" that those words form are also valid.

The tree builder

At this point, we've progressed from a series of raw characters to a `ClassDeclaration` that we know is syntactically well-formed. Knowing what construct is formed is great, but it'd be even better to be able to get inside that `ClassDeclaration` and change the code. Let's suppose we have a method declaration inside that class:

```
# examples/introduction/hello_class_method.rb

1    class Hello
2      def hi
3      end
4    end
```

Let's further suppose we want to delete that method declaration, or change the number of parameters that it accepts. The JJTree utility makes it much easier for us to do that. First, through a variety of grammar additions, you can cause JavaCC to generate a class for each syntactical unit as well as an annotated grammar. JavaCC can then process that annotated grammar to create a parser. When we compile and run the parser with some input data, the parser forms the JJTree-generated classes into an object tree. In our example, the `ClassDeclaration` object will have the `MethodDeclaration` object as a child node, and the `MethodDeclaration` will also have a reference back to the `ClassDeclaration` as a parent. We can then traverse this object tree and make changes as we see fit.

JJTree adds several new keywords and constructs to the JavaCC grammar. This means that in addition to working with the object tree as described above, we can write our grammar using JJTree-specific constructs that can manipulate the object tree as it's being built. For example, with a few simple statements we could tell our generated grammar to drop `MethodDeclaration` nodes from the object tree. The grammar would still validate the syntactic structure of the input data, including the method declarations, but it would not include those method declarations in the runtime object tree.

A Simple Example

That's a high-level overview of the major JavaCC components. To make these components a bit more concrete, let's look at an actual JavaCC grammar file. This grammar is as simple as we can make it while still demonstrating as many of the JavaCC pieces as possible. Here's the first example: a JavaCC grammar that parses the letter "a".

Example 1.1. A simple grammar

 # examples/introduction/simple.jj

```
1    options {
2      STATIC=true;
3      JDK_VERSION="1.5";
4    }
5    PARSER_BEGIN(SimpleParser)
6    import java.io.*;
7    public class SimpleParser {
8      public static void main (String args[] ) throws ParseException {
9        String someInput = "a";
10       StringReader sr = new StringReader(someInput);
11       Reader r = new BufferedReader(sr);
12       SimpleParser parser = new SimpleParser(r);
13       parser.A();
14     }
15   }
16   PARSER_END(SimpleParser)
17   TOKEN :
18   {
19     <A : "a">
20   }
21   void A() : {}
22   {
23     <A> {System.out.println("Found an 'a'!");}
24   }
```

There are four primary parts to a JavaCC grammar file.

- The grammar starts with an optional `options` section that contains key/value pairs for various JavaCC settings. JavaCC 4.1 and later releases allow for an empty `options` section; versions 4.0 and earlier of JavaCC required at least one setting inside an `options` block if that block appeared in the grammar. Some settings can be set to a boolean value, others expect string or numeric values. We'll cover the meanings of each of these settings in detail later on, but for the example above you can see how it works; we've set the `STATIC` parameter to a value of `true` and the `JDK_VERSION` parameter to a value of `1.5`.

- Next is the parser class definition. This is mandatory and must contain at least a class declaration with the name of the parser. It is surrounded by `PARSER_BEGIN`/`PARSER_END` markers. You can place arbitrary Java code in this section and invoke it from the parser; the code supplied here gets inserted into the generated parser's source code. As with the above example, this is something you'll see frequently, a class declaration and a `main` method definition. It will allow you to more easily test the parser once it's generated.

- Next comes the lexical specification. This is where you define the tokenizer settings: the lexical states, the token names, and the token definitions. We define only a single `A` token that matches the literal value `a`.

- Last comes the syntactic specification. This is where we specify token sequences that we expect to encounter. In this case, we're only expecting one letter `a`, so we have one *nonterminal* named `A` that consists of one token that was defined back in the lexical specification. We've also placed a *syntactic action* after the token. This syntactic action is a string of Java code which will be inserted into the generated parser and executed whenever the body of this nonterminal is encountered.

Here's an example of the output created when we use JavaCC to generate a parser, then compile and run it:

```
$ javacc simple.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java SimpleParser
Found an 'a'!
```

Success! By now you're probably impatient to achieve such impressive results on your own, so let's see what it takes to get JavaCC up and running on your workstation.

Installing JavaCC

There are a few prerequisites to installing JavaCC. You need a bit of free disk space. Once unpacked, JavaCC takes up slightly over two megabytes. Since it's written in Java, it runs on any operating system which supports a Java Virtual Machine (JVM), and since it generates Java source code, you need the entire Java Development Kit (JDK) to compile that code. So, if the latest JDK is not installed on your computer, you need to download and install it from the Sun web site.[2] The installation notes for the JDK, which vary between operating systems, are well outlined on the Sun web site, so I won't reproduce them here.

Once you've got a JDK in place, you'll need to download and install JavaCC itself. For the most recent version (4.2 at this writing), go to the JavaCC home page [3] where you'll see links to both a `zip` file and a compressed `tar` file (i.e., a `.tar.gz` file) containing the binary distributions of JavaCC.

Unix installation

To install JavaCC on a Unix variant such as Linux, Solaris, or FreeBSD, download the `.tar.gz` file. Once you've got the file on your computer, put it in a handy spot. If you have root privileges on the machine, you may want to place it in `/usr/local/`. If not, placing it in your home directory should work well. Now you've got a file like this:

```
$ ls -l /usr/local/javacc-4.2.tar.gz
-rw-r--r--  1 root  wheel  471820 Dec 12 20:44 /usr/local/javacc-4.2.tar.gz
```

Next, unpack the file using the following commmand:

```
$ sudo tar -zxf javacc-4.2.tar.gz -C /usr/local/
```

The `-f` file tells `tar` which file we're working with, the `-z` option uncompresses the file, the `-x` option unpacks the file, and the `-C` tells `tar` to do the operation in the `/usr/local/` directory. This command will unpack the files into a `/usr/local/javacc-4.2` directory.

[2] http://java.sun.com/javase/downloads/index.jsp
[3] https://javacc.dev.java.net/

That's all that's (strictly) necessary to install JavaCC, but here are a couple of conveniences. To upgrade easily when the next version of JavaCC is released, add a symbolic link for the current version of JavaCC. Enter this command:

```
$ sudo ln -s /usr/local/javacc-4.2 /usr/local/javacc
```

You can then use `/usr/local/javacc/bin/javacc` in your scripts to run JavaCC. When the next JavaCC release comes out, you change the symbolic link to point to the new version and all your scripts will begin to use the new version.

Typing `/usr/local/javacc/bin/javacc` every time you run JavaCC is mind-numbing—let's add an environment variable to avoid that. The syntax to do this varies with each Unix shell, but for Bash, adding this to your `.bash_profile` should do the trick:

```
$PATH=$PATH:/usr/local/javacc/bin/ && export PATH
```

Note that we're already using the symbolic link to good advantage by using `/usr/local/javacc/bin/` rather than the directory name with the version number included. To ensure it's working properly, type `javacc` from a terminal window and you should see:

```
$ javacc
Java Compiler Compiler Version 4.2 (Parser Generator)
[ ... lots of other output elided ... ]
```

It's possible you'll get an error along these lines:

```
$ javacc /usr/local/javacc/bin/javacc:
line 8: java: command not found
```

If that happens, it means that the shell script running JavaCC is unable to locate the Java executable. This can usually be fixed by ensuring that `java` is in your `PATH` somewhere. For example, if `java` is in the directory `/usr/local/java/bin/`, ensure that directory is in the `PATH`.

That's it for installing JavaCC on Unix. On to Windows!

Windows installation

The same prerequisite applies for Windows; in order to run JavaCC you'll need to have a JDK installed. Once that's in place, download the `javacc-4.2.zip` file and place it in a convenient location (I usually place it on the root directory of my primary hard drive). Next, unzip it using a decompression tool that supports PKZIP; WinZip® will do this, but you can also find various free tools that do the job [4].

Once you've got it unzipped into, say, `c:\javacc-4.2\` , you may want to put the `javacc-4.2\bin` directory on your `PATH` so that you can run the `javacc` command from any command line window. To add that directory to your `PATH` , navigate to your "Control Panel", open the "System" dialog box, click the "Advanced" tab, and click the "Environmental Variables" button. Now look for a `PATH` variable under the "System variables" section. If it's there, you can open a dialog box with the "Edit" button; if not, you can create it by clicking the "New" button. Add `c:\javacc-4.2\bin\` to the end of the "Variable value" field and click "OK" numerous times to close all the dialog boxes. Note that this change doesn't affect the `PATH` setting in any

[4] Like Info-Zip: http://www.info-zip.org/UnZip.html#Win32

command line windows that you currently have open, but the next time you open a command line window, you'll be able to simply type `javacc` to get JavaCC rolling.

Running JavaCC

There are various ways to run JavaCC: from the command line, from an Ant script, using a Maven goal, or from an integrated development environment (IDE). We'll start by using the command line and move on from there.

Command line operation

A simple way to see JavaCC in action is to open a terminal window and run it. If you've already got JavaCC on your `PATH` , just type `javacc`. You'll be rewarded with a flood of output:

```
$ javacc
Java Compiler Compiler Version 4.2 (Parser Generator)

Usage:
    javacc option-settings inputfile

"option-settings" is a sequence of settings separated by spaces.
Each option setting must be of one of the following forms:

    -optionname=value (e.g., -STATIC=false)
    -optionname:value (e.g., -STATIC:false)
    -optionname       (equivalent to -optionname=true.  e.g., -STATIC)
    -NOoptionname     (equivalent to -optionname=false. e.g., -NOSTATIC)

Option settings are not case-sensitive, so one can say "-nOsTaTiC" instead
of "-NOSTATIC".  Option values must be appropriate for the corresponding
option, and must be either an integer, a boolean, or a string value.

The integer valued options are:

    LOOKAHEAD              (default 1)
    CHOICE_AMBIGUITY_CHECK (default 2)
    OTHER_AMBIGUITY_CHECK  (default 1)

The boolean valued options are:

    STATIC                 (default true)
    SUPPORT_CLASS_VISIBILITY_PUBLIC (default true)
    DEBUG_PARSER           (default false)
    DEBUG_LOOKAHEAD        (default false)
    DEBUG_TOKEN_MANAGER    (default false)
    ERROR_REPORTING        (default true)
    JAVA_UNICODE_ESCAPE    (default false)
    UNICODE_INPUT          (default false)
    IGNORE_CASE            (default false)
    COMMON_TOKEN_ACTION    (default false)
    USER_TOKEN_MANAGER     (default false)
    USER_CHAR_STREAM       (default false)
    BUILD_PARSER           (default true)
    BUILD_TOKEN_MANAGER    (default true)
    TOKEN_MANAGER_USES_PARSER (default false)
    SANITY_CHECK           (default true)
    FORCE_LA_CHECK         (default false)
    CACHE_TOKENS           (default false)
    KEEP_LINE_COLUMN       (default true)

The string valued options are:

    OUTPUT_DIRECTORY       (default Current Directory)
    TOKEN_EXTENDS          (default java.lang.Object)
```

```
    TOKEN_FACTORY           (default none)
    JDK_VERSION             (default 1.5)
    GRAMMAR_ENCODING        (defaults to platform file encoding)

EXAMPLE:
    javacc -STATIC=false -LOOKAHEAD:2 -debug_parser mygrammar.jj
```

We'll sort through all these options later; for now, just be aware that you can easily get JavaCC to process a grammar file by running it from the command line. Let's go ahead and do that with the simple grammar from page 4; you saw the output earlier and now you should see the same results on your machine:

```
$ javacc simple.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ ls *.java
ParseException.java
SimpleCharStream.java
SimpleParser.java
SimpleParserConstants.java
SimpleParserTokenManager.java
Token.java
TokenMgrError.java
$ javac *.java
$ java SimpleParser
Found an 'a'!
```

Looks good! In the sample run above, we did a couple of things:

- We ran JavaCC to generate the tokenizer and parser from the `simple.jj` grammar definition. The tokenizer is contained by `SimpleParserTokenManager.java`, the parser is defined in `SimpleParser.java`, and there are a couple of other supporting files as well. Note that flurry of "`Token.java does not exist. Will create a new one.`" output messages. Those will be replaced with a slightly different message (`File "Token.java" is being rebuilt.`) on subsequent JavaCC runs; JavaCC regenerates the various files each time it's run unless the files have been modified by hand after they were generated. Note that the only file we've edited is `simple.jj`; all the others were generated by JavaCC.
- We ran the Java compiler `(javac)` to compile the source files that were generated by JavaCC.
- We ran the JVM on the `SimpleParser` class file that was produced by the Java compiler. This executed our grammar, which successfully parsed our hardcoded string of "a".

To use one of the JavaCC options, we can add it to the grammar file itself as we did with the `STATIC` option. Alternatively, we can pass it in the command line using the format described in the help message. The example below sets the `DEBUG_TOKEN_MANAGER` option to `true`:

```
$ javacc -DEBUG_TOKEN_MANAGER=true simple.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java SimpleParser
Current character : a (97) at line 1 column 1
   No more string literal token matches are possible.
   Currently matched the first 1 characters as a "a" token.
****** FOUND A "a" MATCH (a) ******

Found an 'a'!
```

Since we added the `DEBUG_TOKEN_MANAGER` option we got an additional flurry of output as the tokenizer processed the input stream.

Options set on the command line take precedence over those specified in the `options` section of the grammar. This is quite handy when you're experimenting with a grammar; just pass in one of the debug options on the command line and you'll get lots of information without having to tweak the grammar file itself.

The Ant task

Ant is a cross-platform Java build utility. It helps manage compilation, generates Javadoc, creates Java archives (`jar` files), and does pretty much anything else you might want to do with Java source code. In addition, it has built-in facilities for running JavaCC and JJTree.

If you don't have Ant installed, go to the Ant home page [5] and click on the "binary distributions" link to get the latest version. Once you've got the appropriate file (a `.tar.gz` file for Unix, a `.zip` for Windows), unpack it into a directory. To get it ready to go you'll need to set a few environment variables:

- ANT_HOME: This points to the directory in which you installed Ant.
- JAVA_HOME: This points to your JDK directory, e.g., `/usr/local/jdk1.6.0/`.

See the earlier notes on creating environment variables (page 6) for details on setting `JAVA_HOME` and `ANT_HOME`. You'll probably want to add Ant's `bin` directory to the `PATH` and use the symlink technique so you can easily upgrade to future Ant versions. Once all that's all set you should be able to run Ant from the command line, e.g.:

```
$ ant -version
Apache Ant version 1.7.1 compiled on June 27 2008
```

The Ant web site has lots of documentation on Ant fundamentals. For more, see the bibiography for an excellent book on Ant. For now, the following quick tour of the important features will do. An Ant build file is an eXtensible Markup Language (XML) document consisting of a top-level `project` element containing `target` elements. The `target` elements contain invocations of Ant tasks, of which there are many: `jar`, `javadoc`, `copy`, and so forth.

Ant comes with tasks for JavaCC and JJTree built in [6]. Here's an Ant build file that's roughly equivalent to the previous section's invocation of JavaCC from the command line:

```
# examples/introduction/ant_simple.xml
```

[5] http://ant.apache.org/
[6] The class file is in ant-nodeps.jar

```
1    <project name="simple" basedir=".">
2      <target name="runjavacc">
3        <taskdef name="javacc"
4          classname="org.apache.tools.ant.taskdefs.optional.javacc.JavaCC"/>
5        <javacc target="simple.jj"
6          outputdirectory="." javacchome="/usr/local/javacc/" />
7        <javac srcdir="." destdir="."/>
8      </target>
9    </project>
```

There's a `project` element and a `runjavacc` target. The `runjavacc` target contains a `taskdef` element which tells Ant what class to instantiate when someone uses a `javacc` task. Then there's the `javacc` task invocation itself, which specifies the name of the grammar, the directory in which to place the generated parser source files, and the location of JavaCC itself. Finally, there's a call to the `javac` task to compile the generated parser.

To run this build file using Ant, run `ant` and pass it the `runjavacc` target name:

```
$ ant -f ant_simple.xml runjavacc
Buildfile: ant_simple.xml

runjavacc:
   [javacc] Java Compiler Compiler Version 4.2 (Parser Generator)
   [javacc] (type "javacc" with no arguments for help)
   [javacc] Reading from file /private/tmp/0.212320347461761/simple.jj . . .
   [javacc] File "TokenMgrError.java" does not exist.  Will create one.
   [javacc] File "ParseException.java" does not exist.  Will create one.
   [javacc] File "Token.java" does not exist.  Will create one.
   [javacc] File "SimpleCharStream.java" does not exist.  Will create one.
   [javacc] Parser generated successfully.
    [javac] Compiling 7 source files to /private/tmp/0.212320347461761

BUILD SUCCESSFUL
Total time: 1 second
```

The output of the Ant file is straightforward. First the tokenizer, parser, and associated files were generated from the grammar; then the Java compiler was run on those source files, and the directory now contains a couple of class files that contain the ready-to-run parser.

You can pass options through Ant to JavaCC by adding them to the `javacc` task invocation. Here's how the Ant build file looks if we want to turn on the token manager debugging output—note the new `debugtokenmanager` attribute inside the `javacc` element:

```
# examples/introduction/ant_param.xml

1    <project name="simple" default="runjavacc" basedir=".">
2      <target name="runjavacc">
3        <taskdef name="javacc"
4          classname="org.apache.tools.ant.taskdefs.optional.javacc.JavaCC"/>
5        <javacc target="simple.jj"
6          outputdirectory="."
7          debugtokenmanager="true"
8          javacchome="/usr/local/javacc/" />
9        <javac srcdir="." destdir="."/>
10     </target>
11   </project>
```

All the standard JavaCC options can be set in the same way; just add them to the `javacc` element.

Ant also uses a concept of dependencies. If the `runjavacc` target should be run before the `compile` target, you can specify that in a `depends` attribute inside the `target` element:

```
# examples/introduction/ant_depends.xml

1    <project name="simple" basedir=".">
2      <target name="runjavacc">
3        <taskdef name="javacc"
4          classname="org.apache.tools.ant.taskdefs.optional.javacc.JavaCC"/>
5        <javacc target="simple.jj"
6          outputdirectory="." javacchome="/usr/local/javacc/" />
7      </target>
8      <target name="compile" depends="runjavacc">
9        <javac srcdir="." destdir="."/>
10     </target>
11   </project>
```

You can see the difference in the output as Ant first runs the `runjavacc` target and then the `compile` target:

```
$ ant -f ant_depends.xml compile
Buildfile: ant_depends.xml

runjavacc:
   [javacc] Java Compiler Compiler Version 4.2 (Parser Generator)
   [javacc] (type "javacc" with no arguments for help)
   [javacc] Reading from file /private/tmp/0.154338715887121/simple.jj . . .
   [javacc] File "TokenMgrError.java" does not exist.  Will create one.
   [javacc] File "ParseException.java" does not exist.  Will create one.
   [javacc] File "Token.java" does not exist.  Will create one.
   [javacc] File "SimpleCharStream.java" does not exist.  Will create one.
   [javacc] Parser generated successfully.

compile:
    [javac] Compiling 7 source files to /private/tmp/0.154338715887121

BUILD SUCCESSFUL
Total time: 2 seconds
```

With this configuration the `target` elements are a bit smaller and easier to manage. Also, if you have a target that should do something after the parser is generated (like generating JavaDoc), you can specify that it also depends on the `runjavacc` target. This is preferable to duplicating the `javacc` invocation every time it's needed. You'll see this sort of target dependency management in most Ant build files.

Running JavaCC with Maven

Maven [7] is a "software build and comprehension" tool. It's also fair to call it "Ant on steroids", although without the hair loss and bad temper. Where Ant provides "tasks", Maven provides "goals" which apply to most projects; e.g., `install`, `compile`, and, of course, `javacc`. Maven 2 is the current stable release so we'll discuss running JavaCC with that version.

The Maven web site contains copious documentation [8] on getting started, so only a very brief overview is in order here. Maven is built around a central configuration file called the Project Object Model (POM). The POM, which is usually named `pom.xml`, describes a project: the project name, location of the source code, the path to the tests, and so forth. We'll step through creating a simple project and running the `javacc` goal to generate a parser.

[7] http://maven.apache.org/
[8] http://maven.apache.org/guides/getting-started/index.html

First, download the latest Maven release from the Maven web site. Then, unzip it into a directory and add the `maven-x.x.x/bin` directory to your `PATH` environment variable as described on page 6. Just as with the Ant setup, you'll need to set `JAVA_HOME` if you haven't already. To test your setup, run the `mvn` command:

```
$ mvn --version
Maven version: 2.0.10
Java version: 1.6.0_07
OS name: "mac os x" version: "10.5.6" arch: "x86_64" Family: "mac"
```

Next, you can generate a standard Maven layout for a test project by running the Maven `archetype` command. When you run this command, Maven will download a number of jar files from a central repository to get things working:

```
$ mvn archetype:generate -DgroupId=example.javacctest -DartifactId=javacc-test -
DinteractiveMode=false
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[... lots of other output elided ...]
```

After Maven finishes, it'll leave a directory structure that looks like this:

```
$ find .
.
./javacc-test
./javacc-test/pom.xml
./javacc-test/src
./javacc-test/src/main
./javacc-test/src/main/java
./javacc-test/src/main/java/example
./javacc-test/src/main/java/example/javacctest
./javacc-test/src/main/java/example/javacctest/App.java
./javacc-test/src/test
./javacc-test/src/test/java
./javacc-test/src/test/java/example
./javacc-test/src/test/java/example/javacctest
./javacc-test/src/test/java/example/javacctest/AppTest.java
```

There's no JavaCC grammar in place yet, but we'll fix that. Create a `src/main/javacc` directory, then copy the `simple.jj` grammar to that directory. Next, copy in the customized version of `pom.xml` that's located in `examples/introduction/pom.xml` - this `pom.xml` contains an XML snippet that tells Maven that it's dealing with code that include Java 1.5 language constructs. Then tell Maven to run JavaCC and generate a parser from the grammar. Note that we're now running Maven with the `-o` option to put it in 'offline' mode. In offline mode, Maven doesn't check for updated plugins so the build runs a bit quicker.

```
$ mvn -o javacc:javacc compile
[INFO]
NOTE: Maven is executing in offline mode. Any artifacts not already in your local
repository will be inaccessible.

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'javacc'.
[INFO] ------------------------------------------------------------------------
[INFO] Building javacc-test
[INFO]    task-segment: [javacc:javacc, compile]
[INFO] ------------------------------------------------------------------------
[INFO] [javacc:javacc]
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file /private/tmp/0.862326561276547/src/main/javacc/simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
[INFO] Processed 1 grammar
[INFO] [resources:resources]
[WARNING] Using platform encoding (MacRoman actually) to copy filtered resources,
 i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /private/tmp/0.862326561276547/src/
main/resources
[INFO] [compiler:compile]
[INFO] Compiling 8 source files to /private/tmp/0.862326561276547/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Thu Apr 02 00:37:17 EDT 2009
[INFO] Final Memory: 20M/48M
[INFO] ------------------------------------------------------------------------
```

This will generate the parser (plus lots of console output) and place it in the `target/generated-sources/javacc/` directory. It then compiles the parser along with any code in `src/java/main` and puts the class files into `target/classes`.[9]

You can pass parameters to the JavaCC goal by specifying them on the command line. To continue with the token manager debugging switch example, use a `-DdebugTokenManager` flag to turn on that particular option:

```
$ mvn -o -DdebugTokenManager=true javacc:javacc compile
[INFO]
NOTE: Maven is executing in offline mode. Any artifacts not already in your local
repository will be inaccessible.

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'javacc'.
[INFO] ------------------------------------------------------------------------
[INFO] Building javacc-test
[INFO]    task-segment: [javacc:javacc, compile]
[INFO] ------------------------------------------------------------------------
[INFO] [javacc:javacc]
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file /private/tmp/0.695988592033154/src/main/javacc/simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
[INFO] Processed 1 grammar
[INFO] [resources:resources]
[WARNING] Using platform encoding (MacRoman actually) to copy filtered resources,
 i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /private/tmp/0.695988592033154/src/
main/resources
[INFO] [compiler:compile]
[INFO] Compiling 8 source files to /private/tmp/0.695988592033154/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Thu Apr 02 00:37:25 EDT 2009
[INFO] Final Memory: 19M/45M
[INFO] ------------------------------------------------------------------------
$ java -classpath target/classes/ SimpleParser
Current character : a (97) at line 1 column 1
   No more string literal token matches are possible.
```

[9] You'll note that Maven has expectations about where a project's artifacts (code, grammars, etc) should be placed. You can override these locations, but if you're starting a new project using Maven from the beginning, it's worth considering using the default locations. If you stay with them, anyone familiar with Maven will immediately understand your directory layout.

```
   Currently matched the first 1 characters as a "a" token.
****** FOUND A "a" MATCH (a) ******

Found an 'a'!
```

As expected, Maven compiles the parser with the `DEBUG_TOKEN_MANAGER` option enabled and we see the tokenizer's debug output when we run the generated parser.

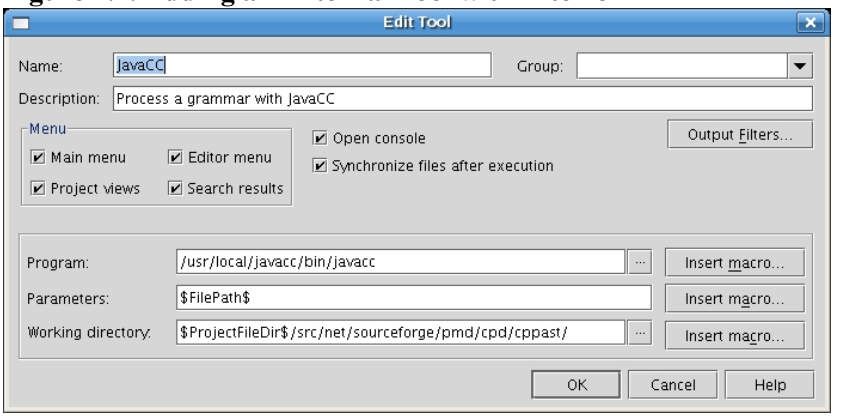
Maven includes other goals which do nifty things like: create a jar file from your project's class files, run your unit tests, generate Javadoc, and much more. If you want to try something beyond Ant, give it a whirl.

Running JavaCC From an IDE

Another way to run JavaCC is from your favorite Integrated Development Environment (IDE). Most IDEs support running arbitrary programs in some way, typically by specifying a program name and a series of arguments. Since these all work in much the same way, I'll describe how to set up JavaCC to run within the deservedly popular IDE produced by Jetbrains: IntelliJ IDEA[10].

IntelliJ supports running programs through an "External tools" option. To configure JavaCC using this option, select the "File" menu heading followed by the "Settings" menu item. This takes you to a dialog box with a list of the currently configured external tools. To add a new one, click the "Add" button and fill in the dialog box as shown below:

Figure 1.1. Adding an External Tool with IntelliJIDEA

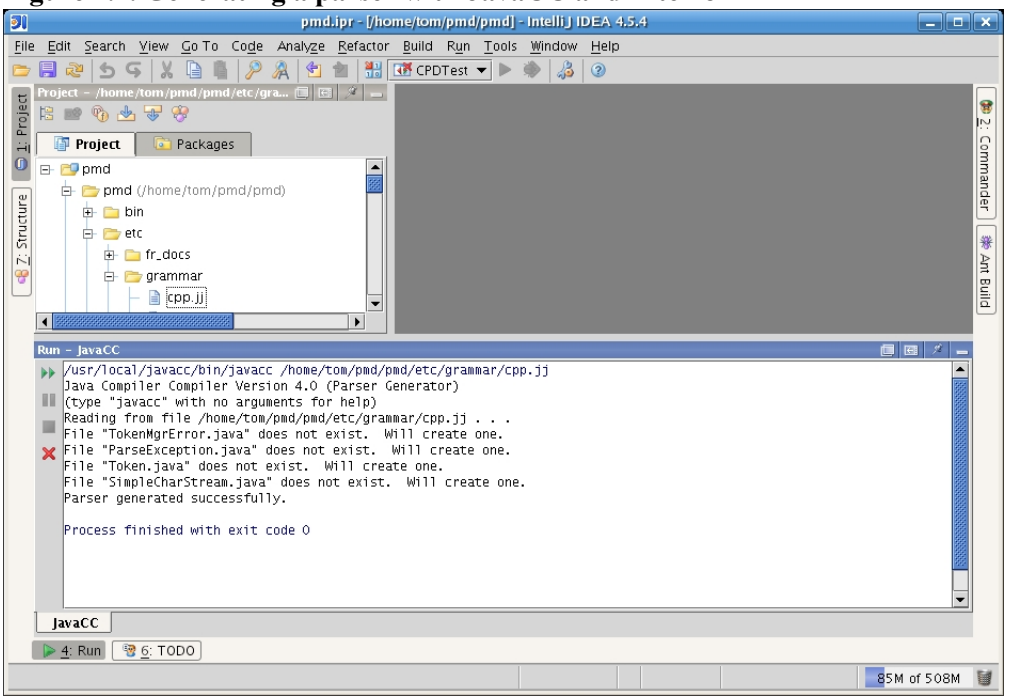


IntelliJIDEA now knows where JavaCC is located (via the "Program" field), what parameters should be passed to JavaCC (via the "Parameters" field), and where the files that JavaCC creates should be placed (via the "Working directory" field). If you want to pass additional parameters to JavaCC, modify the "Parameters" field to include them. The format is the same as on the command line, for example, you could change that field to be `$FilePath$ -DEBUG_TOKEN_MANAGER=true`.

Now that JavaCC is configured, you can run it on the grammar file of your choice. Right-click on the grammar file, select the "JavaCC" menu item, and you'll get output similar to this:

[10] http://www.jetbrains.com/idea/

Figure 1.2. Generating a parser with JavaCC and IntelliJ IDEA



Note, this procedure will vary for each IDE, but you've got the general idea.

Because the default JavaCC output directory is the current working directory, output for the above example will go to the working directory. If you have several projects that use JavaCC, you probably don't want the generated parser code for all projects in the same directory—so you might find it more useful to use the `OUTPUT_DIRECTORY` option (see page 62 for more details) in each grammar file. Simply set the working directory to the root of your project directory tree. For IntelliJIDEA, that means putting "`$ProjectFileDir$`" in the "Working Directory" field.

Some IDEs support a plugin Application Programming Interface (API) for adding new functionality to the IDE. The enables you (or someone else) to write Java code that the IDE loads at runtime to interface between the IDE and a third party utility, like JavaCC. This allows tight integration between the IDE and the utility—you can usually add new menu items, icons, a configuration panel, customized output, and so forth. The plugin API varies from IDE to IDE, so it's up to whoever uses (or sells) each IDE to write plugins for that IDE. There's an especially nice plugin written for the Eclipse IDE, and I've devoted chapter 10 to demonstrating it.

JavaCC and ANTLR

JavaCC isn't the only Java parser generator in town. Another popular parser generator is ANother Tool for Language Recognition (ANTLR)[11]. ANTLR and JavaCC share many characteristics (e.g., they're both LL(k)), but there are a few differences worth noting:

[11] http://antlr.org/

- ANTLR is both more flexible—you can generate code in Ruby, Objective C, Java, C++, C# or Python—and more limited—you can't blithely embed Java code in the grammar and expect the generated parser to work. Because JavaCC is limited to generating Java code only, you can embed Java code in lexical and syntactical actions and semantic lookahead directives and things will Just Work.
- ANTLR has a runtime dependency. That is, in order to use an ANTLR-generated parser, you'll need to have (in the case of Java) an `antlr.jar` file in the `CLASSPATH`. Other languages generated by ANTLR have similar requirements. The C++ runtime, for example, requires a shared object library. JavaCC generates self-contained parsers that have no dependencies on JavaCC itself. This may or may not be a problem in your environment, but it's something to consider.
- The ANTLR documentation is superb and ANTLR's author, Dr. Terence Parr, has written a book on ANTLR[12] that was published in May of 2007. JavaCC's documentation is a bit more muddled, although this book aims to rectify that situation.

Both ANTLR and JavaCC are fine parser generators, and you won't go wrong either way.

Summary

At this point you've seen some quick examples of JavaCC in action and a variety of ways to get JavaCC up and running. In the next chapter, we'll delve much more deeply into the first stage of the parsing process by studying the JavaCC tokenizer.

[12] http://www.pragmaticprogrammer.com/titles/tpantlr/index.html

Chapter 2. Tokenizing

Lepidus: But small to greater matters must give way. Domitus Enobarbus: Not if the small come first. --William Shakespeare, "Anthony and Cleopatra"

The Tokenizing Process

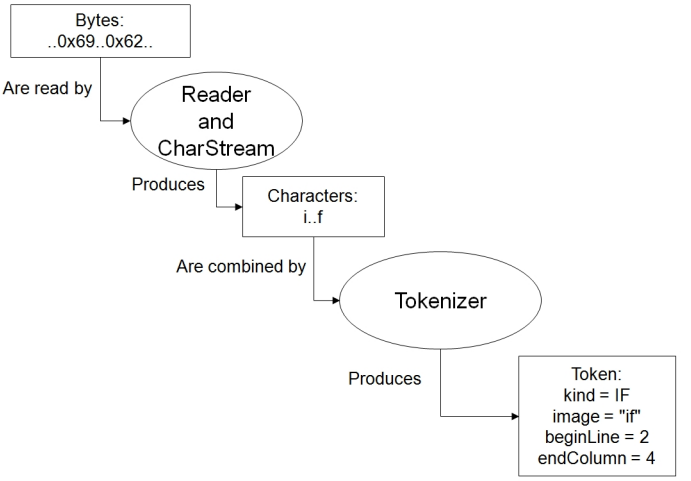
From Bytes to Tokens

As you'll recall from the overview in Chapter One, a JavaCC-generated parser tokenizes a file first, then parses it. In this chapter we'll discuss the tokenizing process—the process by which a set of input data is converted from a stream of bytes into a series of `Token` objects that can be fed to the parser.

The tokenizing process follows two distinct steps. First, the input data is converted from bytes into `char` primitives by a standard Java `java.io.Reader` implementation. These characters are then pulled through a character stream class, which can either be one of the JavaCC built in character stream classes `SimpleCharStream` or `JavaCharStream` or, in an edge case, can be an implentation of the `CharStream` interface. The character stream efficiently buffers the input data, keeps track of the current line and column number, and, in some cases, combines multiple bytes into a single character. Next, the tokenizer accepts those characters and combines them into the tokens. These tokens represent the "words" of the input data.

The diagram below illustrates this process. Two bytes with the hexadecimal values `0x69` and `0x62` are read from a byte stream. Next, a `Reader`, following the ASCII (American Standard Code for Information Interchange) character encoding, converts those bytes into two characters: `i` and `f`. The character stream implementation accepts these characters and tracks the line and column numbers. The two characters are then handed off to the tokenizer which combines them into a single `Token` object. The `Token` object is identified with a type value (`IF`). In addition, the `Token` is decorated with the line and column information collected by the `CharStream`.

Figure 2.1. The tokenizing process



The byte to character transformation is straightforward. In most instances, you can use the built-in character stream classes `SimpleCharStream` and `JavaCharStream` (though there are some Unicode wrinkles that we'll address in-depth in chapter 6).

One important point to keep in mind throughout these tokenizer discussions is that tokenizing happens before, and independently of, the parsing process. The input data is converted into a stream of tokens before the parser is brought into the picture; in most of the examples in this chapter we don't even generate the parser source code. Occasionally a message will be posted to the JavaCC user's list that betrays some confusion on this point. Someone might ask the question "how do I get my parser to view [some character] as a certain type of token?" But that's the wrong approach. Instead, you should expect the tokenizer to supply your parser with a stream of tokens that's independent of the structure that your parser imposes on them.

With that brief overview, let's take a look at the mechanisms that JavaCC provides for specifying the token definitions. By the end of this chapter you'll feel confident that you can write a token definition for anything that your JavaCC grammar might encounter.

From Characters to Tokens

The transformation of characters to tokens is controlled by the *lexical specification* section of the JavaCC grammar. A lexical specification consists of two primary components: *regular expression productions* and *lexical states*. A regular expression production consists primarily of a *regular expression* that defines the possible sequences of characters that can compose a particular token. A lexical state is a container for regular expression productions.

While all regular expression productions consist of regular expressions, not all regular expression productions do the same thing with the characters they consume. There are four distinct regular expression productions used to define the tokenizer: `TOKEN`, `SKIP`, `MORE`, and `SPECIAL_TOKEN`. Each does something different with the characters that it consumes; you'll see examples of all four in the following sections.

Generated Files

Generating the Tokenizer

Before we get into the nuts and bolts of regular expression productions, let's take a look at the files that JavaCC generates to form a tokenizer. Our first example grammar parses a single word: `hello`. Since we're not going to build a parser there's no need for JavaCC to create one, so we'll set the `BUILD_PARSER` option to `false` in the options section of the grammar. This will create fewer source files, a cleaner directory, a shorter compile time, and less clutter all around. Note that despite the fact that `BUILD_PARSER` is `false` we still need to declare a `PARSER_BEGIN`/`PARSER_END` section containing a `Literals` class definition; if we don't, JavaCC will exit with an error.

Example 2.1. A grammar that parses "hello"

```
# examples/tokenizer/literals_plain.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Literals)
5    public class Literals {}
6    PARSER_END(Literals)
7    TOKEN : {
8      <HELLO : "hello">
9    }
```

And to create the grammar, we'll run it through JavaCC:

```
$ javacc literals_plain.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_plain.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

JavaCC has now generated the following list of tokenizer files:

Example 2.2. The tokenizer files

```
$ ls -l *.java
-rw-r--r--  1 tom  wheel    440 Apr  2 00:37 LiteralsConstants.java
-rw-r--r--  1 tom  wheel   6357 Apr  2 00:37 LiteralsTokenManager.java
-rw-r--r--  1 tom  wheel   6147 Apr  2 00:37 ParseException.java
-rw-r--r--  1 tom  wheel  12144 Apr  2 00:37 SimpleCharStream.java
-rw-r--r--  1 tom  wheel   4055 Apr  2 00:37 Token.java
-rw-r--r--  1 tom  wheel   4399 Apr  2 00:37 TokenMgrError.java
```

The names of several of the generated files (`LiteralsTokenManager`, `LiteralsConstants`) are derived from the name in `PARSER_BEGIN` declaration. This can be, and in this case is, different than the grammar file name. It's a common practice to keep these names similar though; for example, a Java grammar might be held in a `Java-1.5.jj` file and contain a `PARSER_BEGIN(Java)` declaration, which would result in files named with a `Java` prefix.

The first file is `LiteralsConstants.java`; it's an interface that contains integer constants for each token and lexical state that's defined in the grammar. It also contains

an array of String objects indexed by the `Token.kind` integer to provide a more readable token display string. It's short enough to display here:

Example 2.3. The tokenizer constants

 # examples/tokenizer/LiteralsConstants.java

```
1    /* Generated By:JavaCC: Do not edit this line. LiteralsConstants.java */
2
3    /**
4     * Token literal values and constants.
5     * Generated by org.javacc.parser.OtherFilesGen#start()
6     */
7    public interface LiteralsConstants {
8
9      /** End of File. */
10     int EOF = 0; 1
11     /** RegularExpression Id. */
12     int HELLO = 1;
13
14     /** Lexical state. */
15     int DEFAULT = 0;  2
16
17     /** Literal token values. */
18     String[] tokenImage = { 3
19       "<EOF>",
20       "\"hello\"",
21     };
22
23   }
```

The various parts of the constants file are:

1. The tokens. There's a definition for an `EOF` (end of file) token even though we didn't put one in the grammar; it's generated automatically by JavaCC. These constant values are not meant to be relied upon by code external to the tokenizer since they can change during future tokenizer regenerations.
2. The lexical states. This `DEFAULT` definition for the default lexical state; it's inserted automatically by JavaCC as well
3. The token images. These are used for displaying the tokens in case of errors.

Next up is `LiteralsTokenManager.java`. This contains the tokenizer code itself—it's long and convoluted as generated code tends to be. While it's interesting to see how JavaCC represents a particular grammar in the generated code, at this point it's safe to treat it as a black box.

`SimpleCharStream.java` contains code to read in characters from any `java.io.InputStream` or `java.io.Reader` implementation. It's responsible for tracking line and column numbers and providing that information to the tokenizer. It uses a two kilobyte buffer to make reading data more efficient.

Next is `Token.java`, which, no surprise, contains the definition of the `Token` class. A token object contains a `kind` integer field that refers to one of the constants defined in `LiteralsConstants.java`. It also holds a String containing the "image" of the token—that is, the characters which compose the token. In addition, it contains the line/column information that the `CharStream` calculated, a reference to the next `Token` object, and finally, in certain cases, a reference to a "special token."

Last, there are two error classes: `TokenMgrError.java` and `ParseException.java`. `TokenMgrError.java` contains a `java.lang.Error` subclass that's used to signal vari-

ous errors that occurred in the tokenizing process. `ParseException.java` contains a subclass of `java.lang.Exception` that's used to signal parsing problems. `ParseException` contains some fields that provide a useful context for parsing errors, such as the last token that was successfully parsed and the tokens that the tokenizer expected to see. It also contains code for reproducing the input that caused the parsing error. For those who are interested, both `ParseException` and `TokenMgrError` contain extensive comments. In this chapter, we're going to focus on the tokenizer, so we won't see much of `ParseException`.

Just to make sure the examples we've seen so far actually work, here's a small program to run the tokenizer:

Example 2.4. A Driver For the Tokenizer

```
# examples/tokenizer/LiteralsRunner.java

1    import java.io.*;
2    public class LiteralsRunner {
3      public static void main(String[] args) {
4        StringReader r = new StringReader(args[0]);
5        SimpleCharStream s = new SimpleCharStream(r);
6        LiteralsTokenManager ltm = new LiteralsTokenManager(s);
7        Token t = ltm.getNextToken();
8        System.out.println("Got a '" + t.image + "' token");
9      }
10   }
```

And the driver program in action:

```
$ javacc literals_plain.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_plain.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsRunner "hello"
Got a 'hello' token
```

Looks like things are working as expected! Next we'll take a closer look at the lexical specification itself.

Tokens and Regular Expressions

A regular expression is a concise description of a set of strings. When JavaCC generates the tokenizer, it turns the regular expressions in the lexical specification into code that quickly and accurately matches the incoming data to produce `Token` objects.

Regular expressions types

Regular expressions are best described by example, so Table 2.1 [22] shows the various regular expression types that JavaCC supports, along with an example of each.

Table 2.1. Regular expressions in JavaCC

Name	Example	Description
Literal	"a"	Matches only an "a"
Character class	["a","b","c"]	Either an "a", a "b", or a "c"
Ranged character class	["a"-"z"]	All lowercase letters
Negation	~["a"]	Anything single character other than an "a"
Repetition	("a"){4}	Matches 4 "a"s
Repetition ranges	("a"){2,4}	Matches at least 2 but not more than 4 "a"s
One or more items	("a")+	At least one "a"
Zero or one items	("a")?	Either zero or one "a"
Zero or more items	("a")*	Any number of "a"s (including none)

Let's work through a series of grammars that show each type of regular expression in action. This may feel like overkill, but it will familiarize you with each type and provide opportunities to discuss related matters.[1]

Literals

First, let's look closely at a modified version of the `literals_plain.jj` JavaCC grammar from page 19. Remember, this grammar's purpose is to tokenize a simple string literal, `hello`:

Example 2.5. Revisiting the `hello` tokenizer

```
# examples/tokenizer/literals.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Literals)
5    import java.io.*;
6    public class Literals {}
7    PARSER_END(Literals)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       LiteralsTokenManager mgr = new LiteralsTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           debugStream.println("Found token:" + t.image);
16       }
17     }
18   }
19   TOKEN : {
20     <HELLO : "hello">
21   }
```

We still needed to provide a `PARSER_BEGIN`/`PARSER_END` section along with a `Literals` class definition since this section is required by JavaCC. Also, notice that we're not using `System.out.println` to display the string of characters that the token matched (that is, the `image` field on the `Token` object). That's because JavaCC generates a public `debugStream` field into the tokenizer; this is an instance of

[1]For a much more detailed treatment of regular expressions, pick up Jeffrey Friedl's excellent book, "Mastering Regular Expressions", published by O'Reilly and Associates.

`java.io.PrintStream` and is set to `System.out` by default. This is a level of indirection that can come in handy, so we'll use it to print any debug messages.

This time we've used the `TOKEN_MGR_DECLS` section to embed a `main` method inside the grammar file rather than using an external driver file. This does add a bit of noise to the grammar, but it's contained in one area and, after seeing a few of these, you won't notice.

The `main` method reads the first parameter passed on the command line. It wraps this in a `StringReader` and passes that into the default JavaCC character stream implementation—a `SimpleCharStream` object. It then sends the `SimpleCharStream` to `LiteralsTokenManager` and begins a loop. The loop calls `getNextToken` repeatedly until it hits an *end of file* (`EOF`) token and exits.

We're testing for an end of file token by checking the `kind` field on the `Token` object; `kind` is set when the `Token` object is created by the tokenizer. JavaCC generates a `LiteralsTokenManager` that implements the `LiteralsConstants` interface, so we can just use the identifier `EOF` rather than the fully-qualified name `LiteralsConstants.EOF`. Note that this "implement the interface to bring in the namespace" is not really a recommended coding pattern and may well be replaced with the use of static imports in a later release of JavaCC.

The one `HELLO` literal token is defined using a `TOKEN` regular expression production; when a series of characters that matches this regular expression is encountered, JavaCC will create a `Token` object.

Here's a sample run showing the `Literals` tokenizer properly recognizing the word `hello` as a `HELLO` token:

```
$ javacc literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsTokenManager hello
Found token:hello
```

That's all well and good, but what happens if we say hello several times?

Example 2.6. Encountering a lexical error

```
$ java LiteralsTokenManager "hello hello"
Found token:hello
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 6.
Encountered: " " (32), after : ""
 at LiteralsTokenManager.getNextToken(LiteralsTokenManager.java:242)
 at LiteralsTokenManager.main(LiteralsTokenManager.java:12)
```

Our first real JavaCC error message; savor the moment! As you read the error message, you'll see that JavaCC has identified the problem quite nicely: there's a "lexical error", which makes sense because we're in the tokenizing, or lexical analysis stage. The problem cropped up at line one, column six, which is understandable since the first unrecognized character was a space, and the first space in the input appears after

the fifth character (the `o` in `hello`). JavaCC even goes so far as to say exactly which character caused the problem, including the ASCII code (32) of the culprit.

This error message, however, does have one flaw. It says that the error occurred `after : ""`. That's not very helpful. Wouldn't it be better to say `after : "o"` so we'd know exactly what the last "good" character was?

To understand this, consider that the tokenizer first read one character, an `h`, from the stream of characters provided by `SimpleCharStream`. It determined that there was a possible match in the `HELLO` token definition. It continued this process with the second, third, fourth, and fifth characters and then determined that it had found a complete token match. It created a new `Token` object, set the `Token.kind` field to `LiteralsConstants.HELLO`, and moved on to the next character in the input stream. That's when it hit the space, which was an unknown character. It had not read any characters except the one it couldn't recognize; thus, there was nothing to display in the error message.

To demonstrate this, look at what happens when we enter an intentional partial match. In the example below, we're feeding in `helper`. The first few characters, `hel`, can be matched successfully by the `HELLO` token definition. But when JavaCC hits the `p` character and finds that it has no token definitions that begin with the characters `help`, the expected error occurs:

```
$ java LiteralsTokenManager helper
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 4.
Encountered: "p" (112), after : "hel"
 at LiteralsTokenManager.getNextToken(LiteralsTokenManager.java:242)
 at LiteralsTokenManager.main(LiteralsTokenManager.java:11)
```

This time the tokenizer matched the first few characters of a token (`hel`), so those characters appear in the error message.

As mentioned earlier, more details on error handling, including some pointers on error recovery, are in chapter 7.

Character Classes

The next kind of regular expression type, the *character class*, provides a way to match one of several characters. The grammar below shows a token definition that matches either an `a`, a `b`, or a `c`. To save space we'll not display the `main` method as it's more or less the same as the one in `literals.jj`; of course, the source code for this book contains the entire program:

Example 2.7. Tokenizing with a character class

```
# examples/tokenizer/character_class.jj (lines 19 to 21)

19   TOKEN : {
20     <A_OR_B_OR_C : ["a", "b", "c"]>
21   }
```

This time we'll feed in one character at a time:

```
$ javacc character_class.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file character_class.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java CharacterClassTokenManager "a"
Found token:a
$ java CharacterClassTokenManager "b"
Found token:b
$ java CharacterClassTokenManager "c"
Found token:c
```

As expected, passing in each different member of the character class resulted in a match of the same token definition. If we pass in an unexpected character, such as a "d", the expected lexical error results:

```
$ java CharacterClassTokenManager "d"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
 at CharacterClassTokenManager.getNextToken(CharacterClassTokenManager.java:227)
 at CharacterClassTokenManager.main(CharacterClassTokenManager.java:11)
```

Character classes can contain any character and you can mix and match characters as necessary; to JavaCC, the character class `["a","!","1"]` provides a perfectly valid set of options. Character classes are case-sensitive by default, so you'll need both `["a"]` and `["A"]` if you want to catch both upper and lowercase versions.

Ranged Character Classes

If you want to match any digit you could list each digit in a character class: `["0","1","2","3","4","5","6","7","8","9"]`. That would be tedious, and doing the same thing for every letter would be even more painful. Instead, you can use a *ranged character class* to specify an upper and lower bound on a range of characters. Here's an example that matches the letters from `a` to `c`:

Example 2.8. A ranged character class

```
# examples/tokenizer/ranged_character_class.jj (lines 19 to 21)

19   TOKEN : {
20     <A_TO_C : ["a"-"c"]>
21   }
```

Here it is in action, first with a character within the expected range, then with one outside that range:

```
$ javacc ranged_character_class.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file ranged_character_class.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RangedCharacterClassTokenManager "a"
Found token:a
$ java RangedCharacterClassTokenManager "d"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
 at RangedCharacterClassTokenManager.getNextToken
(RangedCharacterClassTokenManager.java:227)
 at RangedCharacterClassTokenManager.main
(RangedCharacterClassTokenManager.java:11)
```

As expected, it happily accepted `a` but raised a lexical error on `d`.

It's time to introduce another regular expression operator—*alternation*. Alternation indicates a choice in the regular expression. In the example below, the `A_TO_C_OR_X_TO_Z` token can be just what it says—either a character between `a` and `c` or a character between `x` and `z`. The character classes are separated by the alternation operator (a pipe symbol, `|`) to indicate a choice between alternatives:

Example 2.9. Alternation and ranges

```
# examples/tokenizer/ranged_character_class_alternation.jj (lines 19 to 21)

19   TOKEN : {
20     <A_TO_C_OR_X_TO_Z: ["a"-"c"] | ["x"-"z"] >
21   }
```

Here's the grammar above handling an input character in each of its ranges:

```
$ javacc ranged_character_class_alternation.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file ranged_character_class_alternation.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RangedCharacterClassTokenManager "c"
Found token:c
$ java RangedCharacterClassTokenManager "y"
Found token:y
```

These ranges are also case-sensitive by default, so that `["a"-z"]` and `["A"-"Z"]` are both necessary if you want to catch both upper and lowercase English letters.

You can mix a character class and a ranged character class; for example, if you want to catch `a`, `c` and `x` to `z`:

Example 2.10. Mixing character classes and ranged character classes

```
# examples/tokenizer/ranged_character_class_mix.jj (lines 19 to 21)

19   TOKEN : {
20     <A_OR_C_OR_X_TO_Z: ["a","c","x"-"z"] >
21   }
```

Here are the (unsurprising) results when we enter a few expected characters and then an unexpected one:

```
$ javacc ranged_character_class_mix.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file ranged_character_class_mix.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RangedCharacterClassMixTokenManager "a"
Found token:a
$ java RangedCharacterClassMixTokenManager "y"
Found token:y
$ java RangedCharacterClassMixTokenManager "b"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
```

```
 at RangedCharacterClassMixTokenManager.getNextToken
(RangedCharacterClassMixTokenManager.java:227)
 at RangedCharacterClassMixTokenManager.main
(RangedCharacterClassMixTokenManager.java:11)
```

Negation

You can also specify *negation* regular expressions which accept anything other than the specified character or characters. The grammar below will accept anything except an `a`; it expresses this negative match using the tilde character (`~`):

Example 2.11. Negation grammar

```
# examples/tokenizer/negation.jj (lines 19 to 21)

19   TOKEN : {
20     <ANYTHING_BUT_A : ~["a"]>
21   }
```

And in action:

```
$ javacc negation.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file negation.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java NegationTokenManager "b"
Found token:b
$ java NegationTokenManager "a"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
 at NegationTokenManager.getNextToken(NegationTokenManager.java:237)
 at NegationTokenManager.main(NegationTokenManager.java:11)
```

Negation works fine with character ranges, too. Here's a grammar that accepts any characters other than those in the range of `a` to `c`:

Example 2.12. Negation ranges grammar

```
# examples/tokenizer/negation_range.jj (lines 19 to 21)

19   TOKEN : {
20     <ANYTHING_BUT_A_TO_C : ~["a"-"c"]>
21   }
```

Since seeing is believing, here's that same grammar accepting an `x` but not a `b`:

```
$ javacc negation_range.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file negation_range.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java NegationRangeTokenManager "x"
Found token:x
$ java NegationRangeTokenManager "b"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
```

```
 at NegationRangeTokenManager.getNextToken(NegationRangeTokenManager.java:237)
 at NegationRangeTokenManager.main(NegationRangeTokenManager.java:11)
```

Repetition

One way to specify three consecutive `a` characters is to repeat them, e.g., `<THREE_A : "a" "a" "a">`. A cleaner method is to use *repetition* in the regular expression. Here's how to accept `aaa` but not accept either `bbb` or `aa`:

Example 2.13. Repetition grammar

```
# examples/tokenizer/repetition.jj (lines 19 to 21)

19   TOKEN : {
20     <THREE_A : ("a"){3} >
21   }
```

And this grammar in action accepting input of `aaa` but rejecting `aab` and `aa`:

```
$ javacc repetition.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file repetition.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RepetitionTokenManager "aaa"
Found token:aaa
$ java RepetitionTokenManager "aab"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 4.
Encountered: <EOF> after : "aab"
 at RepetitionTokenManager.getNextToken(RepetitionTokenManager.java:235)
 at RepetitionTokenManager.main(RepetitionTokenManager.java:11)
$ java RepetitionTokenManager "aa"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 3.
Encountered: <EOF> after : "aa"
 at RepetitionTokenManager.getNextToken(RepetitionTokenManager.java:235)
 at RepetitionTokenManager.main(RepetitionTokenManager.java:11)
```

The tokenizer code generated is exactly the same whether you use a repetition operator or specify the expression multiple times. Using the repetition operator makes the grammar shorter and cleaner, and it leaves the door open in case JavaCC optimizes this case in future versions.

If you need to specify a repetition range with an upper and lower bound, JavaCC handles that too:

Example 2.14. Repetition range grammar

```
# examples/tokenizer/repetition_range.jj (lines 19 to 21)

19   TOKEN : {
20     <TWO_TO_FOUR_A : ("a"){2,4} >
21   }
```

And the expected results: it accepts three consecutive `a` characters but rejects five of them:

```
$ javacc repetition_range.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file repetition_range.jj . . .
```

```
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RepetitionRangeTokenManager "aaa"
Found token:aaa
$ java RepetitionRangeTokenManager "aaaaa"
Found token:aaaa
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 6.
Encountered: <EOF> after : ""
 at RepetitionRangeTokenManager.getNextToken
(RepetitionRangeTokenManager.java:245)
 at RepetitionRangeTokenManager.main(RepetitionRangeTokenManager.java:12)
```

There's a sublety in this particular tokenizer error. The tokenizer recognized the first four `a` characters as a valid instance of the `TWO_TO_FOUR_A` token, but then found only one `a` character after that, and thus couldn't form another token. Had we entered six `a` characters instead, we'd get two instances of the `TWO_TO_FOUR_A` token; the first having four `a` characters and the second containing only two `a` characters.

Quantifiers

Next we'll look at *quantifiers*. Quantifiers are operators that specify a count for the pattern preceding the quantifier. There are three types of quantifiers: "one or more", "zero or one", and "zero or more."

The first quantifier we'll look at is the `+` quantifier, which matches one or more items. Here's how it looks:

Example 2.15. Quantifier grammar—one or more

```
# examples/tokenizer/quantifier_oneormore.jj (lines 19 to 21)

19   TOKEN : {
20     <ONE_OR_MORE_A : ("a")+ >
21   }
```

This matches `a` or `aaa`, but not `b`:

```
$ javacc quantifier_oneormore.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file quantifier_oneormore.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java QuantifierOneOrMoreTokenManager "a"
Found token:a
$ java QuantifierOneOrMoreTokenManager "aaa"
Found token:aaa
$ java QuantifierOneOrMoreTokenManager "b"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: <EOF> after : ""
 at QuantifierOneOrMoreTokenManager.getNextToken
(QuantifierOneOrMoreTokenManager.java:229)
 at QuantifierOneOrMoreTokenManager.main(QuantifierOneOrMoreTokenManager.java:11)
```

The `+` quantifier (and the others as well) support matching entire expansions, not just single characters. You could specify an expansion like `("a" ["a"-"c"])+` to match one or more sequences of an `a` followed a character between `a` and `c`:

Example 2.16. More complicated quantifier grammar

```
# examples/tokenizer/quantifier_oneormore_complex.jj (lines 19 to 21)

19   TOKEN : {
20     <COMPLEX : ("a" ["a"-"c"])+ >
21   }
```

This expression matches `aa` as well as `abacaa`:

```
$ javacc quantifier_oneormore_complex.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file quantifier_oneormore_complex.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java QuantifierOneOrMoreComplexTokenManager "aa"
Found token:aa
$ java QuantifierOneOrMoreComplexTokenManager "abacaa"
Found token:abacaa
```

These quantifiers are all "greedy"; that is, they match as many characters as possible. So, a quantifier regular expression like `["a"-"z"]+` will match all the way to the end of the input data if it consists solely of lowercase characters. Some regular expression engines implement "possessive" quantifiers, which provide a performance optimization at the cost changing the match behavior; JavaCC is not such an engine.

Next up is the `?` quantifier, which matches zero or one occurrence of the specified pattern. This is more useful than it may appear. You could, for example, use it to match a single optional plus or minus sign followed by a digit:

Example 2.17. Quantifier grammar—zero or one

```
# examples/tokenizer/quantifier_zeroorone.jj (lines 19 to 21)

19   TOKEN : {
20     <SIGNED_DIGIT : (["+","-"])? ["0"-"9"]>
21   }
```

This can match `+5`, or `-9`, or you can omit the sign altogether:

```
$ javacc quantifier_zeroorone.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file quantifier_zeroorone.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java QuantifierZeroOrOneTokenManager "+5"
Found token:+5
$ java QuantifierZeroOrOneTokenManager "-9"
Found token:-9
$ java QuantifierZeroOrOneTokenManager "4"
Found token:4
```

Finally, there's the `*` quantifier, which matches zero or more occurrences of a given pattern. You can, for example, use this to match any number of digits:

Example 2.18. Quantifier grammar—zero or more

```
# examples/tokenizer/quantifier_zeroormore.jj (lines 19 to 21)

19   TOKEN : {
20     <SIGNED_NUMBER : ["+","-"] (["0"-"9"])* >
21   }
```

This will work with a sign followed by any number of digits—including none:

```
$ javacc quantifier_zeroormore.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file quantifier_zeroormore.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java QuantifierZeroOrMoreTokenManager "+42"
Found token:+42
$ java QuantifierZeroOrMoreTokenManager "-987654321"
Found token:-987654321
$ java QuantifierZeroOrMoreTokenManager "+"
Found token:+
```

Of course, you can't use this particular regular expression with a number with a decimal point since the token definition doesn't include any provisions for the . character:

```
$ java QuantifierZeroOrMoreTokenManager "+3.14159"
Found token:+3
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 3.
Encountered: "." (46), after : ""
 at QuantifierZeroOrMoreTokenManager.getNextToken
(QuantifierZeroOrMoreTokenManager.java:235)
 at QuantifierZeroOrMoreTokenManager.main
(QuantifierZeroOrMoreTokenManager.java:12)
```

As usual, JavaCC will do what you say, not what you mean!

Defining Multiple Tokens

So far we've focused on illustrating various regular expression features on a single token definition. You can define as many tokens as you want, however, by separating them by the alternation operator (|):

Example 2.19. Multiple tokens

```
# examples/tokenizer/multiple_tokens.jj (lines 19 to 23)

19   TOKEN : {
20     <HELLO : "hello">
21     | <THERE : "there">
22     | <WORLD : "world">
23   }
```

Now we can feed in a series of words and have them separated into the proper tokens:

```
$ javacc multiple_tokens.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file multiple_tokens.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java MultipleTokensTokenManager "hellohellothereworld"
Found token:hello
Found token:hello
Found token:there
Found token:world
```

What happens if there is more than one possible match? Two rules apply to that case. The first rule is the *maximal munch* rule, which states that JavaCC will match the token that consumes the largest amount of input data. The second rule states that if several tokens match the same length of data, JavaCC will select the token that appears first in the lexical specification.

Maximal Munch

To illustrate the maximal munch rule, consider the following grammar. It matches either `hello` or a series of one or more lowercase letters. We'll show the `main` method in this case since it's slightly different; it checks `Token.kind` before displaying an appropriate message:

Example 2.20. Maximal munch

```
# examples/tokenizer/longest_match.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(LongestMatch)
5    import java.io.*;
6    public class LongestMatch {}
7    PARSER_END(LongestMatch)
8    TOKEN_MGR_DECLS : {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       LongestMatchTokenManager mgr = new LongestMatchTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           debugStream.println("Found token: " + t.image + " (type " + (t.kind
== HELLO ? "<HELLO>" : "<LETTERS>")  +")");
16       }
17     }
18   }
19   TOKEN : {
20     <HELLO : "hello">
21     | <LETTERS : (["a"-"z"])+ >
22   }
```

Given the string `helloworld`, the tokenizer will choose to create a `LETTERS` token since that consumes the most input data:

```
$ java LongestMatchTokenManager "helloworld"
Found token: helloworld (type <LETTERS>)
```

First Match

To demonstrate the "first matching token definition" rule, let's use the same grammar and pass in data that either token definition could match: e.g., `hello`:

```
$ java LongestMatchTokenManager "hello"
Found token: hello (type <HELLO>)
```

As expected, JavaCC selected the HELLO token definition since it was the first matching token that appeared in the grammar.

Redundant Tokens

JavaCC will warn you when a grammar attempts to specify redundant token definitions. Suppose you have a grammar that defines first a token for hello possibly followed by world, and then another token for hello:

Example 2.21. "Cannot be matched" grammar

```
# examples/tokenizer/never_match.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(NeverMatch)
5    public class NeverMatch {}
6    PARSER_END(NeverMatch)
7    TOKEN : {
8      <HELLO_MAYBE_WORLD : "hello" ("world")? >
9      | <HELLO : "hello">
10   }
```

JavaCC will generate a warning when it processes this grammar:

```
$ javacc never_match.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file never_match.jj . . .
Warning:  "hello" cannot be matched as a string literal token at line 9, column
5. It will be matched as  <HELLO_MAYBE_WORLD>.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

As the warning states, the regular expression that defines the HELLO token will never match! Any hello strings that the tokenizer encounters will be matched by the HELLO_MAYBE_WORLD token's regular expression. In effect, the HELLO_MAYBE_WORLD definition is masking the HELLO definition.

The fix for this warning is simple; move the HELLO token definition above the HELLO_MAYBE_WORLD definition. Again, when more than one token definition can match, JavaCC will use the one that's defined first in the grammar. So this will do the trick:

Example 2.22. Fixing the "cannot match" grammar

```
# examples/tokenizer/first_match.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(FirstMatch)
5    public class FirstMatch {}
6    PARSER_END(FirstMatch)
7    TOKEN : {
8      <HELLO : "hello">
9      | <HELLO_MAYBE_WORLD : "hello" ("world")? >
10   }
```

And now the warning goes away:

```
$ javacc first_match.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file first_match.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

Fixing this problem is one of the permathreads on the JavaCC mailing list, and quite a few tokenizer questions eventually boil down to a need to rearrange the token definitions.

Private Token Definitions

Some regular expressions are more common than others and show up several times in a single lexical specification. For example, `["a"-"z","A"-"Z"]` for upper and lowercase letters and `["0"-"9"]` for digits tend to crop up repeatedly. You can improve a grammar that repeats these constructs by using *private regular expressions*. Private regular expressions can only be used as part of another token; they are not token definitions in themselves. For example, if we had a `ID` token that consisted of a letter, a number, and another letter, we could define a private regular expression for that ID by preceding it with an octothorpe, also known as the "pound sign" (`#`):

Example 2.23. Private token definition grammar

```
# examples/tokenizer/private_regex.jj (lines 19 to 23)

19   TOKEN : {
20     <ID : <LETTER> <DIGIT> <LETTER> >
21     | <#LETTER : ["a"-"z","A"-"Z"] >
22     | <#DIGIT  : ["0"-"9"] >
23   }
```

Composing a token that uses private regular expressions is just like using the conventional regular expresssions. Thus, this grammar accepts the expected input like `a1A` but rejects consecutive letters like `xyzzy`:

```
$ javacc private_regex.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file private_regex.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PrivateRegexTokenManager "a1A"
Found token:a1A
$ java PrivateRegexTokenManager "xyzzy"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: "y" (121), after : "x"
 at PrivateRegexTokenManager.getNextToken(PrivateRegexTokenManager.java:235)
 at PrivateRegexTokenManager.main(PrivateRegexTokenManager.java:11)
```

Judicious use of private regular expressions can make your grammar more readable, and you'll frequently see them in large grammars. For example, the Java language grammar example that comes with JavaCC includes this (rather long) private regular expression:

Example 2.24. DECIMAL_EXPONENT token definition

```
< #DECIMAL_EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
```

`DECIMAL_EXPONENT` apears four times in the lexical specifications, and seeing that token name rather than the regular expression makes the grammar much easier to read. In addition, the generated code is virtually identical so there's no performance hit.

As a final note, private token definitions cannot contain lexical actions. This is rarely a problem as private token definitions are most frequently used inside another token definition where you can insert a lexical action if you're so inclined.

The SKIP Regular Expression Production

Now we'll look at another regular expression production, `SKIP`. `SKIP` token definitions are used to discard unwanted data. Remember our "hello hello" example back on page 23? We got an error when we tried to tokenize "hello hello" because our lexical specification didn't know what to do with a space character. Of course, one way to specify tokens with spaces between them is to add an explicit token definition for spaces, e.g.:

Example 2.25. Explicit space token grammar

 # examples/tokenizer/literals_space_token.jj (lines 19 to 22)

```
19   TOKEN : {
20     <SPACE : " ">
21     | <HELLO : "hello">
22   }
```

Here's it in action, showing which tokens are spaces:

```
$ javacc literals_space_token.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_space_token.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsSpaceTokenTokenManager "hello hello hello"
Found token:hello
Found token: (space)
Found token:hello
Found token: (space)
Found token:hello
```

Now we have three `HELLO` tokens with `SPACE` tokens interspersed. It works, but the whole point of feeding the data through JavaCC is to discard irrelevant characters such as spaces. If we use a `SKIP` token definition, we can drop the data we don't need. We can add this new regular expression production definition before our normal token definitions:

Example 2.26. SKIP token grammar

```
# examples/tokenizer/literals_skip_named.jj (lines 19 to 24)

19   SKIP : {
20     <SPACE : " ">
21   }
22   TOKEN : {
23     <HELLO : "hello">
24   }
```

Then we can regenerate the tokenizer and rerun the sample input to make sure that the `SKIP` regular expression production is working as expected:

```
$ javacc literals_skip_named.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_skip_named.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsSkipNamedTokenManager "hello hello hello"
Found token:hello
Found token:hello
Found token:hello
```

As expected, we get our three `HELLO` tokens and the spaces are discarded.

Since we're just discarding the spaces we really don't need to explicitly name the `SPACE` token. Instead, we can use a `SKIP` definition with the space character but without a name. JavaCC will take care of tracking the token identifiers for us:

Example 2.27. SKIP token definition without named token

```
# examples/tokenizer/literals_skip.jj (lines 19 to 24)

19   SKIP : {
20     " "
21   }
22   TOKEN : {
23     <HELLO : "hello">
24   }
```

Recall from page 19 that token identifiers are generated into a "constants" interface that's named after the class name that's specified in the `PARSER_BEGIN`/`PARSER_END` section. In this case, the class name is `LiteralsSkipConstants.java` and you can see that no integer identifier is generated for the space character:

Example 2.28. SKIP token grammar constants file

```
# examples/tokenizer/LiteralsSkipConstants.java

1    /* Generated By:JavaCC: Do not edit this line. LiteralsSkipConstants.java */
2
3    /**
4     * Token literal values and constants.
5     * Generated by org.javacc.parser.OtherFilesGen#start()
6     */
7    public interface LiteralsSkipConstants {
8
9      /** End of File. */
10     int EOF = 0;
11     /** RegularExpression Id. */
12     int HELLO = 2;
13
14     /** Lexical state. */
15     int DEFAULT = 0;
16
17     /** Literal token values. */
18     String[] tokenImage = {
19       "<EOF>",
20       "\" \"",
21       "\"hello\"",
22     };
23
24   }
```

When run, this tokenizer behaves the same as before:

```
$ javacc literals_skip.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_skip.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsSkipTokenManager "hello hello hello"
Found token:hello
Found token:hello
Found token:hello
```

Unless you need to refer to a `SKIP`'d regular expression production later in your grammar, letting JavaCC handle simple `SKIP` token names internally is a good technique for keeping your grammar concise.

You can use `SKIP` to discard any characters, not just whitespace. Here's a grammar that skips uppercase letters. Note that since it's a more complex regular expression, we need to give it an explicit token name; if we don't, JavaCC won't be able to parse the grammar file.

Example 2.29. SKIPing uppercase letters

```
# examples/tokenizer/skip_uppercase.jj (lines 19 to 24)

19   SKIP : {
20     <NO_UPPER : ["A"-"Z"] >
21   }
22   TOKEN : {
23     <HELLO : "hello">
24   }
```

Now we can wrap the `HELLO` token in uppercase characters and it'll be properly extracted:

```
$ javacc skip_uppercase.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file skip_uppercase.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java SkipUppercaseTokenManager "HIhelloHI"
Found token:hello
```

Skipping uppercase letters in this way won't work if the uppercase letters are embedded in a token definition:

```
$ java SkipUppercaseTokenManager "heHIllo"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 3.
Encountered: "H" (72), after : "he"
 at SkipUppercaseTokenManager.getNextToken(SkipUppercaseTokenManager.java:336)
 at SkipUppercaseTokenManager.main(SkipUppercaseTokenManager.java:11)
```

That's because the tokenizer had already begun to match the `HELLO` token when it encountered the uppercase characters. Those uppercase characters weren't part of any token definition, so the tokenizer bailed out with an exception.

In this next example, you'll see that including a `SKIP` regular expression production that discards spaces doesn't prevent you from including spaces in a conventional `TOKEN` definition. Suppose you want a token that matches `hello world`. You can include the space in your token definition:

Example 2.30. A token definition that includes spaces

```
# examples/tokenizer/skip_greedy.jj (lines 19 to 24)

19   SKIP : {
20     " "
21   }
22   TOKEN : {
23     <HELLO_WORLD : "hello world">
24   }
```

This lexical expression will produce a single token from the input `hello world`:

```
$ java SkipGreedyTokenManager "hello world"
Found token:hello world
```

This further illustrates the "maximal munch" rule which we introduced on page 32. When the tokenizer saw the `h` character it began to match the `HELLO_WORLD` token, and once it started to consume characters it kept going until it got to the end of the token.

A further example: leading and trailing spaces will be discarded since they match the `SKIP` token definition:

```
$ java SkipGreedyTokenManager "  hello world  "
Found token:hello world
```

Skipping Single Line Comments

Recently I've been working on a grammar to parse the Log ASCII Standard file format promulgated by the Canadian Well Logging Society[2]. This file format can

[2]http://cwls.org/

include comments that start with an octothorpe (#) and extend to the end of the line, like this:

```
# This is a comment
```

Here's a SKIP regular expression production that discards these comments. Notice the flow of the definition. We start with an opening octothorpe, than we follow with a negation expansion for the characters that don't end the match. This negation expansion is enclosed in a "zero or more" expansion since a # followed by an end of line sequence is a valid comment. The definition ends with an expansion containing the possible end of line sequences.

```
SKIP : {
  <SINGLE_LINE_COMMENT : "#" (~["\n", "\r"])* ("\r" | "\n" | "\r\n") >
}
```

A further wrinkle on this SKIP production is that some languages (for example, Java) allow a single line comment to end a file without that comment ending with a newline character. If the language you're parsing allows for that, just change that last production to be optional, e.g., `("\n" | "\r" | "\r\n")?`.

Generally, SKIP is a handy and efficient regular expression production. It's useful for moving past any data that you don't need; you'll see it used in many programming language grammars to discard comments and whitespace.

The MORE Regular Expression Production

MORE is another regular expression production which, like SKIP, can be used to consume large numbers of characters in one fell swoop. However, instead of discarding those characters, it appends them to the next Token[3] that's created. It might be helpful to think of MORE as specifying a "partial token." For example, if you wanted to accept a person's name and append it to the next token, you could use something like this:

Example 2.31. MORE grammar

```
# examples/tokenizer/more_name.jj (lines 19 to 24)

19   MORE : {
20     <NAME : ["A"-"Z"] (["a"-"z"])+ >
21   }
22   TOKEN : {
23     <DONE : " is my name">
24   }
```

The MORE regular expression production definition above specifies a mandatory uppercase letter followed by at least one lowercase letter. When run, the DONE token's image will be prefixed by whatever the MORE regular expression production consumed:

```
$ java MoreNameTokenManager "Fred is my name"
Found token:Fred is my name
```

This could just as easily be done by including `["A"-"Z"] (["a"-"z"])+` in the DONE token definition. That would produce slightly different behavior, though, since the

[3]Or **SPECIAL_TOKEN**, which we'll discuss in the next page or two.

DONE token would then require a name whenever it was used. With the MORE regular expression production we can use this name prepending feature as part of any token.

You'll often see MORE used in grammars which don't want to discard certain blocks of data but don't want to do anything with them either. For example, the Java grammar that comes with JavaCC uses MORE to retain all the comment data. We'll see additional uses for MORE when we discuss lexical states starting on page 45.

The SPECIAL_TOKEN Regular Expression Production

The next regular expression production is SPECIAL_TOKEN which, like TOKEN, creates a Token object. In the case of SPECIAL_TOKEN, however, the newly-created Token is not added directly to the token stream. Instead, it's assigned to a field, specialToken, on the next Token object that's created. In other words, if you're using getNextToken() to move along through a token stream you won't see the tokens that were produced by SPECIAL_TOKEN.

This differs from MORE's behavior in that MORE simply appends the characters it consumes to the next Token. In the case of SPECIAL_TOKEN, the characters are consumed, placed in a normal token, and then added to the specialToken field of the next "real" Token. You can use SPECIAL_TOKEN to create and attach any number of Token objects to the next "real" Token.

Here's an example of using SPECIAL_TOKEN to build Token objects for all the a characters prior to a b:

Example 2.32. SPECIAL_TOKEN grammar

```
# examples/tokenizer/special_a.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(SpecialA)
5    import java.io.*;
6    public class SpecialA {}
7    PARSER_END(SpecialA)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       SpecialATokenManager mgr = new SpecialATokenManager(scs);
13       Token t = mgr.getNextToken();
14       debugStream.println("Found token: " + t.image);
15       while (t.specialToken != null) {
16         debugStream.println("There's a special token:" +
t.specialToken.image);
17         t = t.specialToken;
18       }
19     }
20   }
21   SPECIAL_TOKEN : {
22     <A : "a" >
23   }
24   TOKEN : {
25     <B : "b">
26   }
```

In the TOKEN_MGR_DECLS section of the above grammar we're doing something slightly different. Instead of repeatedly calling getNextToken on the TokenManager object, we're just getting the first token in the stream. Then we're accessing a series of Token instances that are linked together using the specialToken field. To see it working we'll feed in several a characters followed by a b, and each character will result in the creation of a Token object:

```
$ javacc special_a.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file special_a.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java SpecialATokenManager "aaab"
Found token: b
There's a special token:a
There's a special token:a
There's a special token:a
```

SPECIAL_TOKEN turns out to be quite handy in programming language comment parsing. It lets a series of comments that precede a statement be retained and tracked without having to be explicitly listed in the syntactical portion of a grammar. You'll see more concrete examples of SPECIAL_TOKEN usage later in this chapter.

Lexical Actions

You've now seen a variety of ways to define tokens as well as several different types of tokens. At times, however, you'll want a tokenizer to do something nonstandard when it matches a certain token. Those situations are (sometimes) best handled by *lexical actions*. Lexical actions enable you to insert Java code into your lexical specification and have it run whenever a particular token definition is encountered.

Simple Lexical Actions

A lexical action doesn't have to do anything complicated. It can simply print a message as seen in the grammar below:

Example 2.33. A simple lexical action

 # examples/tokenizer/lexical_action.jj (lines 16 to 24)

```
16   SKIP : {
17     " "
18   }
19   TOKEN : {
20     <HELLO : "hello">
21       { debugStream.println("Got a HELLO token"); }
22     | <WORLD : "world">
23       { debugStream.println("Got a WORLD token"); }
24   }
```

We placed the lexical action immediately after the TOKEN definition to which it applies. We're also using a SKIP regular expression production to make our input more readable. Thus we can pass in hello hello world and get our three messages:

```
$ javacc lexical_action.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_action.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LexicalActionTokenManager "hello hello world"
Got a HELLO token
Got a HELLO token
Got a WORLD token
```

We're not limited to printing things out; we can do something that involves tracking a value throughout the tokenizing process. For example, we can tot up the number of times a particular token occurs by placing a field declaration in the `TOKEN_MGR_DECLS` section and referencing that field from a lexical action:

Example 2.34. Tracking token counts with a lexical action

 # examples/tokenizer/lexical_action_add.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(LexicalActionAdd)
5    import java.io.*;
6    public class LexicalActionAdd {}
7    PARSER_END(LexicalActionAdd)
8    TOKEN_MGR_DECLS: {
9      private static int count;
10     public static void main(String[] args) {
11       StringReader sr = new StringReader(args[0]);
12       SimpleCharStream scs = new SimpleCharStream(sr);
13       LexicalActionAddTokenManager mgr = new LexicalActionAddTokenManager
(scs);
14       while (mgr.getNextToken().kind != EOF) {}
15       debugStream.println("Got " + count + " hellos");
16     }
17   }
18   SKIP : {
19     " "
20   }
21   TOKEN : {
22     <HELLO : "hello"> { count++; }
23     | <WORLD : "world">
24   }
```

And to see it in action:

```
$ javacc lexical_action_add.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_action_add.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LexicalActionAddTokenManager "hello hello world"
Got 2 hellos
```

If you delve a bit deeper and look at the generated tokenizer code, you'll see that the lexical action is inserted into the `LexicalActionAddTokenManager.java` file in a `switch` statement. This means that the type of Java code you can put in a lexical

action is limited to code that's valid inside a `switch` statement. Inserting something illegal (e.g., a method definition) will result in a long JavaCC error message that looks something like this:

```
Reading from file lexical_action_add.jj . . .
org.javacc.parser.ParseException: Encountered " "public" "public "  "void" "void
"" at line 20, column 23.
Was expecting one of:
    "LOOKAHEAD" ...
    "IGNORE_CASE" ...
    [.... many lines excluded ... ]
```

Here's an additional wrinkle[4] that we see as a result of lexical actions being placed in `case` branches of `switch` statements. Consider the following grammar:

Example 2.35. Lexical actions share scope

```
# examples/tokenizer/case_scope.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(CaseScope)
5    public class CaseScope {}
6    PARSER_END(CaseScope)
7    TOKEN : {
8      <HELLO : "hello"> { int x = 2; }
9      | <WORLD : "world"> { int x = 3; }
10   }
```

Looks straightforward, and JavaCC generates a tokenizer, but when we try to compile the code we get an error:

```
$ javacc case_scope.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file case_scope.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
CaseScopeTokenManager.java:266: x is already defined in TokenLexicalActions
(Token)
                        int x = 3;
                            ^
1 error
```

The problem here is that all the `case` branches appear in the same Java basic code block. When we declare the same variable name twice in different lexical actions, it's just as if we tried to declare it twice in a method, e.g.:

```
public class Foo {
  void bar() {
    int x = 2;
    int x = 3;
  }
}
```

Obviously this results in a compilation error, and that's the situation the lexical actions are in as well.

Finally, as mentioned on page 35, lexical actions can't be attached to private token definitions.

[4] I'm indebted to Dr. Nathan Ryan for pointing this out.

Changing Tokens

In addition to invoking your own methods in lexical actions, you can also manipulate the currently matched token. For example, if you wanted to replace every occurrence of `A` with a `AB`, you can do that by manipulating the `matchedToken` field in the `TokenManager` object. `matchedToken` contains the `Token` object that's being created by the tokenizer, and you can modify the token image by changing the `image` field on that object:

Example 2.36. Transforming a token with a lexical action

```
# examples/tokenizer/lexical_action_transform.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(LexicalActionTransform)
5    import java.io.*;
6    public class LexicalActionTransform {}
7    PARSER_END(LexicalActionTransform)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       LexicalActionTransformTokenManager mgr = new
LexicalActionTransformTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           debugStream.println("Found token:" + t.image);
16       }
17     }
18   }
19   TOKEN : {
20     <A : "A"> {matchedToken.image = image.append("B").toString();}
21   }
```

Here's how this looks in action:

```
$ javacc lexical_action_transform.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_action_transform.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LexicalActionTransformTokenManager A
Found token:AB
```

A real-world application of token manipulation surfaced on the JavaCC user's mailing list. The question asked about decoding tokens that contained a construct called "run length encoding." In other words, a token would start with a digit indicating the number of characters to follow, so that `5:abcde` was a valid token and so was `2:hi`. Since JavaCC doesn't support a somewhat advanced regular expression construct known as a backreferences, this was hard to process with a normal regular expression. With a lexical action, it was straightforward to create the token with an image of `3:` and then call into the character stream to glom on the additional characters. Here's how it looks:

Example 2.37. Solving run length encoding with a lexical action

```
# examples/tokenizer/run_length_encoding.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(RunLengthEncoding)
5    import java.io.*;
6    public class RunLengthEncoding {}
7    PARSER_END(RunLengthEncoding)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       RunLengthEncodingTokenManager mgr = new RunLengthEncodingTokenManager
(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           debugStream.println("Found a token: " + t.image);
16       }
17     }
18   }
19   TOKEN : {
20     <RL_STR : ["0"-"9"] ":">
21     {
22       int length = Integer.parseInt(matchedToken.image.substring(0,1));
23       try {
24         for (int i=0; i<length; i++) {
25           matchedToken.image = matchedToken.image + input_stream.readChar();
26         }
27       } catch (IOException ioe) {
28         ioe.printStackTrace();
29       }
30     }
31   }
```

And in action:

```
$ javacc run_length_encoding.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file run_length_encoding.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java RunLengthEncodingTokenManager "2:hi5:there"
Found a token: 2:hi
Found a token: 5:there
```

This example shows how a few lines of code to decode a particularly tricky token can be quite useful. It also shows how you can consume a few more characters from the input stream which, though not usually necessary, is a nice option to have. In practice, you could choose to clean this up by moving the lexical action into a function inside `TOKEN_MGR_DECLS`.

Lexical States

Overview

Lexical states are one of the JavaCC lexical specification building blocks. They're fundamental to JavaCC, which raises a question: "If they're so fundamental, why

haven't you mentioned them earlier?" That's because all of the examples we've seen so far have defined tokens in the default lexical state. Way back on page 19 we talked about the `LiteralConstants.java` file and the fact that it had a `DEFAULT` constant defined; that constant identified the ubiquitous default lexical state. As we have seen, you can do a lot with JavaCC without defining additional lexical states, so let's see what they are and why they're important.

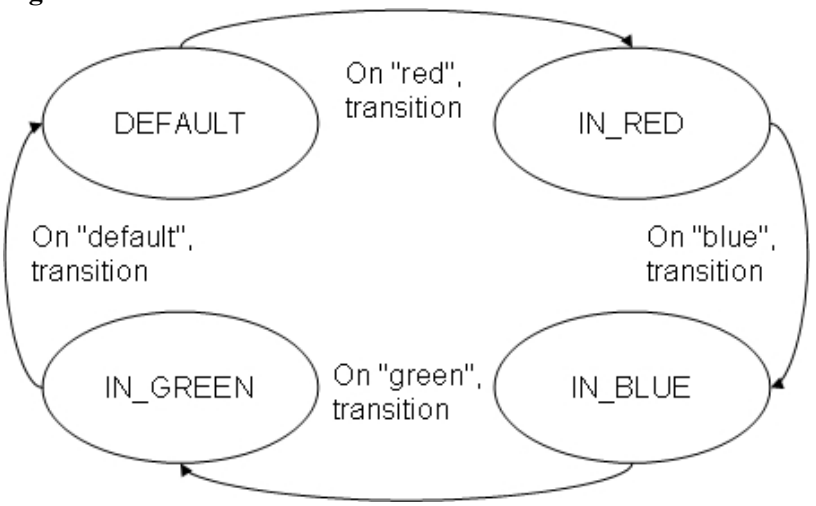
A lexical state is a container for token definitions of various kinds: `TOKEN`, `SKIP`, `MORE`, or `SPECIAL_TOKEN`. While in a particular lexical state, JavaCC will use only the regular expression productions defined in that state. This means that when the tokenizer encounters a region of data that needs to be tokenized differently than the rest, it must switch to a lexical state specifically defined for that type of data. Think of a lexical state as a diligent but narrowly focused worker who can handle a few specific responsibilities really well.

Lexical states frequently make a grammar more readable. Since the token definitions for each state need handle only items that occur within that state, a given state could potentially contain just a few relatively simple token definitions. A standard example is that of parsing Javadoc comments in a Java source file. Since those comments are a mix of freeform text and keywords (like `@param` and `@author`), a single regular expression to handle them would be quite complex. By using lexical states, we can isolate that complexity into a smaller portion of the lexical specification. Using lexical states also has a rather useful side effect—you can get a high level overview of the lexer structure by reading through the state transitions.

Changing Colors

On to a lexical state example! Let's look at a grammar that defines three lexical states: `IN_RED`, `IN_BLUE`, and `IN_GREEN`. A lexical state is defined by enclosing a name in angle brackets. The state definition is followed by the token definitions for that state. To switch from one state to another, you place the new state name after the token that triggers the change. To switch back to the `DEFAULT` state, just use its name, `DEFAULT`. Here's a diagram that illustrates these state changes:

Figure 2.2. Lexical states for different colors



Here's the grammar to produce these states and their transitions. Notice that there's a lexical action after each token definition so we can keep track of what's going on. Notice also that on line 15 we don't define a lexical state for the `RED` token definition. Why? Because the tokenizer starts in the `DEFAULT` state and the token is therefore in that state unless we indicate otherwise - although if you want you can also make this explicit by preceding the token production with `<DEFAULT>`. Finally, you can see that a lexical state definition must precede a token definition. This is a reasonable restriction since the whole purpose of a lexical state is to contain certain token definitions.

Example 2.38. Lexical state transition grammar

```
# examples/tokenizer/lexical_states_colors.jj (lines 16 to 33)

16   TOKEN : {
17     <RED: "red"> {debugStream.println("Switching to IN_RED");} : IN_RED
18   }
19
20   <IN_RED>
21   TOKEN : {
22     <BLUE : "blue"> {debugStream.println("Switching to IN_BLUE");} : IN_BLUE
23   }
24
25   <IN_BLUE>
26   TOKEN : {
27     <GREEN: "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
28   }
29
30   <IN_GREEN>
31   TOKEN : {
32     <DONE : "default"> {debugStream.println("Switching to DEFAULT");} :
DEFAULT
33   }
```

We can feed this grammar a series of color names and see the state transitions:

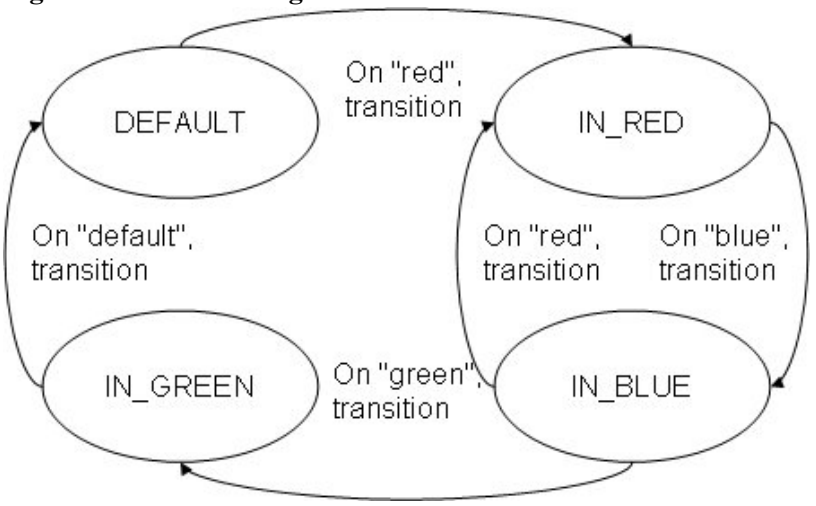
```
$ javacc lexical_states_colors.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_states_colors.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LexicalStatesColorsTokenManager "redbluegreendefault"
Switching to IN_RED
Switching to IN_BLUE
Switching to IN_GREEN
Switching to DEFAULT
```

We've only defined certain specific state transitions: `IN_RED` to `IN_BLUE` to `IN_GREEN` to `DEFAULT`. Thus a misplaced `red` will produce a lexical error:

```
$ java LexicalStatesColorsTokenManager "redbluered"
Switching to IN_RED
Switching to IN_BLUE
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 8.
Encountered: "r" (114), after : ""
 at LexicalStatesColorsTokenManager.getNextToken
(LexicalStatesColorsTokenManager.java:487)
 at LexicalStatesColorsTokenManager.main(LexicalStatesColorsTokenManager.java:11)
```

We can add another state transition easily. To allow a transition from `IN_BLUE` back to `IN_RED` just add another token definition to `IN_BLUE` along with the target state name. Here's how the state diagram looks now:

Figure 2.3. After adding a transition from blue to red



And here's the lexical specification change to the `IN_BLUE` state:

Example 2.39. Adding a new state transition

```
# examples/tokenizer/lexical_states_back_to_red.jj (lines 25 to 29)

25   <IN_BLUE>
26   TOKEN : {
27     <GREEN : "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
28     | <RED_2: "red"> {debugStream.println("Switching back to IN_RED");} :
IN_RED
29   }
```

Now we can go from `IN_RED` to `IN_BLUE` and back again:

```
$ java LexicalStatesBackToRedTokenManager "redblueredbluegreendefault"
Switching to IN_RED
Switching to IN_BLUE
Switching back to IN_RED
Switching to IN_BLUE
Switching to IN_GREEN
Switching to DEFAULT
```

In the grammar above we use a clunky name `RED_2` when defining a token for `red` in the state `IN_BLUE`. If we tried to reuse the token `RED` from the `DEFAULT` lexical state we'd get this error:

```
$ javacc lexical_states_duplicate_name.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_states_duplicate_name.jj . . .
Error: Line 19, Column 3: Multiply defined lexical token name "RED".
Detected 1 errors and 0 warnings.
```

This happens because JavaCC stores all named tokens as keys in a single `Hashtable` object. In our case it's easy to define a new token name, `RED_2`, and use that instead. Of course, defining a new name results in duplication of the string literal `red` in the grammar. That's not a big deal in this case, but if `RED` were defined by a long regular expression, it would be a shame to duplicate that regular expression in the grammar. To avoid that duplication, you can *relabel* the token like this:

Example 2.40. Relabeling a token

```
# examples/tokenizer/lexical_states_relabel.jj (lines 18 to 22)

18   <IN_BLUE>
19   TOKEN : {
20     <GREEN : "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
21     | <RED_2 : <RED>> {debugStream.println("Switching to IN_RED");} : IN_RED
22   }
```

That takes care of the token image duplication. In some cases you can consolidate the definitions even further by listing multiple lexical states for a token definition:

Example 2.41. Consolidating token definitions across lexical states

```
# examples/tokenizer/lexical_states_back_to_red_combined.jj (lines 16 to 19)

16   <DEFAULT, IN_RED>
17   TOKEN : {
18     <RED: "red"> {debugStream.println("Switching to IN_RED");} : IN_RED
19   }
```

Listing multiple lexical states for one token definition is usually only done when you want the same thing to happen when that token is encountered in each of the lexical states. When that's the usage case, however, it saves some space and makes your grammar a bit more readable.

The input data would be easier to read if we had spaces between the color names, but there's a wrinkle. Each regular expression production must be preceded by the lexical state name, so just adding a `SKIP` regular expression production definition for `IN_RED` won't work:

Example 2.42. The wrong way to SKIP spaces with multiple lexical states

```
# examples/tokenizer/this_skip_wont_work.jj

1    <IN_RED>
2    SKIP : {
3        " "
4    }
5    TOKEN : {
6        <BLUE : "blue"> {debugStream.println("Switching to IN_BLUE");} : IN_BLUE
7    }
```

The lexical specification above is telling JavaCC that the `IN_RED` state consists of a `SKIP` regular expression production that discards spaces. Since the `BLUE` token definition is not immediately preceded by a lexical state name, it's placed in the `DEFAULT` lexical state. This means that when the tokenizer is in any other state, spaces won't be discarded, and we'll see lexical errors.

One way to do this correctly is to explicitly list a `SKIP` token definition for each lexical state:

Example 2.43. SKIPing spaces the hard way

```
# examples/tokenizer/lexical_states_spaces.jj (lines 34 to 50)

34   <IN_BLUE>
35   SKIP : {
36     " "
37   }
38   <IN_BLUE>
39   TOKEN : {
40     <GREEN : "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
41   }
42
43   <IN_GREEN>
44   SKIP : {
45     " "
46   }
47   <IN_GREEN>
48   TOKEN : {
49     <DONE : "default"> {debugStream.println("Switching to DEFAULT");} :
DEFAULT
50   }
[..... etc, etc .....]
```

It's more concise to combine those `SKIP` definitions, but again, this only works if we're doing the same action in each lexical state:

Example 2.44. SKIPing spaces the easy way

```
# examples/tokenizer/lexical_states_spaces_combined.jj (lines 17 to 40)

17   <IN_RED, IN_BLUE, IN_GREEN, DEFAULT>
18   SKIP : {
19     " "
20   }
21
22   <DEFAULT, IN_BLUE>
23   TOKEN : {
24     <RED: "red"> {debugStream.println("Switching to IN_RED");} : IN_RED
25   }
26
27   <IN_RED>
28   TOKEN : {
29     <BLUE : "blue"> {debugStream.println("Switching to IN_BLUE");} : IN_BLUE
30   }
31
32   <IN_BLUE>
33   TOKEN : {
34     <GREEN : "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
35   }
36
37   <IN_GREEN>
38   TOKEN : {
39     <DONE : "default"> {debugStream.println("Switching to DEFAULT");} :
DEFAULT
40   }
```

Another way to do this is to substitute the entire `<IN_RED, IN_BLUE, IN_GREEN, DEFAULT>` lexical state name list with a shortcut notation of `<*>`. This indicates that the next regular expression production belongs to all states. The problem with this construct is that any new lexical states will automatically get that behavior, which can lead to surprises. I think it's safer to explicitly name the lexical states that you want to use with a particular definition, but if you have a lot of lexical states and a straightforward token definition this shortcut notation is an option.

So, we're back to more readable input now; with those additional state names in place, each lexical state knows to discard spaces:

```
$ java LexicalStatesSpacesTokenManager "red blue red blue green default"
Switching to IN_RED
Switching to IN_BLUE
Switching to IN_RED
Switching to IN_BLUE
Switching to IN_GREEN
Switching to DEFAULT
```

Lexical State Tips and Tricks

Reusing a Token Name

If you have a token name, such as `RED`, that you want to use in several lexical states, you might be tempted to reuse it like this:

Example 2.45. Reusing a token name the wrong way

```
# examples/tokenizer/lexical_states_free_standing.jj (lines 7 to 19)

7    TOKEN : {
8      <RED: "red"> {debugStream.println("Switching to IN_RED");} : IN_RED
9    }
10
11   <IN_BLUE>
12   SKIP : {
13     " "
14   }
15   <IN_BLUE>
16   TOKEN : {
17     <GREEN : "green"> {debugStream.println("Switching to IN_GREEN");} :
IN_GREEN
18   | <RED> {debugStream.println("Switching back to IN_RED");} : IN_RED
19   }
```

JavaCC will create a tokenizer from this grammar but will print a warning:

```
$ javacc lexical_states_free_standing.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_states_free_standing.jj . . .
Warning: Line 18, Column 3: Ignoring free-standing regular expression reference.
 If you really want this, you must give it a different label as <NEWLABEL:<RED>>.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

More importantly, the tokenizer generated from that grammar won't work as you expect. The `<RED>` regular expression reference will be ignored and so there's no way to transition directly from `IN_BLUE` back to `IN_RED`. The correct solution is to relabel the token using the technique discussed on page 49.

Token with Same Name as State

Here's another lexical state-related error situation. Suppose you try to define a token and a lexical state with the same name, like this:

Example 2.46. Names for tokens and lexical states

```
# examples/tokenizer/lexical_states_token_same_name_as_state.jj (lines 7 to 14)

7    TOKEN : {
8      <RED: "red"> : RED
9    }
10
11   <RED>
12   TOKEN : {
13     <DONE : "done"> : DEFAULT
14   }
```

You'll receive this error message:

```
$ javacc lexical_states_token_same_name_as_state.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file lexical_states_token_same_name_as_state.jj . . .
Error: Line 8, Column 3: Lexical token name "RED" is the same as that of a
lexical state.
Detected 1 errors and 0 warnings.
```

You'll also see this message if you attempt to define a token named `DEFAULT`. Remember, that name is already taken by the default lexical state. The same applies if you attempt to name a token `EOF` - that's a reserved keywords as well.

Starting the Tokenizer in a Non-Default State

As stated earlier, the tokenizer starts (by default, if you will) in the `DEFAULT` lexical state. You can change this by calling a tokenizer constructor that accepts a lexical state as a parameter. For example, suppose we have some input data that starts with a header full of arbitrary characters terminated with a `---`. We can start off in an `IN_HEADER` state and use `SPECIAL_TOKEN` to gather up the header data and attach it to the first real token:

Example 2.47. Setting an initial lexical state

 # examples/tokenizer/start_in_other_lexical_state.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(NonDefault)
5    import java.io.*;
6    public class NonDefault {}
7    PARSER_END(NonDefault)
8    TOKEN_MGR_DECLS : {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       NonDefaultTokenManager mgr = new NonDefaultTokenManager(scs, IN_HEADER);
13       while (mgr.getNextToken().kind != EOF) {}
14     }
15   }
16
17   <IN_HEADER>
18   MORE : {
19     < ~[] >
20   }
21
22   <IN_HEADER>
23   SPECIAL_TOKEN : {
24     <HEADER_NOTES: "---" > : DEFAULT
25   }
26
27   TOKEN : {
28     <GREETING: "hello">
29     { debugStream.println("Got a greeting, header was " +
matchedToken.specialToken.image);}
30   }
```

Here's this grammar tokenizing the input and displaying the header information:

```
$ java NonDefaultTokenManager "some header information, various characters, and
whatnot---hello"
Got a greeting, header was some header information, various characters, and
whatnot---
```

As the grammar is shown here the `HEADER_NOTES` token includes the `---` terminator. In practice we might want to strip that off using a lexical action (see page 44).

In the above example we initialized the `TokenManager` with the `IN_HEADER` state. Another way to get the same effect is to use the tokenizer constructor that accepts a `CharStream` and then call `TokenManager.SwitchTo` to change states.

To give credit where credit is due: these techniques for starting off in a non-default lexical state were suggested by Sreenivasa Viswanadha on the JavaCC mailing list[5].

A Custom DebugStream

JavaCC generates the tokenizer with a public `debugStream` field with a default value of standard out. You can set this field to any value you choose since JavaCC also generates a `setDebugStream` method. Here's how we can set the debug stream to write to a file:

[5]https://javacc.dev.java.net/servlets/ReadMsg?list=users&msgNo=1440

Example 2.48. Setting a Custom DebugStream

 # examples/tokenizer/debug_stream.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(DebugStream)
5    import java.io.*;
6    public class DebugStream {}
7    PARSER_END(DebugStream)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws IOException {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       DebugStreamTokenManager mgr = new DebugStreamTokenManager(scs);
13       FileOutputStream fos = new FileOutputStream("debug.txt");
14       PrintStream debugStream = new PrintStream(fos);
15       mgr.setDebugStream(debugStream);
16       while (mgr.getNextToken().kind != EOF) {}
17       debugStream.close();
18     }
19   }
20   TOKEN : {
21     <HELLO : "hello"> {debugStream.println("Found a <HELLO>!");}
22   }
```

If you feed this example a couple of strings that it accepts you'll won't get any output on the terminal window:

```
$ java DebugStreamTokenManager "hellohellohello"
```

Instead, the output has been redirected to the file, `debug.txt`:

```
$ cat debug.txt
Found a <HELLO>!
Found a <HELLO>!
Found a <HELLO>!
```

Since the code in `TOKEN_MGR_DECLS` is inside the `TokenManager` class, we could have just assigned a value to `debugStream` directly rather than using the accessor method. If you need to set this field from another class (say, the parser), however, you'll need to do that through `TokenManager.setDebugStream`.

Real World Example: Tokenizing Comments

Lexical states are frequently used for tokenizing comments in source code files. For example, one type of Java multiline comment begins with `/*` and ends with `*/`:

Example 2.49. A Java multi-line comment

```
/* This is a
 multi-line Java comment */
```

The Java grammar that comes bundled with JavaCC [6] handles multiline comments using a combination of lexical states, a `MORE` definition, and a `SPECIAL_TOKEN` definition. Here's the relevant part of the lexical specification:

[6] See **`javacc/examples/JavaGrammars/1.5/Java1.5.jj`** in the JavaCC distribution

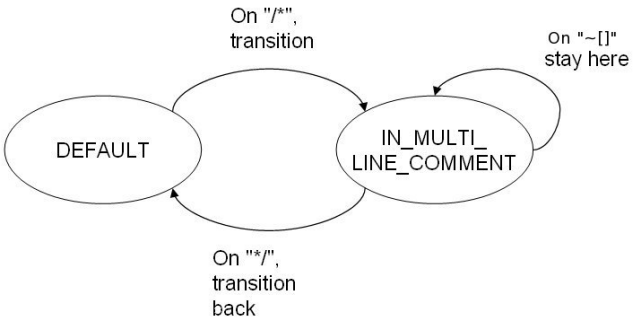
Example 2.50. Parsing Java multi-line comments with lexical states

```
# examples/tokenizer/lexical_state_java_comment.txt

1    MORE : {
2      "/*" : IN_MULTI_LINE_COMMENT
3    }
4
5    <IN_MULTI_LINE_COMMENT>
6    MORE : {
7      < ~[] >
8    }
9
10   <IN_MULTI_LINE_COMMENT>
11   SPECIAL_TOKEN : {
12     <MULTI_LINE_COMMENT: "*/" > : DEFAULT
13   }
```

And here's a diagram that may help you visualize the states and their transitions:

Figure 2.4. Lexical states for parsing comments



In lines 1-3 above the `DEFAULT` lexical state has a `MORE` token definition. This `MORE` token is defined so that when it encounters `/*` it moves into the `IN_MULTI_LINE_COMMENT` state. The tokenizer then uses another `MORE` definition to accumulate all the text (including newline characters) that it encounters. The only thing that can stop that `MORE` from consuming a character is a match to the `SPECIAL_TOKEN` definition of `*/`. When encountered, the `SPECIAL_TOKEN` pulls all the accumulated text into a `Token` object and transitions the tokenizer back into the `DEFAULT` state.

Real world example: Tokenizing Strings

One of the more complex token definitions you'll see is the `STRING_LITERAL` token as used in the Java grammar. Here it is in its entirety:

Example 2.51. STRING_LITERAL Definition

```
# examples/tokenizer/Java1.5.jj (lines 323 to 334)
```

```
323    < STRING_LITERAL:
324        "\""
325        (   (~["\"","\\","\n","\r"])
326          | ("\\"
327               ( ["n","t","b","r","f","\\","'","\""]
328               | ["0"-"7"] ( ["0"-"7"] )?
329               | ["0"-"3"] ["0"-"7"] ["0"-"7"]
330               )
331             )
332        )*
333        "\""
334    >
```

If you can understand this token definition you'll probably feel confident with just about any other definition you'll find, so let's go through this line by line to see what's happening. First we start with the token name and an opening double quote, which has to be escaped with a backslash:

```
< STRING_LITERAL:
  "\""
```

Next we see the first of two primary alternatives: "any single character other than a closing double quote, an escaped backslash character, or an end of line marker." The "any single character other than" is achieved via the negation operator, `~`. This covers all the usual text that you'd expect inside a string, including any Unicode characters:

```
  (   (~["\"","\\","\n","\r"])
```

Next comes the second alternative, which handles various escape sequences. A string can contain various types of character escapes: `\t` to indicate an escaped tab, `\b` for a backspace, and so forth, and those are enumerated here:

```
  | ("\\"
  ( ["n","t","b","r","f","\\","'","\""]
```

A string can also contain octal escape sequences like `\02` or `\253`; these must begin with a digit between zero and seven or, for a three digit octal escape, a digit between zero and three. Those are defined here:

```
    | ["0"-"7"] ( ["0"-"7"] )?
    | ["0"-"3"] ["0"-"7"] ["0"-"7"]
```

The next few lines close the parentheses of the various expansions and include a "zero or more" repetition operator. It's "zero or more" not "one or more" since `""` is a valid string literal.

```
    )
   )
  )*
"\""
>
```

For a good exercise, try removing various bits of this token definition and see what breaks. What happens, for example, if you remove the last character (the double quote) listed in the escaped character expansion?

Tokenizer Options

In this chapter we've hit upon some of the JavaCC options that affect the tokenizer. In this next section we'll explore these options and how they interact, but first let's look at the different ways JavaCC options can be set.

The first method is to set the option in the `options` header section of the grammar:

```
options {
  DEBUG_TOKEN_MANAGER=true;
  STATIC=false;
}
```

Because this is the most straightforward method for setting options in any grammars you create, it's a good default. But if you need to override those option settings, read on.

The second method is to pass the option name and (sometimes) its value on the command line. The "sometimes" comes into play because boolean valued options can have a default value, so passing a value might not be necessary. For example, `DEBUG_TOKEN_MANAGER` is `false` by default. To turn it on you can pass it in like this:

```
$ javacc -debug_token_manager=true some_grammar_file.jj
```

Or, more concisely, like this:

```
$ javacc -debug_token_manager some_grammar_file.jj
```

Here's a third way to set boolean valued options: use the option name preceded by `NO`. For example, if `DEBUG_TOKEN_MANAGER` was set to `true` in the `options` section of the grammar, you could turn it off by running JavaCC like this:

```
$ javacc -nodebug_token_manager some_grammar_file.jj
```

JavaCC will print out a helpful warning message when you override an option from the command line:

```
$ javacc -build_parser=true literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals.jj . . .
Warning: Line 2, Column 3: Command line setting of "BUILD_PARSER" modifies option
 value in file.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

If you mistype an option name, you'll get a warning for that as well:

```
$ javacc -debg_token_manger=false literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Warning: Bad option "-debg_token_manger=false" will be ignored.
Reading from file literals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

Note that option names are case-insensitive, so `-NODEBUG_TOKEN_MANAGER` and `-NOdebug_TOKEN_manager` are equivalent. Finally, in a few places I've done some perfor-

mance measurements. These should be taken with a grain of salt and then thoroughly tested in your own application; the JVM's runtime bytecode optimizations and native code translations may skew the performance measurements considerably.

With that groundwork in place, here are the options that affect the tokenizer.

BUILD_PARSER

You've already seen the `BUILD_PARSER` option in action throughout this chapter. It's a boolean valued option that's `true` by default; when set to `false` the parser portion of the grammar is not generated.

Some non-intuitive points: you still need to include the `PARSER_BEGIN/PARSER_END` section of the grammar even if this option is `false` (i.e., you're not building the parser). Also, JavaCC will generate the `ParseException.java` source file, which is only used by the parser, regardless of the `BUILD_PARSER` value.

In spite of these limitations, taking the time to set this option to `false` is a big win if you only need to tokenize a data stream. The open source project PMD[7] includes a C/C++ grammar for tokenizing files, and for a long time I left `BUILD_PARSER` at its default `true` value. When I realized that I had an option, I set it to `false` and the PMD jar file size dropped by 50 KB. No wonder—an unused 8000 line C++ parser was no longer being generated, compiled, and built into the jar file.

BUILD_TOKEN_MANAGER

The `BUILD_TOKEN_MANAGER` option is the counterpart of the `BUILD_PARSER` option. It's `true` by default, but when set to `false` it prevents the tokenizer from being generated. The JavaCC documentation suggests that this option is most useful when you've tweaked the syntactical portion of a grammar but haven't changed the lexical specification, and don't want to take the time to unnecessarily regenerate the tokenizer files. I've used this option on several large grammars; it does indeed cut generation time by about twenty percent. Considering that the total generation time is usually about a second, that's not such a big win, but it's there if you need it.

Note that `BUILD_TOKEN_MANAGER` is unrelated to the `USER_TOKEN_MANAGER` option. `USER_TOKEN_MANAGER` is used when you want to write your own tokenizer from scratch.

COMMON_TOKEN_ACTION

The `COMMON_TOKEN_ACTION` option is a boolean valued option that's `false` by default. When enabled, it tells JavaCC to insert a call to a user-defined method `CommonTokenAction` after the tokenizer reads each token. If you enable this `COMMON_TOKEN_ACTION` but forget to implement `CommonTokenAction`, JavaCC will provide a warning:

```
Warning: You have the COMMON_TOKEN_ACTION option set. But it appears you have not
 defined the method :
  void CommonTokenAction(Token t)
```

[7] http://pmd.sf.net/

You'll get a more severe message concerning the generated tokenizer from the Java compiler. Since the calls to `CommonTokenAction` have been inserted into the tokenizer but that method hasn't been defined, the tokenizer will fail to compile.

Here's an example of a tokenizer with `COMMON_TOKEN_ACTION` defined and a simple `CommonTokenAction` implementation that prints a message every time a token is encountered. We put the `CommonTokenAction` method definition inside the `TOKEN_MGR_DECLS` section:

Example 2.52. Declaring COMMON_TOKEN_ACTION

 # examples/tokenizer/literals_common_token_action.jj

```
1    options {
2      BUILD_PARSER=false;
3      COMMON_TOKEN_ACTION=true;
4    }
5    PARSER_BEGIN(Literals)
6    import java.io.*;
7    public class Literals {}
8    PARSER_END(Literals)
9    TOKEN_MGR_DECLS: {
10     public static void main(String[] args) {
11       StringReader sr = new StringReader(args[0]);
12       SimpleCharStream scs = new SimpleCharStream(sr);
13       LiteralsTokenManager mgr = new LiteralsTokenManager(scs);
14       while (mgr.getNextToken().kind != EOF) {}
15     }
16     public static void CommonTokenAction(Token t) {
17       if (t.kind == EOF) {
18         debugStream.println("Found the end of file token");
19       } else {
20         debugStream.println("Found token:" + t.image);
21       }
22     }
23   }
24   TOKEN : {
25     <HELLO : "hello">
26   }
```

When fed some test data, this produces the expected results: a few "hello"s followed by an end of file marker:

```
$ java LiteralsTokenManager hellohello
Found token:hello
Found token:hello
Found the end of file token
```

DEBUG_TOKEN_MANAGER

This is a boolean valued option that's set to `false` by default. When set to `true`, it inserts lots of debugging statements into the generated tokenizer to assist you in troubleshooting. This option roughly doubles the size of the grammar and produces such a flood of output that I usually find it most helpful when used on a small test input data set.

For an example of `DEBUG_TOKEN_MANAGER` in action, here's what the `literals.jj` tokenizer produces when created with this option set and an input string of `hello`:

Example 2.53. DEBUG_TOKEN_MANAGER in action

```
$ java LiteralsTokenManager hello
Current character : h (104) at line 1 column 1
Possible string literal matches : { "hello" }
Current character : e (101) at line 1 column 2
Possible string literal matches : { "hello" }
Current character : l (108) at line 1 column 3
Possible string literal matches : { "hello" }
Current character : l (108) at line 1 column 4
Possible string literal matches : { "hello" }
Current character : o (111) at line 1 column 5
No more string literal token matches are possible.
Currently matched the first 5 characters as a "hello" token.
****** FOUND A "hello" MATCH (hello) ******
Found token:hello
Returning the <EOF> token.
```

All this output greatly degrades a tokenizer's runtime performance, so you don't want to enable this option when you generate your production tokenizer.

IGNORE_CASE

The `IGNORE_CASE` option is boolean valued and is `false` by default. When set to `true`, this option causes the tokenizer to be case-insensitive so that `hello` and `HELLO` are both accepted by a token definition of `hello`. Here's how this looks: first, the default behavior for a token defined to catch `hello`:

```
$ javacc literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsTokenManager HELLO
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 1.
Encountered: "H" (72), after : ""
 at LiteralsTokenManager.getNextToken(LiteralsTokenManager.java:242)
 at LiteralsTokenManager.main(LiteralsTokenManager.java:11)
```

And now the same grammar with the `IGNORE_CASE` option set from the command line:

```
$ javacc -ignore_case literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsTokenManager HELLO
Found token:HELLO
```

You can set this option on an individual regular expression kind. In the grammar below, the `HELLO` token can be either upper or lower case, while `WORLD` is lowercase only:

Example 2.54. Local IGNORE_CASE

```
# examples/tokenizer/literals_ignore_case.jj (lines 19 to 24)

19   TOKEN [IGNORE_CASE] : {
20     <HELLO  : "hello">
21   }
22   TOKEN : {
23     <WORLD  : "world">
24   }
```

And here's proof:

```
$ javacc literals_ignore_case.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals_ignore_case.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsTokenManager "HELLOworld"
Found token:HELLO
Found token:world
$ java LiteralsTokenManager "helloworld"
Found token:hello
Found token:world
$ java LiteralsTokenManager "HELLOWORLD"
Found token:HELLO
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 6.
Encountered: "W" (87), after : ""
 at LiteralsTokenManager.getNextToken(LiteralsTokenManager.java:273)
 at LiteralsTokenManager.main(LiteralsTokenManager.java:12)
```

The JavaCC performance notes indicate that `IGNORE_CASE` is fastest when used globally, so avoid using it locally if possible. Also, `IGNORE_CASE` uses the `String` instance method `toUpperCase` to normalize cases for comparisons, so it generally works as well as a particular locale supports case rules.

JAVA_UNICODE_ESCAPE

The `JAVA_UNICODE_ESCAPE` option is a boolean-valued option that's set to `false` by default. This option is useful to those who are using JavaCC to parse programs that are written in Java. When set to `true`, it causes JavaCC to generate an input stream class that will decode Java's Unicode escape sequences. For example, if you have the escape sequence `\u015F` in a Java source file, the `JAVA_UNICODE_ESCAPE` option will create a input stream that will properly convert this to the Unicode character "Latin small letter s with cedilla", or ş.

This option is ignored if you set either `USER_TOKEN_MANAGER` or `USER_CHAR_STREAM` to `true`. That's reasonable, since setting either of these options indicates that you're writing your own code to transform bytes into tokens.

Processing Unicode data with JavaCC is a big topic, so I've devoted Chapter 6 to nailing it down.

JDK_VERSION

The JDK_VERSION option targets JavaCC's code generator to a specific Java language version. For example, if JDK_VERSION is set to 1.5, JavaCC generates code that uses Java 1.5 features such as generics and enumerations. The default value of JDK_VERSION is 1.4.

As of JavaCC 4.2, JDK_VERSION has no effect on the tokenizer code. In spite of this, it's not a bad idea to specify this option just in case future versions of JavaCC do introduce version-specific tokenizer code generation optimizations.

KEEP_LINE_COLUMN

KEEP_LINE_COLUMN is a boolean option that defaults to true. It controls the CharStream's collection of the line and column data (beginLine, beginColumn, endLine, endColumn) for each character.

Setting KEEP_LINE_COLUMN to false results in a decent time savings:

Table 2.2. KEEP_LINE_COLUMN Benchmark Results

Grammar	KEEP_LINE_COLUMN disabled	KEEP_LINE_COLUMN enabled	Performance gain
Java 1.5	647 ms	760 ms	15%
IDL	693 ms	936 ms	26%
C	129 ms	144 ms	11%

A big drawback, however, is that without the line and column information it may be harder to track down input data errors. Your mileage may vary.

OUTPUT_DIRECTORY

The OUTPUT_DIRECTORY option is a string-valued option that's set to the current directory by default. If you specify the OUTPUT_DIRECTORY option with a directory that does not exist, JavaCC will create that directory for you.

This option simply places the files in the indicated directory; it doesn't add a package name. When you use this option, you'll usually want to include a package declaration in the PARSER_BEGIN/PARSER_END portion of the grammar as well.

STATIC

The STATIC option is a boolean valued option that's set to true by default. If left at true it generates a tokenizer that contains only fields and methods that are static. This is primarily a performance enhancement. As you can see in the results below it's a fairly considerable win:

Table 2.3. STATIC Benchmark Results

Grammar	STATIC disabled	STATIC enabled	Performance gain
Java 1.5	754 ms	535 ms	29%
IDL	994 ms	686 ms	31%
C	152 ms	128 ms	16%

The drawback to `STATIC` is some increased complexity when using the generated parser. Your client code will need to create a parser object with the first usage and then call the `ReInit` method on subsequent usages. For example:

Example 2.55. Accommodating the STATIC option

```
if (!parserInitialized) {
  new MyParser(new FileInputStream("some_data.dat"));
  parserInitialized = true;
} else {
  MyParser.ReInit(new FileInputStream("some_data.dat"));
}
// now use the parser by calling the start symbol, e.g.:
MyParser.TranslationUnit();
```

This increased complexity may well be offset by the performance gain.

TOKEN_EXTENDS

`TOKEN_EXTENDS` is a string valued option that default to the empty string. If set, JavaCC will generate a `Token` source file in which the `Token` type is declared to extend the specified class. For example, setting `TOKEN_EXTENDS` to `SomeClass` results in a `Token` class declaration like this:

```
$ grep "public class" Token.java
public class Token extends SomeClass implements java.io.Serializable {
```

Note that JavaCC does not generate the specified class, and you may wish to supply a fully qualified type name to avoid needing to add `import` statements to the `Token` source file after it has been generated.

TOKEN_FACTORY

Sometimes you need to customize the way that JavaCC creates tokens. When this is the case, you can use `TOKEN_FACTORY` to insert a factory method into the token manager's token instantiation code. For example, suppose you want to print a message each time a token is created. You can do this by implementing a `TokenRecord` class with a `newToken` class method:

```
# examples/tokenizer/TokenRecord.java

1    public class TokenRecord {
2      public static Token newToken(int kind, String image) {
3        System.out.println("Got a new token; type is " +
LiteralsTokenManager.tokenImage[kind]);
4        return new Token(kind, image);
5      }
6    }
```

Then, when you generate the tokenizer with the `TOKEN_FACTORY` option, the token manager delegates token creation to your method which displays the message:

```
$ javacc -TOKEN_FACTORY=TokenRecord literals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file literals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LiteralsTokenManager "hello"
Got a new token; type is "hello"
Found token:hello
Got a new token; type is <EOF>
```

This also gives you a hook for token creation variations; you could, for example, subclass `Token` and create an instance of that subclass in your factory method.

TOKEN_MANAGER_USES_PARSER

`TOKEN_MANAGER_USES_PARSER` is a boolean-valued option that's `false` by default. If set to `true`, this option adds a `parser` field to the tokenizer so that the tokenizer can invoke methods on the parser.

This option is incompatible with the `STATIC` option. If `STATIC` is set to `true`, this option will have no effect. This is understandable—if the parser is composed of static methods, there's no use passing a reference to it on to the tokenizer; the tokenizer can just invoke the static methods on the parser class.

UNICODE_INPUT

`UNICODE_INPUT` is a boolean-valued option that's `false` by default. When set to `true`, this option purports to cause the tokenizer to be generated with code to handle non-ASCII values even if the lexical specification contains only ASCII values. You should avoid this option and use the techniques suggested in chapter 6, "JavaCC and Unicode" instead.

USER_CHAR_STREAM

`USER_CHAR_STREAM` is a boolean-valued option that's `false` by default. When set to `true`, it indicates to JavaCC that you have written your own code to handle reading characters from a stream and passing them to the tokenizer. Instead of generating a `SimpleCharStream` source file, JavaCC generates a `CharStream` interface that you must implement.

`USER_CHAR_STREAM` is most useful when you need complete control over the characters that are being fed to the tokenizer. For example, if your user enters characters into a graphical user interface (GUI), you could hook that GUI to a custom `CharStream` implementation and process each character as necessary. JavaCC includes just such an example program in the `examples/GUIParsing/TokenMgrVersion/` directory.

USER_TOKEN_MANAGER

`USER_TOKEN_MANAGER` is a boolean-valued option that defaults to `false`. It's much like `USER_CHAR_STREAM` except it applies to the tokenizer rather than the input stream. When this option is set to `true`, JavaCC generates a `TokenManager` interface that you'll need to implement. This interface consists of only one method:

Example 2.56. The TokenManager interface

```
$ cat TokenManager.java
/* Generated By:JavaCC: Do not edit this line. TokenManager.java Version 4.1 */
/* JavaCCOptions:SUPPORT_CLASS_VISIBILITY_PUBLIC=true */
/**
 * An implementation for this interface is generated by
 * JavaCCParser.  The user is free to use any implementation
 * of their choice.
 */

public interface TokenManager {

  /** This gets the next token from the input stream.
   *  A token of kind 0 (<EOF>) should be returned on EOF.
   */
  public Token getNextToken();

}
/* JavaCC - OriginalChecksum=cf732452a976af24d5dc6cd76954297c (do not edit this
line) */
```

JavaCC includes an example of this option in the `examples/GUIParsing/ParserVersion/` directory.

Summary

The tokenizer is a key component of most JavaCC grammars; in fact, many parsing tasks are solvable using only a tokenizer. Hopefully this chapter has given you a fuller appreciation of the capabilities of JavaCC-generated tokenizers.

Chapter 3. Parsing

"When I use a word", Humpty Dumpty said, in a scornful tone, "it means just what I choose it to mean—neither more nor less." --Lewis Carroll, "Through The Looking Glass"

Parsing Overview

Parsing Vs Tokenizing

Once the tokenizer finishes coordinating the conversion of the incoming data into a stream of tokens, the parser takes over. The parser's job is to ensure the tokens are structured according to a syntactic specification. If the input data can be parsed by a parser that's generated from the specification, the parsing stage succeeds; if not, it fails.

To clarify the role of the syntactic specification, let's look at some Java code that's arranged in an unconventional order:

```
{ void ) hello ( } public
```

Each item in this example is a valid token in the Java language: `{` is a delimiter, `void` is a type, and so forth. But they don't make sense syntactically because the Java Language Specification (JLS) dictates they be ordered like this:

```
public void hello() {}
```

Let's describe the required token order informally. First there's an access modifier, `public`. It's followed by a type, `void` and a method name, `hello`. The method has no parameters, so there's an open parenthesis followed by a close parenthesis. Since the method body is empty, the close parenthesis is followed by a set of curly braces. That's a reasonable informal description, but it leaves a lot of questions open, such as "what does the code look like if there are parameters?" and "what are the possible access modifiers for a method?" The job of the syntactic specification is to answer those questions by formalizing the description. Assuming the formalization is properly written, JavaCC reads it and generates a parser that can verify that the incoming tokens appear in a valid order.

A JavaCC syntactic specification is built from a set of *productions*, or *production rules*. A production consists of a *nonterminal* name on the left hand side and the specification for that nonterminal on the right hand side. The nonterminal specification contains references to other nonterminals as well as *terminals*, which reference token definitions. Sometimes you'll hear a production referred to as "for" a particular nonterminal, as in "this is the production for `WhileStatement`." Let's look at an example grammar that illustrates all of these terms.

Simple Parsing

Consider a U.S. phone number: `888-555-1212`. Informally, we'd describe this number format as three digits followed by a hyphen, followed by three more digits followed by a hyphen, followed by four digits. Now let's express this formally in a grammar:

Example 3.1. A phone number grammar

```
# examples/parser/phone.jj
```

```
1    PARSER_BEGIN(PhoneParser)
2    import java.io.*;
3    public class PhoneParser {
4      public static void main(String[] args) {
5        Reader sr = new StringReader(args[0]);
6        PhoneParser p = new PhoneParser(sr);
7        try {
8          p.PhoneNumber();
9        } catch (ParseException pe) {
10         pe.printStackTrace();
11       }
12     }
13   }
14   PARSER_END(PhoneParser)
15   TOKEN : {
16     <FOUR_DIGITS : (<DIGITS>){4}>
17     | <THREE_DIGITS : (<DIGITS>){3}>
18     | <#DIGITS : ["0"-"9"]>
19   }
20   void PhoneNumber() : {} {
21     <THREE_DIGITS> "-" <THREE_DIGITS> "-" <FOUR_DIGITS> <EOF>
22   }
```

There are some differences between this grammar and the examples you saw in chapter 2:

- The `options` section has disappeared. In the previous chapter we set the `BUILD_PARSER` option to `false` to prevent the parser from being created. By omitting the `options` section, we leave all the settings at their default values.
- In the previous chapter, the `PARSER_BEGIN/PARSER_END` section held only an empty class declaration. Now we're embedding a `main` method in that class declaration. We're using the same name in the `PARSER_BEGIN`, `PARSER_END`, and class declarations; if we don't JavaCC will exit with an error.
- Rather than creating a `CharStream` and passing it to the tokenizer, we create a `Reader` and pass it to the `PhoneParser` instance. `PhoneParser` will handle the details of creating the `CharStream` and the tokenizer. If you need more control, there are several other constructors available. One accepts an `InputStream`, another accepts an `InputStream` and an encoding string, and another accepts a token manager. But constructing the parser with a `Reader` is probably the most common technique.
- Instead of iterating across the tokens, we call the grammar's *start symbol*, which is the top-level production. In this case, the start symbol is the production for `PhoneNumber`. JavaCC has transformed the `PhoneNumber` production into a method; when we call this method, the parsing process proceeds from there.
- The lexical specification (that is, the contents of the `TOKEN` regular expression production) is standard stuff. There are just a few tokens defined including a private regular expression, `#DIGITS`, to reduce duplication.
- Note that we could have placed the lexical specification after the syntactic specification; JavaCC doesn't care which comes first. Since the input data is tokenized before it's handed to the parser, I prefer to put the lexical specification first. Having the grammar file in the same order feels right to me, but that's just a convention.

- The individual items in a production, such as `<THREE_DIGITS>` and `"-"` are called *expansions*, or *expansion choices*.[1]
- The last token in `PHONE_NUMBER` is an `EOF` token. As we discussed in chapter 2, this token is generated automatically by JavaCC rather than being declared in the lexical specification. It's used here in the syntactic specification to indicate that the input data should end after the phone number.

The biggest difference is near the end of the grammar where a syntactic specification now follows our lexical specification. The `PhoneNumber` nonterminal declaration is our first encounter with a JavaCC production rule. As you can see, it looks a lot like a Java method declaration. This isn't sleight-of-hand; when JavaCC generates the parser, this will be transformed into a larger (and less readable) Java method. The `PhoneNumber` name is followed by a colon and a set of open/close braces; in later examples, we'll see that we can place Java code here. The open/close braces are followed by an opening brace which begins the body of the nonterminal.

This nonterminal consists of a series of explicit token references interspersed with literal values that JavaCC will transform into token references. If you read them in order, they sound like the description of the phone number—"there are THREE_DIGITS, followed by a - character, followed by THREE_DIGITS and another -, followed by `FOUR_DIGITS` and then the end of the data." To use the terminology from our introduction, all these items—the token references and the literals—are terminals; this grammar contains only one nonterminal, `PhoneNumber`.

To generate the parser, we run JavaCC on the grammar file as we did in the tokenizer chapter. Here's a list of the files that are generated:

Example 3.2. The parser files

```
$ javacc -JDK_VERSION=1.5 phone.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ ls -l *.java
-rw-r--r--  1 tom  wheel   6147 Apr  2 00:38 ParseException.java
-rw-r--r--  1 tom  wheel   6556 Apr  2 00:38 PhoneParser.java
-rw-r--r--  1 tom  wheel    611 Apr  2 00:38 PhoneParserConstants.java
-rw-r--r--  1 tom  wheel   7485 Apr  2 00:38 PhoneParserTokenManager.java
-rw-r--r--  1 tom  wheel  12172 Apr  2 00:38 SimpleCharStream.java
-rw-r--r--  1 tom  wheel   4055 Apr  2 00:38 Token.java
-rw-r--r--  1 tom  wheel   4399 Apr  2 00:38 TokenMgrError.java
```

The only new file in this list is the parser source file, `PhoneParser.java`; all the others were present when we generated the tokenizer without the parser. Note, however, that we're passing in the `JDK_VERSION=1.5` option; this tells JavaCC that it can generate code that uses Java 1.5 language constructs are prevents us from seeing warnings when we compile the generated source code.

Now that we've covered the preliminaries, here's the phone number parser doing its job:

[1]This is the terminology that JavaCC uses internally; the JavaCC source code, for example, contains an **`Expansion`** class and a **`Choice`** subclass.

```
$ javac *.java
Note: PhoneParser.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java PhoneParser "888-555-1212"
```

Hm, that wasn't much of a demonstration, was it? JavaCC generated the parser, the parser compiled cleanly. We fed it a phone number, there were no visible results. Actually, that's a good sign. Had we entered bad data, we would have seen an error:

```
$ java PhoneParser "888-abc-1212"
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 5.
Encountered: "a" (97), after : ""
 at PhoneParserTokenManager.getNextToken(PhoneParserTokenManager.java:263)
 at PhoneParser.jj_consume_token(PhoneParser.java:132)
 at PhoneParser.PhoneNumber(PhoneParser.java:17)
 at PhoneParser.main(PhoneParser.java:8)
```

That was a lexical error, which is old hat now that you've seen lots of tokenizer examples. To get a parsing error, try entering a four digit area code (or city code, for those not familiar with the North American numbering plan):

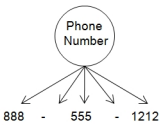
```
$ java PhoneParser "8888-555-1212"
ParseException: Encountered " <FOUR_DIGITS> "8888 "" at line 1, column 1.
Was expecting:
    <THREE_DIGITS> ...

 at PhoneParser.generateParseException(PhoneParser.java:202)
 at PhoneParser.jj_consume_token(PhoneParser.java:140)
 at PhoneParser.PhoneNumber(PhoneParser.java:15)
 at PhoneParser.main(PhoneParser.java:8)
```

There's our parser error! Four consecutive digits is a valid token, so JavaCC was able to transform the four digits from characters to a `Token` object instance. But according to the syntactic specification `FOUR_DIGITS` is not a valid token at that particular point. Thus, the parser complains that it's expecting one token type (a `THREE_DIGITS` token) but receiving another, (a `FOUR_DIGITS`). It even supplies the characters contained by the unexpected token.

If a sequence of tokens (sometimes referred to as a *sentence*) can be parsed by a particular syntactic specification, we say that the tokens can be *derived* by that specification. As you can see, JavaCC doesn't produce any visible output for successful derivations; the fact that the derivation occurred successfully serves as a sort of output. We can graphically depict the nonterminals that appear in a derivation and the tokens consumned by those nonterminals as a tree structure; this depiction is called a *parse tree*. For example, the parse tree for the sentence `888-555-1212` appears below:

Figure 3.1. Parse Tree for a Phone Number



The parse tree is purely a conceptual structure. In other words, JavaCC does not build an actual in-memory tree structure for you to manipulate with code. That's the job of the tree building utility, JJTree, to which chapter 4 is devoted.

Defining Tokens in the Syntactic Specification

In our phone number example we defined some of the tokens, like the numbers, in the lexical specification but embedded the `HYPHEN` (e.g., the literal - characters) directly into the syntactic specification. JavaCC can handle simple, literal tokens in the syntactic specification; for example, here's a parser with one production that expects an `A` followed by a `B`:

Example 3.3. Simple tokens in the syntactic specification

 # examples/parser/tokens_defined_in_grammar.jj

```
1    PARSER_BEGIN(LetterParser)
2    import java.io.*;
3    public class LetterParser {
4      public static void main(String[] args) {
5        Reader sr = new StringReader(args[0]);
6        LetterParser p = new LetterParser(sr);
7        try {
8          p.Start();
9        } catch (ParseException pe) {
10         pe.printStackTrace();
11       }
12     }
13   }
14   PARSER_END(LetterParser)
15   void Start() : {} {
16     "A" "B"
17   }
```

These token definitions end up in the `LetterParserConstants.java` file alongside any other token definitions, and they're placed in the `DEFAULT` lexical state. If that token image had already been defined in the lexical specification, JavaCC would recognize the earlier definition and refer to it rather than create a new token definition.

Embedding simple literals is a fairly common idiom. Why? Consider the difference between "`"1" "2"`" and "`<NUMBER_ONE> <NUMBER_TWO>`" in a syntactic specification. Embedding the actual character is more concise and (probably) more readable. Since JavaCC doesn't duplicate those token definitions, there's no runtime cost.

You can easily add a token that's defined by a regular expression to the lexical specification, like `["0"-"9"]` for the digits between zero and nine. If you place that same regular expression in the syntactic specification, you'll get a error message when you run it through JavaCC. Instead, you'll need to define a token within the syntactic specification by adding a full regular expression production:

Example 3.4. Complex tokens in the syntactic specification

 # examples/parser/token_regex_in_grammar.jj (lines 15 to 17)

```
15   void Start() : {} {
16     "A" "B" <DIGITS :["0"-"9"]>
17   }
```

In practice, you'll rarely see tokens defined like this. There's no need to clutter the syntactic specification with items that could just as easily be placed in the lexical specification. But the capability is there if you need it.

Syntactic Actions

Wouldn't it be nice (Beach Boys fans, take note) to see what phone number is being parsed in our example grammar? To display that information, we need to collect the token images as they're consumed. We can do this by embedding *syntactic actions* in the nonterminal. In this example, we declare a `StringBuffer` and then use it in the `PhoneNumber` production to accumulate the digits:

Example 3.5. Collecting the phone number using syntactic actions

```
# examples/parser/phone_actions.jj

1    PARSER_BEGIN(PhoneParser)
2    import java.io.*;
3    public class PhoneParser {
4      public static void main(String[] args) {
5        Reader sr = new StringReader(args[0]);
6        PhoneParser p = new PhoneParser(sr);
7        try {
8          StringBuffer n = p.PhoneNumber();
9          System.out.println("The number is " + n);
10       } catch (ParseException pe) {
11         pe.printStackTrace();
12       }
13     }
14   }
15   PARSER_END(PhoneParser)
16   TOKEN : {
17     <FOUR_DIGITS : (<DIGITS>){4}>
18     | <THREE_DIGITS : (<DIGITS>){3}>
19     | <#DIGITS : ["0"-"9"]>
20   }
21   StringBuffer PhoneNumber() : {
22     StringBuffer number = new StringBuffer();
23   }
24   {
25     <THREE_DIGITS> {number.append(token.image);}
26     "-" <THREE_DIGITS> {number.append(token.image);}
27     "-" <FOUR_DIGITS> {number.append(token.image);}
28     <EOF> {return number;}
29   }
```

Now the `PhoneNumber` nonterminal returns a `StringBuffer`, which our `main` method receives and prints to the standard output stream. Inside `PhoneNumber`, syntactic actions after each important token collect that token's image and append it to the `StringBuffer`. The last action in the production returns the `StringBuffer` that we've been building up. Note that since we wrote the `PhoneNumber` declaration so that it declares a return type other than `void`, we need to return something in the last syntactic action. If we don't do this, the parser will be generated but it won't compile.

Here's a sample run of this parser:

```
$ javacc -JDK_VERSION=1.5 phone_actions.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_actions.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "888-555-1212"
The number is 8885551212
```

When we reference the `token` identifier in the syntactic action, we're accessing a field on the parser object. This `token` field is continually updated to refer to the most recent `Token` object consumed by the parser. You can use any of the members of the `Token` class; in this case we're just using `image` which is a `String` containing the characters matched by this token's definition.

Using only one nonterminal doesn't really do justice to JavaCC's capabilities. Let's expand our example by letting `PhoneNumber` delegate to a new `AreaCode` nonterminal for parsing the area code numbers:

Example 3.6. The AreaCode Nonterminal

```
# examples/parser/phone_two_nonterminals.jj (lines 20 to 25)

20   void PhoneNumber() : {} {
21     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS> <EOF>
22   }
23   void AreaCode() : {} {
24     <THREE_DIGITS> {System.out.println("Area code = " + token.image);}
25   }
```

We can now reuse this `AreaCode` nonterminal anywhere we want to parse out an area code, and it makes the top-level `PhoneNumber` nonterminal cleaner. Now when we run it we get just the area code in the output:

```
$ javacc -JDK_VERSION=1.5 phone_two_nonterminals.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_two_nonterminals.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "888-555-1212"
Area code = 888
```

This is the general layout of a JavaCC syntactic specification: we define a start symbol which is a high level nonterminal that references lower level nonterminals. These lower level nonterminals in turn reference the tokens defined in the lexical specification. It sounds simple, and in a sense it probably is. Still, there are enough interesting details to keep us from ending this chapter just yet.

EBNF Notation

You're not limited to literals and simple token references when building a syntactic specification. You can use the *Extended Backus-Naur Form (EBNF)* notation, introduced briefly on page 2, to create more powerful expansions. EBNF notation is much like the quantifier syntax that we discussed in the tokenizer chapter (see page 29). In the context of parsing, however, it deals with nonterminals and expansions rather than the regular expressions that compose a token.

The three types of quantifiers are:

- "zero or one", or "optional", denoted by `(expansion)?`
- "one or more", denoted by `(expansion)+`
- "zero or more", denoted by `(expansion)*`

Here's an example that demonstrates each of the quantifiers:

Example 3.7. Demonstration of EBNF quantifiers

```
# examples/parser/ebnf_quantifiers.jj (lines 20 to 34)

20   void Id() : {} {
21     ZeroOrOne()  {System.out.println("ZeroOrOne\n");} |
22     OneOrMore() {System.out.println("OneOrMore\n");} |
23     ZeroOrMore() {System.out.println("ZeroOrMore\n");}
24     <EOF>
25   }
26   void ZeroOrOne() : {} {
27     "-" (<DIGIT>)?
28   }
29   void OneOrMore() : {} {
30     (<LOWERCASE_LETTER>)+
31   }
32   void ZeroOrMore() : {} {
33     (<UPPERCASE_LETTER>)*
34   }
```

In this example, we're using the EBNF notation to specify the expansion in the `ZeroOrOne` nonterminal, i.e., `(<DIGIT>)?`. There's an alternate version of the notation for this particular expansion that uses square brackets, e.g., `[<DIGIT>]`. Either works, although the square bracket usage is probably slightly more concise.

We've also got a `"-"` literal before the `DIGIT` expansion. Why? Since we're using a "zero or one" expansion, it can match zero occurrences of `DIGIT`. In other words, this expansion could match on a "sequence" that contained nothing, and if that were the only expansion in the nonterminal, that nonterminal would match everything! By adding a `"-"` to that nonterminal we constrain the range of token sequences that it can match.

```
$ java IdParser "-"
ZeroOrOne
$ java IdParser "-1"
ZeroOrOne
```

The "one or more" quantifier signals that the expansion can occur once or multiple times:

```
$ java IdParser "a"
OneOrMore
$ java IdParser "aaa"
OneOrMore
```

The "zero or more" quantifier refers to an expansion that can be skipped altogether or repeated multiple times:

```
$ java IdParser "B"
ZeroOrMore
$ java IdParser "ABCD"
ZeroOrMore
```

Each of the quantifiers can refer to an expansion containing any number of tokens or other expansions. Thus, an expansion like `(A() B() C() <EFG>)*` is perfectly valid as long as you've properly declared those other terminals and nonterminals. You'll see more complex examples of this sort as we proceed.

Communication Between Productions

Continuing our phone number example, suppose we want to collect the entire phone number while still using multiple nonterminals. We'll need to find a way to communicate between the two nonterminals. Since JavaCC maps much of the grammar contents directly into Java code, our options are similar to those available in any other type of Java code. In short, we can communicate information with a method parameter, a return value, or a field.

Communication with Parameters

First up is an example using a method parameter for communication. We'll use a `StringBuffer` as the vehicle for aggregating the phone number. We can pass the `StringBuffer` object to the `AreaCode` production. Since JavaCC turns each nonterminal into a Java method, this is as straightforward as coding up a standard Java method call:

Example 3.8. Communication with a method parameter

```
# examples/parser/phone_method_param.jj (lines 20 to 31)

20   void PhoneNumber() : {
21     StringBuffer sb = new StringBuffer();
22   }
23   {
24     AreaCode(sb) "-"
25     <THREE_DIGITS> {sb.append(token.image);} "-"
26     <FOUR_DIGITS> {sb.append(token.image);}
27     <EOF> {System.out.println("Number: " + sb.toString());}
28   }
29   void AreaCode(StringBuffer buf) : {} {
30     <THREE_DIGITS> {buf.append(token.image);}
31   }
```

Notice that the call to `AreaCode` now includes an argument, `sb`, and the `AreaCode` nonterminal declaration now includes a parameter, the `StringBuffer buf`. Here's how this looks in the actual parser code that JavaCC generates:

```
$ grep -A 3 "void AreaCode" PhoneParser.java
static final public void AreaCode(StringBuffer buf) throws ParseException {
    jj_consume_token(THREE_DIGITS);
                  buf.append(token.image);
  }
```

Using this form of communication means that any code using the `AreaCode` nonterminal will need to pass in a `StringBuffer` object. Of course, the calling code could immediately discard that `StringBuffer` (e.g., `AreaCode(new StringBuffer())`), and the Java compiler will let you know if you aren't calling the nonterminal with the proper arguments.

You'll often see grammars use method parameters to communicate. For a real world example, the Java grammar that comes with JavaCC[2] uses a parameter in a nonterminal to do a semantic check. It ensures that interfaces don't implement other interfaces:

[2]See **`javacc/examples/JavaGrammars/1.5/Java1.5.jj`** in the JavaCC distribution

```
void ImplementsList(boolean isInterface): {} {
  "implements" ClassOrInterfaceType() ( "," ClassOrInterfaceType() )*
  {
    if (isInterface) {
      throw new ParseException("An interface cannot implement other interfaces");

    }
  }
}
```

Communication with Return Values

Another option is to use a return value from the `AreaCode` production. This looks a lot like standard Java code as well:

Example 3.9. Communication with a return value

 # examples/parser/phone_return_value.jj (lines 20 to 32)

```
20   void PhoneNumber() : {
21     StringBuffer sb = new StringBuffer();
22     String areaCodeResult;
23   }
24   {
25     areaCodeResult=AreaCode() {sb.append(areaCodeResult);}  "-"
26     <THREE_DIGITS> {sb.append(token.image);} "-"
27     <FOUR_DIGITS> {sb.append(token.image);}
28     <EOF> {System.out.println("Number: " + sb.toString());}
29   }
30   String AreaCode() : {} {
31     <THREE_DIGITS> {return token.image;}
32   }
```

In this case the `PhoneNumber` nonterminal is declaring a second local variable, `areaCodeResult`, alongside our `StringBuffer`. The `areaCodeResult` is filled with the result of the `AreaCode` nonterminal, and that value is then placed in `sb`.

This technique is a bit more efficient, since now `AreaCode` returns the token image without creating any new objects. This would work if the calling production only wanted to verify the presence and format of the area code, without storing the token object.

Communicating via a return type also shows up in the Java grammar. The `Modifiers` production uses bit manipulation to collect all the modifiers for a Java declaration (e.g., `public`, `native`, and so forth) into an `int` which it then returns to the production which called it.

Communication with Fields

A third possibility is to use a field in the parser itself to collect this information. This technique couples the individual production to the parser by introducing (essentially) a global variable, so it may not be the best choice. It can be hard to track down problems with a field's value when that field can be accessed or overwritten from anywhere in the parser. But this can be a useful tool, so caveat emptor, here's how it works:

Example 3.10. Communication with a parser field

```
# examples/parser/phone_field.jj
```

```
1    PARSER_BEGIN(PhoneParser)
2    import java.io.*;
3    public class PhoneParser {
4      private static String areaCode;
5      public static void main(String[] args) {
6        Reader sr = new StringReader(args[0]);
7        PhoneParser p = new PhoneParser(sr);
8        try {
9          p.PhoneNumber();
10       } catch (ParseException pe) {
11         pe.printStackTrace();
12       }
13     }
14   }
15   PARSER_END(PhoneParser)
16   TOKEN : {
17     <FOUR_DIGITS : (<DIGITS>){4}>
18     | <THREE_DIGITS : (<DIGITS>){3}>
19     | <#DIGITS : ["0"-"9"]>
20   }
21   void PhoneNumber() : {} {
22     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS> <EOF>
23     {System.out.println("Area code: " + areaCode);}
24   }
25   void AreaCode() : {} {
26     <THREE_DIGITS> {areaCode = token.image;}
27   }
```

We declared the `areaCode` field in the `PARSER_BEGIN/PARSER_END` section. Notice that we're not overriding the `STATIC` option on this grammar, so we declare `areaCode` with a `static` modifier so the methods generated for the nonterminals can access it.

For a working example of using a field in the parser, consider the C language grammar on the JavaCC web site in the "grammars" section[3]. This grammar uses several fields to maintain state as an input file is parsed. In one case, it uses a `java.util.Set` instance to track user-defined types (e.g., `typedef long a_long_type`). As each new type definition is found, its image is placed in the `Set`. Later in the parsing process, type declarations are checked against that `Set` to see if the type has been defined.

Throwing an Exception from a Production

You can define a production that throws an exception. In the example below, we declare an inner class that's a checked exception. We also add a `throws` clause to the `AreaCode` nonterminal and place a call to that nonterminal inside a catch block at the beginning of `PhoneNumber`. To force the exception we'll include a `throw` statement in a syntactic action in `AreaCode`:

[3]https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110&expandFolder=110&folderID=0

Example 3.11. Throwing and catching an exception

```
# examples/parser/phone_exception.jj (lines 22 to 34)
```

```
22   void PhoneNumber() : {
23     try {
24       AreaCode();
25     } catch (MyException e) {
26       System.out.println("MyException while parsing AreaCode()");
27     }
28   }
29   {
30     "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
31   }
32   void AreaCode() throws MyException : {} {
33     <THREE_DIGITS> {throw new MyException();}
34   }
```

Here's the example in action:

```
$ java PhoneParser "888-555-1212"
MyException while parsing AreaCode()
```

Notice that we needed to place the call to `AreaCode` inside a syntactic action so that we could catch `MyException`. Alternatively, we could have added a `throws` clause to `PhoneNumber` as well and then caught `MyException` in our `main` method.

Declaring a production that throws an exception isn't something you see done very often. With the exception of error recovery (which you can read much more about in chapter 7), I just don't find many cases where it's useful.

Choosing an Alternate Start Symbol

So far we've been using our highest-level nonterminal, `PhoneNumber`, as the start symbol, but we could designate any nonterminal in the grammar as the start symbol. If we wanted to parse only an area code, for example, we can call that nonterminal from our `main` method:

Example 3.12. Alternate Start Symbol

```
# examples/parser/phone_other_start.jj
```

```
1     PARSER_BEGIN(PhoneParser)
2     import java.io.*;
3     public class PhoneParser {
4       public static void main(String[] args) {
5         Reader sr = new StringReader(args[0]);
6         PhoneParser p = new PhoneParser(sr);
7         try {
8           p.AreaCode();
9         } catch (ParseException pe) {
10          pe.printStackTrace();
11        }
12      }
13    }
14    PARSER_END(PhoneParser)
15    TOKEN : {
16      <FOUR_DIGITS : (<DIGITS>){4}>
17      | <THREE_DIGITS : (<DIGITS>){3}>
18      | <#DIGITS : ["0"-"9"]>
19    }
20    void PhoneNumber() : {} {
21      AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
22    }
23    void AreaCode() : {} {
24      <THREE_DIGITS>
25      {System.out.println("In AreaCode: " + token.image);}
26    }
```

When used, the parser will print out the area code:

```
$ java PhoneParser "888"
In AreaCode: 888
```

This technique can be very handy for testing a particular production without preparing a set of input data which conforms to the entire grammar. Suppose you were writing a grammar for a data set that included credit card numbers. It might be useful to test only the `CreditCardNumber` nonterminal with various numbers to ensure proper vendor extraction, required checksum calculations, and so forth. Sounds good, but there's a potential pitfall: if the lower-level nonterminal you're calling (or one of the nonterminals it calls) depends on some data that's usually initialized in a higher-level nonterminal, you'll probably see some odd errors.

Javacode Productions

So far our phone number example has used several EBNF productions to decompose the syntactic specification into manageable chunks, and in practice, most grammars can be specified using EBNF productions only. But when a particular grammar construct is difficult to write using EBNF, there's another type of production available—the *Javacode production*.

A Javacode production is just as it sounds—a production rule that consists solely of Java code. Here's the phone number parser `AreaCode` rewritten using a Javacode production:

Example 3.13. Using a Javacode production

 # examples/parser/phone_javacode.jj (lines 20 to 30)

```
20   void PhoneNumber() : {} {
21     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS> <EOF>
22   }
23   JAVACODE
24   void AreaCode() {
25     Token tok = getNextToken();
26     if (tok.kind != THREE_DIGITS) {
27       throw new ParseException("Expected an area code!");
28     }
29     System.out.println("Area code = " + tok.image);
30   }
```

There are a couple of differences between this Javacode production and the usual EBNF productions:

- The production definition is preceded by the JavaCC keyword `JAVACODE`.
- The `AreaCode` production is plain Java code; you can't use bare token references like `<THREE_DIGITS>`, expansions, or any of the other usual JavaCC constructs. However, you can access most of the JavaCC functionality, albeit in a clunky way, by invoking methods and referencing fields as you'd normally do when writing a standard Java program.
- The Javacode production in this example calls the `getNextToken` method on the `PhoneParser` object. This method returns the next token in the token stream.

A real-world use of Javacode productions occurs in the JavaCC grammar—that is, the grammar that defines the format of JavaCC grammars. A JJTree node descriptor expression (look ahead to chapter 4 for more on JJTree) can contain nested expressions wrapped by parentheses, and this nesting can be arbitrarily deep. Parsing nested parentheses with regular expressions is impossible since there's no telling how many parentheses will appear, so the grammar uses a Javacode production to solve the problem. Each time it encounters a left parenthesis it increments a local variable, with each right parenthesis it decrements the same variable. When it reaches a final right parenthesis that closes the expression, it returns from the production. Here's the code:

Example 3.14. A real-world Javacode production

 # examples/parser/real_world_javacode.jj

```
1    JAVACODE
2    void node_descriptor_expression() {
3      Token tok;
4      int nesting = 1;
5      while (true) {
6        tok = getToken(1);
7        if (tok.kind == 0) {
8          throw new ParseException();
9        }
10       if (tok.kind == LPAREN) nesting++;
11       if (tok.kind == RPAREN) {
12         nesting--;
13         if (nesting == 0) break;
14       }
15       tok = getNextToken();
16     }
17   }
```

The code above calls `getToken(1)` to get the first token after the current one and then calls `getNextToken` as the last statement in the `while` loop. It does this because the `getToken` method just returns a token without consumning it. That way, the node descriptor expressions can be specified in the grammar file as, more or less, `"(" node_descriptor_expression() ")"`. Only the actual content of the expression is consumed by the Javacode production; the opening and closing parentheses are dealt with in more conventional JavaCC code.

As stated earlier, avoid Javacode productions if possible. Using them means writing lots of clunky code instead of a few concise JavaCC keywords and constructs. Clunky code notwithstanding, it's a good tool to have in your bag, just in case.

Grammar Construction

Thus far we've reviewed a number of items you can use when building a grammar—syntactic actions, various ways of defining nonterminals and expansions, and so forth. Sometimes, however, problems arise when constructing a grammar that can't be solved with the JavaCC tools you have available—you may need to re-design your grammar.

Choice Conflicts

To illustrate one possible problem, let's extend the phone number example a bit. Suppose we need to parse either local phone numbers, which consist of seven digits (e,g., `555-1212`) or U.S. nation-wide numbers, which contain ten digits (e.g., `888-555-1212`). Our first attempt might be to extend the grammar, adding new nonterminals for local and national phone numbers and having our top-level `PhoneNumber` nonterminal allow for either possibility:

Example 3.15. A choice conflict

```
# examples/parser/phone_choice.jj (lines 20 to 31)

20   void PhoneNumber() : {} {
21     (LocalNumber() | CountryNumber()) <EOF>
22   }
23   void LocalNumber() : {} {
24     AreaCode() "-" <FOUR_DIGITS>
25   }
26   void CountryNumber() : {} {
27     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
28   }
29   void AreaCode() : {} {
30     <THREE_DIGITS>
31   }
```

This looks reasonable on the surface—but when we run it through JavaCC we get a warning message about a *choice conflict*:

```
$ javacc phone_choice.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_choice.jj . . .
Warning: Choice conflict involving two expansions at
         line 21, column 4 and line 21, column 20 respectively.
         A common prefix is: <THREE_DIGITS> "-"
         Consider using a lookahead of 3 or more for earlier expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

In addition, even though JavaCC generated a parser source file, it's not valid Java code so it won't compile.

What happened? With many JavaCC problems, reading the warning message closely tells the story. Since both the `LocalNumber` and the `CountryNumber` nonterminals begin with three digits followed by a hyphen, JavaCC can't tell which nonterminal to call when it encounters the `<THREE_DIGITS>` token. This is a *choice point*; that is, a point at which the next token could be the start of more than one nonterminal. JavaCC needs more information so that it can tell which nonterminal to pick; giving it this additional information *resolves* the choice conflict.

Resolving Choice Conflicts with Left Factoring

How can a choice conflict be resolved? The main technique is *left factoring*. Left factoring is a grammar transformation that involves extracting expansions that are common to several nonterminals and moving them to a higher-level nonterminal. For example, we might resolve the phone number choice conflict by left factoring the `AreaCode` and the `"-"` into `PhoneNumber`:

Example 3.16. The grammar after left factoring

```
# examples/parser/phone_left_factored.jj (lines 20 to 31)

20   void PhoneNumber() : {} {
21     AreaCode() "-" (LocalNumber() | CountryNumber()) <EOF>
22   }
23   void LocalNumber() : {} {
24     <FOUR_DIGITS>
25   }
26   void CountryNumber() : {} {
27     <THREE_DIGITS> "-" <FOUR_DIGITS>
28   }
29   void AreaCode() : {} {
30     <THREE_DIGITS>
31   }
```

When we run JavaCC on this revised grammar it's processed with no warnings. More importantly, it produces a parser that can handle both types of phone numbers:

```
$ javacc -JDK_VERSION=1.5 phone_left_factored.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_left_factored.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "555-1212"
$ java PhoneParser "888-555-1212"
```

Left factoring has some consequences. The `PhoneNumber` nonterminal now carries the weight of the parsing load, the other nonterminals serve only minor roles. If several different productions depended on either `LocalNumber` or `CountryNumber`, we'd

need to left factor the common expansions up into all those nonterminals. This could litter the grammar with duplicated expansions.

One technique for reducing that duplication is to move the left factored expansions into a new nonterminal. For example, we can move the `AreaCode() "-"` expansion into a new `Preamble` nonterminal:

Example 3.17. Removing duplication with a new nonterminal

```
# examples/parser/phone_left_factored_new_nonterminal.jj (lines 20 to 34)

20   void PhoneNumber() : {} {
21     Preamble() (LocalNumber() | CountryNumber()) <EOF>
22   }
23   void Preamble() : {} {
24     AreaCode() "-"
25   }
26   void LocalNumber() : {} {
27     <FOUR_DIGITS>
28   }
29   void CountryNumber() : {} {
30     <THREE_DIGITS> "-" <FOUR_DIGITS>
31   }
32   void AreaCode() : {} {
33     <THREE_DIGITS>
34   }
```

Now any other uses of `AreaCode() "-"` can be replaced with the `Preamble` nonterminal. Generally, after creating a new nonterminal, it's a good idea to do a quick scan of the grammar to see if any new refactoring opportunities have presented themselves.

Left and Right Recursion

Another potential grammar problem is *left recursion*, (sometimes called *head recursion*). Left recursion occurs when a nonterminal calls itself recursively in a way that guarantees that the recursion will never end. For example, let's modify the phone number example so that a `CountryNumber` can have several `AreaCode`s. Our first attempt might be to redefine `AreaCode` to indicate nested area codes like this:

Example 3.18. Left recursion

```
# examples/parser/phone_left_recursion.jj (lines 20 to 25)

20   void CountryNumber() : {} {
21     AreaCode() "-" <FOUR_DIGITS>
22   }
23   void AreaCode() : {} {
24     AreaCode() "-" <THREE_DIGITS>
25   }
```

There's a problem in line 23. The first expansion in the `AreaCode` production is the `AreaCode` nonterminal. This means that the first thing JavaCC would do when parsing a series of tokens with this production is to have the production call itself. That call results in another call to the `AreaCode` production, which results in yet another call, and so forth until the JVM raises a `StackOverflowError`.

Fortunately, JavaCC is smart enough to catch this error. When we attempt to process this grammar we get the error message you see below:

```
$ javacc phone_left_recursion.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
```

```
Reading from file phone_left_recursion.jj . . .
Error: Line 23, Column 1: Left recursion detected: "AreaCode... --> AreaCode..."
Detected 1 errors and 0 warnings.
```

One possible fix for this is similar to the left factoring fix: introduce a new nonterminal. In this case, one that encapsulates the repeated `AreaCode` nonterminals:

Example 3.19. Left recursion fixed with a new nonterminal

 # examples/parser/phone_left_recursion_new_nonterminal.jj (lines 20 to 28)

```
20   void CountryNumber() : {} {
21     AreaCodeList() <FOUR_DIGITS>
22   }
23   void AreaCodeList() : {} {
24     (AreaCode() "-")+
25   }
26   void AreaCode() : {} {
27     <THREE_DIGITS>
28   }
```

A common case of left recursion occurs when you're attempting to parse arithmetic expressions. A naive first attempt at a grammar for adding a series of numbers might be to say that an `Expression` consists of adding two things, where each of those things is either an `Expression` or a `Term`:

Example 3.20. A naive arithmetic grammar

 # examples/parser/naive_expression.jj (lines 19 to 24)

```
19   void Expression() : {} {
20     (Expression() | Term()) "+" (Expression() | Term())
21   }
22   void Term() : {} {
23     <DIGIT>
24   }
```

But now `Expression` contains an `Expression` as its leftmost nonterminal, so a left recursion error results. Let's replace the left recursion with iteration:

Example 3.21. A better arithmetic grammar

 # examples/parser/better_expression.jj (lines 19 to 24)

```
19   void Expression() : {} {
20     Term() ("+" Term())+
21   }
22   void Term() : {} {
23     <DIGIT>
24   }
```

Now the `Expression` nonterminal specifies that it has to start with a `Term`, which is reasonable because for the purposes of this example we don't recognize unary expressions such as (`+2+2`). That `Term` is then followed by at least one expansion of `+ Term ()`, which means that we can have as many additions in the series as we need. You'll see a real world example of parsing expressions in chapter 8 when we review the JavaCC grammar.

A less troublesome type of recursion, *right recursion*, occurs when the rightmost nonterminal refers to an enclosing nonterminal. Right recursion doesn't cause problems for JavaCC; consider the following production which parses a series of `AreaCode` nonterminals:

Example 3.22. Right recursion

```
# examples/parser/phone_right_recursion.jj (lines 18 to 22)

18   void AreaCode() : { Token t; } {
19     t=<THREE_DIGITS>
20     {System.out.println("Parsed " + t.image); }
21     "-" [AreaCode()]
22   }
```

JavaCC can generate a parser for this grammar despite the fact that it uses a recursive call. Why? Consider how the calls unfold. When the parser encounters the `AreaCode` nonterminal, it consumes the `THREE_DIGITS` token and the `-` token. Next it finds the optional `AreaCode` expansion, which triggers a recursive call to `AreaCode`. But that call results in another two tokens (`THREE_DIGITS` and `-`) being consumed. The recursive calls continue, and for each call, several more tokens are consumed. We're only limited by the amount of recursion (i.e., the stack size) that the JVM can handle.

One difficulty in learning how to construct a grammar is that most grammars you encounter will be finished—you won't have access to the many difficult and interesting design decisions the author made before publishing the grammar. The best way to learn is to practice by attempting to write grammars for various data formats. When you run into choice conflicts, try various ways of resolving them. Even if you're not entirely successful, at least you'll have an understanding of the difficulties involved.

Lookahead

Choice points, Backtracking, and Lookahead

In the previous section we resolved a choice conflict by left factoring. In this section we'll use another mechanism that JavaCC provides for this purpose: *lookahead*. Lookahead does what it says: it causes the parser to "look ahead" a few tokens to see what's on the horizon. The parser can then use the results of the lookahead to choose which nonterminal to use to handle the incoming tokens.

Some form of lookahead always occurs whether we specify it or not. As JavaCC parses the input data, it's continually looking one token ahead and making the decision on which nonterminal to derive based on that token. We only need to insert an explicit lookahead directive when looking ahead one token will not provide enough information to choose the next nonterminal.

JavaCC needs lookahead to resolve choice conflicts because it does not backtrack. In other words, JavaCC will not choose to move down a particular path of nonterminals, fail, and then back out to the place where it made the choice. Instead, it'll just fail if its chosen path is not successful. Lookahead gives JavaCC the information it needs to choose the right path.

In the phone number example, we could resolve the choice conflict by adding a lookahead directive, instructing JavaCC to generate a parser that looks at the upcoming tokens to determine which phone number style is approaching. Since the first token is `<THREE_DIGITS>` in both cases, looking ahead one token won't be enough.

Since the second token is the hyphen character in both cases, looking ahead two tokens won't suffice either. We'll have to use a *multiple token lookahead* directive of three tokens to find the place where the two nonterminals differ. You can see this in the grammar below where `LOOKAHEAD(3)` appears in the `PhoneNumber` nonterminal:

Example 3.23. Resolving a choice conflict with multiple token lookahead

```
# examples/parser/phone_lookahead.jj (lines 20 to 34)

20   void PhoneNumber() : {} {
21     (
22       LOOKAHEAD(3) LocalNumber() {System.out.println("Found a local number");}
23       | CountryNumber() {System.out.println("Found a country number");}
24     ) <EOF>
25   }
26   void LocalNumber() : {} {
27     AreaCode() "-" <FOUR_DIGITS>
28   }
29   void CountryNumber() : {} {
30     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
31   }
32   void AreaCode() : {} {
33     <THREE_DIGITS>
34   }
```

Let's generate the parser and run it with various numbers to see if it's recognizing them properly:

```
$ javacc -JDK_VERSION=1.5 phone_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_lookahead.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "888-555-1212"
Found a country number
$ java PhoneParser "555-1212"
Found a local number
```

Sweet smell of success! This parsing stuff is a layup.

Notice that we placed the lookahead directive right before the choice conflict; this was not accidental. Had we placed the directive a bit earlier in the nonterminal—e.g., before that opening parenthesis—we would have received a warning from JavaCC:

```
$ javacc phone_lookahead_nonchoice.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_lookahead_nonchoice.jj . . .
Warning: Line 23, Column 14: Encountered LOOKAHEAD(...) at a non-choice location.
  This will be ignored.
Warning: Choice conflict involving two expansions at
         line 24, column 5 and line 25, column 7 respectively.
         A common prefix is: <THREE_DIGITS> "-"
         Consider using a lookahead of 3 or more for earlier expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 2 warnings.
```

Again, the warning messages supplied by JavaCC can be quite helpful. This one tells us the lookahead directive is incorrectly placed and will be ignored. Since the lookahead directive is ignored, we'll receive the same choice conflict warning we saw on page 81.

Insufficient Lookahead

JavaCC assumes you know what you're doing when you add a lookahead specification. In other words, even if the lookahead specification you've added doesn't resolve the choice conflict, JavaCC won't issue a choice conflict warning. Suppose in our local/national phone number example we used a lookahead specification with one token rather than three. That's insufficient since the first token for either a local number or a national number is the same. Here's the grammar:

Example 3.24. Insufficient lookahead, but no warning

```
# examples/parser/phone_insufficient_lookahead.jj (lines 20 to 34)

20   void PhoneNumber() : {} {
21     (
22       LOOKAHEAD(1) LocalNumber() {System.out.println("Found a local number");}
23       | CountryNumber() {System.out.println("Found a country number");}
24     ) <EOF>
25   }
26   void LocalNumber() : {} {
27     AreaCode() "-" <FOUR_DIGITS>
28   }
29   void CountryNumber() : {} {
30     AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
31   }
32   void AreaCode() : {} {
33     <THREE_DIGITS>
34   }
```

When we run JavaCC on this grammar it generates a parser without complaint:

```
$ javacc phone_insufficient_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_insufficient_lookahead.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

But compiling it is a different kettle of fish:

```
$  javac *.java
PhoneParser.java:20: unreachable statement
      CountryNumber();
      ^
Note: PhoneParser.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

Since we used a lookahead specification, JavaCC didn't warn us even though that specification was insufficient.

You can instruct JavaCC to always flag choice conflicts, even if a lookahead specification is present (see the `FORCE_LA_CHECK` option on page 104 for more information), but this will result in warnings for choice conflicts that are resolved by a proper lookahead specification. Your best bet: first, keep JavaCC's behavior in mind when

inserting a lookahead specification; second, consider occasionally running JavaCC with `FORCE_LA_CHECK` enabled to ensure no surprises have crept into the grammar.

Global and Local Lookahead

The example in the prior section showed how to use *local lookahead*. Local lookahead tells JavaCC that it needs to apply a specific multiple token lookahead value to resolve a particular choice, whereas *global lookahead* applies to the entire grammar. Global lookahead defaults to one token, but we can set the global lookahead default to a different value by adding an `options` section with a `LOOKAHEAD=3;` option, or by passing the value on the command line:

Example 3.25. Setting global lookahead

```
$ javacc -LOOKAHEAD:3 phone_global_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_global_lookahead.jj . . .
Warning: Lookahead adequacy checking not being performed since option LOOKAHEAD
is more than 1.  Set option FORCE_LA_CHECK to true to force checking.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

Running JavaCC on the grammar above, we get a warning message about the use of global lookahead:

```
Warning: Lookahead adequacy checking not being performed since option LOOKAHEAD
is more than 1.  Set option FORCE_LA_CHECK to true to force checking.
```

This is (in this case) only a warning, since the generated parser compiles and runs fine. We could turn on the `FORCE_LA_CHECK` option, but we'd only ignore the resulting choice conflict warning because we're setting a large global lookahead value.

Here's one disadvantage of employing the global lookahead: JavaCC will do what we say, not what'd we'd like. Not content with calculating lookahead for a particular choice conflict, it will calculate the lookahead tables for the entire grammar. You can find more detailed performance notes on this option in Table 3.3, “Global Lookahead Benchmark Results” [105]. In general, global lookahead is a bad idea... but it's there if you really need it.

Syntactic Lookahead

There are situations where counting ahead a predetermined number of tokens isn't the best way to structure a choice. For example, suppose we modify our phone number grammar so that both local and national numbers can be preceded by a series of `A` or `B` characters. In this scenario, `ABA555-1212` would be a valid local number and `AAAAAAAAB777-555-1212` would be a valid national number. Why a carrier might use such a twisted numbering scheme is puzzling, but stay with me.

A multiple token lookahead specification won't do the trick here, because there can be a unlimited number of `A` or `B` characters. If we merely added a lookahead specification to `LocalNumber` using, say, `LOOKAHEAD(4)`, we'd be in trouble as soon as someone submitted a national number like `ABABA777-555-1212`. The parser would look ahead for the first four tokens, find `ABAB`, and presume it was dealing with a

`LocalNumber`. That determination made, the parser would then consume the `THREE_DIGITS` token as a local area code. The next expected token would be a `FOUR_DIGITS`, so we'd get a parser exception when it encountered the `THREE_DIGITS` token (`555`) instead.

We could use *infinite lookahead*, or approximate infinity by using the maximum value of a Java integer as a lookahead specification: `LOOKAHEAD(2147483647)`. But that's rolling out a howitzer to target a mouse. Instead, we can use a *syntactic lookahead specification*. A syntactic lookahead specification uses a syntactic construct as a choice resolver. In this case, we can instruct the parser to look for the contents of the `LocalNumber` expansion and, if it finds that, parse the tokens as a local phone number:

Example 3.26. Using syntactic lookahead

```
# examples/parser/phone_syntactic_lookahead.jj (lines 21 to 35)

21   void PhoneNumber() : {} {
22     (
23       LOOKAHEAD(("A"|"B")+ AreaCode() "-" <FOUR_DIGITS>) LocalNumber()
{System.out.println("Found a local number");}
24       | CountryNumber() {System.out.println("Found a country number");}
25     ) <EOF>
26   }
27   void LocalNumber() : {} {
28     ("A"|"B")+ AreaCode() "-" <FOUR_DIGITS>
29   }
30   void CountryNumber() : {} {
31     ("A"|"B")+ AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
32   }
33   void AreaCode() : {} {
34     <THREE_DIGITS>
35   }
```

With that syntactic lookahead in place, let's generate the parser and feed it a few numbers to ensure all's well:

```
$ javacc -JDK_VERSION=1.5 phone_syntactic_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_syntactic_lookahead.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "ABABABA888-555-1212"
Found a country number
$ java PhoneParser "AAAAAA555-1212"
Found a local number
```

You can find an example of a syntactic lookahead in the Java language grammar that ships with JavaCC; see page 181 for more details on that occurrence.

Combining Multiple Token and Syntactic Lookahead

Another option in the toolbox is to combine multiple token and syntactic lookahead. Returning to our previous example, if we knew we had to check only a couple of `A` or `B` characters before choosing a local number, we could look ahead a limited number

of tokens. As long as the syntactic lookahead is satisfied within that number of tokens, the specified choice will be made. Here's how that looks:

Example 3.27. Combining multiple token and syntactic lookahead

```
# examples/parser/phone_combined_mult_syn.jj (lines 20 to 34)
```

```
20   void PhoneNumber() : {} {
21     (
22       LOOKAHEAD(10, ("A"|"B")+ AreaCode() "-" <FOUR_DIGITS>) LocalNumber()
{System.out.println("Found a local number");}
23       | CountryNumber() {System.out.println("Found a country number");}
24     ) <EOF>
25   }
26   void LocalNumber() : {} {
27     ("A"|"B")+ AreaCode() "-" <FOUR_DIGITS>
28   }
29   void CountryNumber() : {} {
30     ("A"|"B")+ AreaCode() "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
31   }
32   void AreaCode() : {} {
33     <THREE_DIGITS>
34   }
```

When run, this handles a few leading As and Bs with aplomb:

```
$ javacc phone_combined_mult_syn.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_combined_mult_syn.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
Note: PhoneParser.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java PhoneParser "AB888-555-1212"
Found a country number
$ java PhoneParser "A555-1212"
Found a local number
```

But, as expected, it collapses when presented with more leading letters than the lookahead specification expects. In this case, we should get a `CountryNumber`, but the parser chooses the `LocalNumber` production and runs into trouble when it sees a `THREE_DIGITS` instead ot the expected `FOUR_DIGITS`:

```
$ java PhoneParser "AABABABABB888-555-1212"
ParseException: Encountered " <THREE_DIGITS> "555 "" at line 1, column 15.
Was expecting:
    <FOUR_DIGITS> ...

 at PhoneParser.generateParseException(PhoneParser.java:383)
 at PhoneParser.jj_consume_token(PhoneParser.java:268)
 at PhoneParser.LocalNumber(PhoneParser.java:61)
 at PhoneParser.PhoneNumber(PhoneParser.java:16)
 at PhoneParser.main(PhoneParser.java:8)
```

There's a plot twist here. When you specify syntactic lookahead without multiple token lookahead (e.g., `LOOKAHEAD(LocalNumber())`), what you're really specifying is syntactic lookahead with infinite multiple token lookahead. In other words, `LOOKAHEAD(LocalNumber())` results in exactly the same code as `LOOKAHEAD (2147483647, LocalNumber())`. Leaving this default value in the grammar is wasteful; in many situations you can estimate a reasonable upper bound on the number of tokens to expect.

Semantic Lookahead

At times neither a multiple token nor a syntactic lookahead specification will provide enough information to resolve a choice conflict. This is uncommon, but it does happen. When it occurs, there's *semantic lookahead*.

Imagine we modify our phone number problem again—now we've parsing local numbers for two towns: Keysville and Farmville. Keysville numbers start with `123` and all other numbers are Farmville numbers. There's some reason why we can't specify a `123` token (perhaps we've got a lexical action associated with the `THREE_DIGITS` token), so instead we have to make do with the current set of token definitions. Placing the nonterminals in the grammar and hoping for the best won't work:

Example 3.28. No magic in parsing

```
# examples/parser/phone_need_semantic.jj (lines 20 to 34)

20   void PhoneNumber() : {} {
21     (
22         KeysvilleNumber() {System.out.println("Found a Keysville number");}
23       | FarmvilleNumber() {System.out.println("Found a Farmville number");}
24     ) <EOF>
25   }
26   void KeysvilleNumber() : {} {
27     AreaCode() "-" <FOUR_DIGITS>
28   }
29   void FarmvilleNumber() : {} {
30     AreaCode() "-" <FOUR_DIGITS>
31   }
32   void AreaCode() : {} {
33     <THREE_DIGITS>
34   }
```

There's a glaring choice conflict between `KeysvilleNumber` and `FarmvilleNumber` since they contain exactly the same expansions. Running JavaCC on this grammar will result in a choice conflict warning, and the resulting parser won't compile. As you no doubt suspected, we can solve this with semantic lookahead.

Semantic lookahead involves embedding Java code in the grammar at the choice point. If the Java code returns `true`, the current expansion is chosen; if not, control passes to the next choice point. The Java code can be a single expression that returns a boolean value, or, if the check is more elaborate, a call to a method (usually defined in the `PARSER_BEGIN/PARSER_END` section) that returns a boolean value.

Here's how we embed a semantic lookahead directive for an area code of `123`:

Example 3.29. Checking the token value with semantic lookahead

```
# examples/parser/phone_semantic_lookahead.jj (lines 20 to 35)

20   void PhoneNumber() : {} {
21     (
22       LOOKAHEAD({getToken(1).image.equals("123")})
23         KeysvilleNumber() {System.out.println("Found a Keysville number");}
24       | FarmvilleNumber() {System.out.println("Found a Farmville number");}
25     ) <EOF>
26   }
27   void KeysvilleNumber() : {} {
28     AreaCode() "-" <FOUR_DIGITS>
29   }
30   void FarmvilleNumber() : {} {
31     AreaCode() "-" <FOUR_DIGITS>
32   }
33   void AreaCode() : {} {
34     <THREE_DIGITS>
35   }
```

Notice that we placed the Java code inside curly braces to differentiate it from other lookahead notations. Now when we generate the parser and pass in a phone number with a `123` area code we get the Keysville number:

Example 3.30. Semantic lookahead in action

```
$ javacc -JDK_VERSION=1.5 phone_semantic_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_semantic_lookahead.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "123-1212"
Found a Keysville number
$ java PhoneParser "321-1212"
Found a Farmville number
```

For a real world example of semantic lookahead, here's the `TypeSpecifier` production in C language grammar (the one mentioned on page 77):

Example 3.31. A real world example of semantic lookahead

```
void TypeSpecifier() : {} {
(
  <VOID>       | <CHAR>     | <SHORT>   |
  <INT>        | <LONG>     | <FLOAT>    |
  <DOUBLE>     | <SIGNED>   |
  <UNSIGNED>   |
  StructOrUnionSpecifier() | EnumSpecifier() |
  LOOKAHEAD({isType(getToken(1).image)}) TypedefName()
)
}
```

First the parser checks against a series of known types such as `int`, `float`, and so forth. If none of those match, it checks a few other expansions (structs, unions, and enumerations). If those don't match, it uses a semantic lookahead specification to call the `isType` method to determine if the token's image is found in a `Set` of user-defined types. If that's the case, the parser enters the `TypedefName` production; if not, a parsing error results. There's no way to discover those user-defined types before actually parsing the file so this sort of check can't be done with multiple token or syntactic lookahead.

Nested Lookahead

One of the trickier corners of JavaCC is its behavior when a grammar contains *nested lookahead*. Nested lookahead occurs when one lookahead directive overlaps another. Consider the following grammar, which parses out a name in a rather roundabout fashion:

Example 3.32. Nested syntactic lookahead

```
# examples/parser/nested_syntactic_la.jj (lines 15 to 33)

15   SKIP : {
16     " "
17   }
18   void Start() : {} {
19     LOOKAHEAD(Fullname()) Fullname() | Douglas()
20     <EOF>
21   }
22   void Fullname() : {} {
23     (
24       LOOKAHEAD(Douglas() Munro())
25       Douglas()
26       |
27       Douglas() Albert()
28     )
29     Munro()
30   }
31   void Douglas() : {} { "Douglas" }
32   void Albert() : {} { "Albert" }
33   void Munro() : {} { "Munro" }
```

Suppose we hand this grammar the name `Douglas Albert Munro`. Clearly the `Start` production needs to determine whether it's dealing with a full name or just a first name. To do this, it uses syntactic lookahead to detect the presence of a `Fullname` production. All well and good, but when that lookahead moves into the `Fullname` production, it encounters another lookahead directive. But there's a catch: in JavaCC, nested syntactic lookahead is ignored. This means that the lookahead specification in `Fullname` is not used, so the parser must resolve the choice between `Douglas()` and `Douglas() Albert()` without the ability to look ahead.

With that limitation in mind, you can see that things will rapidly go downhill. The first token is `Douglas`, since it satisfies the `Douglas()` expansion, the lookahead chugs on to ensure that the next expansion, `Munro`, is a match. But we submitted `Douglas Albert Munro`, so there's no match, and the entire lookahead fails. Having failed, the parser chooses the remaining alternative in the `Fullname` production-that is, the `Douglas()` production. When the next token turns out to be `Albert` rather than the expected `EOF`, the only option that remains is to error out.

You can see the play by play if you generate the parser with the `DEBUG_LOOKAHEAD` option enabled (see page 102 for a complete description of this option). Notice how the lookahead expects to see a `Munro` token, but sees `Albert` instead, then fails:

```
$ javacc -DEBUG_LOOKAHEAD -JDK_VERSION=1.5 nested_syntactic_la.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file nested_syntactic_la.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

```
$ javac *.java
$ java NestedParser "Douglas Albert Munro"
Call:   Start
  Call:   Fullname(LOOKING AHEAD...)
    Visited token: <"Douglas" at line 1 column 1>; Expected token: <"Douglas">
    Visited token: <"Albert" at line 1 column 9>; Expected token: <"Munro">
  Return: Fullname(LOOKAHEAD FAILED)
  Call:   Douglas
    Consumed token: <"Douglas" at line 1 column 1>
  Return: Douglas
Return: Start
ParseException: Encountered " "Albert" "Albert "" at line 1, column 9.
Was expecting:
    <EOF>

 at NestedParser.generateParseException(NestedParser.java:386)
 at NestedParser.jj_consume_token(NestedParser.java:268)
 at NestedParser.Start(NestedParser.java:23)
 at NestedParser.main(NestedParser.java:8)
```

In my experience, the fact that nested lookahead is ignored is more likely to result in an unexpected choice conflict than erroneous results. This can be frustrating if you don't keep this quirk in mind; I've looked at apparently straightforward lookahead paths and wondered why it's not working and why `DEBUG_LOOKAHEAD` doesn't show the lookahead in action. The lesson here may be "trust the debug output"; it shows what's really happening, not what you wish was happening!

To further complicate things, although nested syntactic lookahead is ignored, nested semantic lookahead is not. Suppose, rather than using syntactic lookahead, the `Fullname` nonterminal checks the contents of the next two tokens via a couple of calls to `getToken`:

Example 3.33. Nested semantic lookahead

 # examples/parser/nested_semantic_la.jj (lines 18 to 33)

```
18   void Start() : {} {
19     LOOKAHEAD(Fullname()) Fullname() | Douglas()
20     <EOF>
21   }
22   void Fullname() : {} {
23     (
24       LOOKAHEAD({getToken(1).image == "Douglas" && getToken(2).image ==
"Munro"})
25       Douglas()
26       |
27       Douglas() Albert()
28     )
29     Munro()
30   }
31   void Douglas() : {} { "Douglas" }
32   void Albert() : {} { "Albert" }
33   void Munro() : {} { "Munro" }
```

The nested semantic lookahead handles the problematic `Douglas Albert Munro` input with ease:

Example 3.34. Nested semantic lookahead in action

```
$ javacc -JDK_VERSION=1.5 -DEBUG_LOOKAHEAD nested_semantic_la.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file nested_semantic_la.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java NestedParser "Douglas Albert Munro"
Call:   Start
  Call:   Fullname(LOOKING AHEAD...)
    Visited token: <"Douglas" at line 1 column 1>; Expected token: <"Douglas">
    Visited token: <"Albert" at line 1 column 9>; Expected token: <"Albert">
    Visited token: <"Munro" at line 1 column 16>; Expected token: <"Munro">
  Return: Fullname(LOOKAHEAD SUCCEEDED)
  Call:   Fullname
    Call:   Douglas
      Consumed token: <"Douglas" at line 1 column 1>
    Return: Douglas
    Call:   Albert
      Consumed token: <"Albert" at line 1 column 9>
    Return: Albert
    Call:   Munro
      Consumed token: <"Munro" at line 1 column 16>
    Return: Munro
  Return: Fullname
Return: Start
```

In summary, nested syntactic lookahead is ignored, while nested semantic lookahead is not. This can lead to odd results if you don't expect this behavior and, even if you do, it can be tricky to trace through a complicated series of lookahead directives and the nonterminals they employ. The `DEBUG_LOOKAHEAD` option is your friend; use it and read the output carefully to help clarify puzzling lookahead results.

Using a Lookahead-Only Production

Lookahead is handy for resolving choice conflicts, but it has side benefits as well. Suppose you have a C program that contains a function like this:

```
void a_very_long_function() {
 // hundreds of line of code
}
```

At some point in the grammar, you probably have a syntactic lookahead directive that checks for a function declaration before making a choice, e.g. `LOOKAHEAD(FunctionDeclaration()) FunctionDeclaration()`. While this may work, it has the potential of being slower than necessary when `FunctionDeclaration` includes an expansion of the statements within that function, like this:

```
void FunctionDeclaration() : {} {
  Type() Name() Parameters() "{" Statements() "}"
}
```

In other words, the lookahead will need to scan forward until it hits the closing brace of the function, so it will scan all of the intervening statements and their tokens. In the case of `a_very_long_function`, there may be thousands of tokens to skip over, just to verify the presence of a `FunctionDeclaration` nonterminal.

A cleaner technique is to use a *lookahead-only production*. With a lookahead-only production, you use a new production called, say `FunctionDeclarationLookahead`, that verifies only the information necessary to ensure that what's approaching is indeed a function. For example, you might write this new production so that it only checks for the expansions and tokens up to and including the function's initial opening brace:

```
void FunctionDeclarationLookahead() : {} {
  Type() Name() Parameters() "{"
}
```

Now you can use the lookahead-only production to choose function declarations, e.g, `LOOKAHEAD(FunctionDeclarationLookahead()) FunctionDeclaration()`. You'll avoid unnecessarily scanning those thousands of tokens and your parser will be faster.

Of course, using a lookahead-only production in this way only ensures that what's coming up can syntactically only be a `FunctionDeclaration`. The parser may encounter a syntax error while deriving one of the `Statement` nonterminals in that function; in other words, the function maybe well be invalid and result in a parsing exception. This could affect your error handling strategy; see chapter 7 for more on error handling in general.

A real-world example of lookahead-only productions is found in the Java 1.1 grammar that comes with JavaCC [4]. The `MethodDeclarationLookahead` production in that grammar follows more or less the same pattern as the example above, although the lookahead-only prediction is structured slightly differently to accommodate Java's access modifiers. Here are the benchmark results for each technique when parsing a fairly sizeable data file:

Table 3.1. Lookahead-only Production Benchmark Results

Technique	Completion time
Using lookahead-only productions	870 ms
Not using lookahead-only productions	1128 ms

To quote Unreal Tournament, "Impressive!" In this case, using a lookahead-only production nets a twenty percent performance gain. Due to issues with ignored syntactic lookahead, it's not always possible to write these in a complicated grammar. If you can figure out a way to make it work, though, writing and using a lookahead-only production to skip "deep" productions is worth the effort.

A few more notes on lookahead-only productions:

- While lookahead-only productions are no different syntactically than regular ENBF productions, they do differ in the way they're used. Naming them with a `Lookahead` suffix (e.g., `FunctionDeclarationLookahead`) is a helpful way to differentiate them from the nonterminals in the grammar. Your future grammar maintainer—and that may be you—will thank you!

- Remember that nested syntactic lookahead is ignored (page 93). Thus lookahead-only productions should be written knowing that any syntactic lookahead they contain will, by definition, be ignored. You may still need to include some multiple token lookahead directives within a lookahead-only production to silence

[4] javacc/examples/JavaGrammars/Java1.1.jj

JavaCC's warnings; that'll help keep the output tidy when you're regenerating the parser. If you do, it's a good idea to add a comment to indicate that you're aware that this lookahead will be ignored at runtime. You can see an example of this in the Java grammar that ships with JavaCC; the `CastLookahead` production is preceded by a comment, to the effect that "yes, this is ignored, but it keeps JavaCC happy."

Finally, credit goes to Dr. Kenneth Beesley for his online notes on lookahead-only productions [5]; these proved very helpful and the Java 1.1 grammar case I've benchmarked here is straight from his examples.

LL(k) and All That

Let's wrap up this chapter with a brief excursion into parsing theory and terminology. You'll sometime see JavaCC referred to as a parser generator that creates *LL(k)* parsers. This notation is commonly used to classify parsers based on how they process input data. Let's examine the LL(k) designation letter by letter:

- *L*: The first L indicates that the parser will scan the input data from left to right. Although they may exist, I have yet to encounter a parser that works from right to left.
- *L*: The second L indicates that JavaCC parsers use *left derivation* when building the parse trees. Left derivation means that the leftmost expansion is always processed first. So if a grammar's start symbol nonterminal consisted of two expansions `A()` and `B()`, the `A()` expansion would be processed first. If a parser processed `B()` first, that would naturally be a *right derivation*.
- *(k)*: This indicates that the generated parser can look ahead *k* tokens, where *k* is a positive integer. Some parsers can only look ahead one token and thus are designated *LL(1)* parsers—recall that this is the default behavior for JavaCC. The same principle applies to *LL(2)* parsers; they can look ahead no more than two tokens. But since you can use multiple token lookahead of any amount (via either a local lookahead directive or a global lookahead setting) JavaCC is *LL(k)*. Additionally, you can use syntactic and semantic lookahead, so another notation for parser generators that support such features is `LL(*)`.

You'll also see JavaCC and other *LL(k)* parsers referred to as *recursive descent parsers*. This terminology is remarkably straightforward; it reflects the fact that the "top", or the start symbol, of a JavaCC-generated parser is called first and control proceeds "down" into the child productions as it moves through the stream of input tokens. Sometimes you'll see these referred to as "top down" parsers.

Other parser generators create parsers that are *LR* (Left to right, Rightmost derivation) or *LALR* (Look Ahead, Left to right, Rightmost derivation). You'll also see both of these types with numeric suffixes indicating that they use zero, one, or *k* lookahead.

[5] http://www.engr.mun.ca/~theo/JavaCC-FAQ/kens-javacc-lookahead-summary.txt

Change Tracking

Once you've generated Java source code from your grammar, there's a high probability that you'll change the grammar and need to regenerate the source. But suppose you've modified the generated code? Will it be overwritten? Should it be? The answer is usually yes, but sometimes no. JavaCC 4.1 added mechanisms that allow you to know how a file was generated, and also allows JavaCC to determine whether the file can be safely regenerated.

Modified Files

Suppose you have a simple grammar that handles input of `hello world`:

```
# examples/parser/hello.jj

1    PARSER_BEGIN(Hello)
2    public class Hello {}
3    PARSER_END(Hello)
4    TOKEN : {
5     <HELLO : "hello">
6     | <WORLD : "world">
7    }
8    void hi() : {} {
9     <HELLO> <WORLD>
10   }
```

When we run this grammar through JavaCC it generates the usual flurry of files. Let's look at the last line of `SimpleCharStream.java`:

```
$ tail -1 SimpleCharStream.java
/* JavaCC - OriginalChecksum=f0bb7ce412d7b9984c49f747272736c8 (do not edit this
line) */
```

The `OriginalChecksum` value is, well, a checksum [6] of the contents of `SimpleCharStream.java`. With this in place, JavaCC can tell if a file has been modified since it was last generated. For example, let's append a comment to that file and then rerun JavaCC:

```
$ sed -i -e 's/Whether/Whether or not/' SimpleCharStream.java
$ javacc hello.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file hello.jj . . .
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
Warning: SimpleCharStream.java: File is obsolete.  Please rename or delete this
file so that a new one can be generated for you.
Parser generated with 0 errors and 1 warnings.
```

Note the warning; JavaCC did not regenerate the file since it had been changed. This is the JavaCC equivalent of a source code management system "conflict" - at this point the wisest move for JavaCC is to let the developer know that there's a problem and then punt. My usual tactic here is to check the commit logs for this file, see what changes were made, and regenerate the file and reapply those changes.

[6] To be precise, a Message Digest algorithm 5 (MD5) cryptographic hash

Generation Options

Let's look at another change tracking feature. JavaCC records the relevant options used to generate a file in a comment header. You can see this if we run our `hello.jj` through JavaCC and look at the first few lines of `SimpleCharStream.java`

```
$ javacc hello.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file hello.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ head -2 SimpleCharStream.java
/* Generated By:JavaCC: Do not edit this line. SimpleCharStream.java Version 4.1
*/
/* JavaCCOptions:STATIC=true,SUPPORT_CLASS_VISIBILITY_PUBLIC=true */
```

JavaCC has recorded the options which were used to generate the file. If we regenerate the same parser with a different set of options—say, with `STATIC` disabled—that change will be recorded in the options header:

```
$ javacc -NOSTATIC hello.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file hello.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ head -2 SimpleCharStream.java
/* Generated By:JavaCC: Do not edit this line. SimpleCharStream.java Version 4.1
*/
/* JavaCCOptions:STATIC=false,SUPPORT_CLASS_VISIBILITY_PUBLIC=true */
```

This is useful for manual file inspection - it lets you know what the relevant options were when this file was generated. However, it's also something that JavaCC checks when regenerating files. If you've modified a file, JavaCC will not only warn about the file having changed, it will also warn if you use different options. For example, let's generate the parser, modify `SimpleCharStream.java`, and then regenerated the grammar with a different set of options:

```
$ javacc hello.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file hello.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ sed -i -e 's/Whether/Whether or not/' SimpleCharStream.java
$ javacc -NOSTATIC hello.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file hello.jj . . .
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
Warning: SimpleCharStream.java: File is obsolete.  Please rename or delete this
file so that a new one can be generated for you.
Warning: SimpleCharStream.java: Generated using incompatible options. Please
```

```
rename or delete this file so that a new one can be generated for you.
Parser generated with 0 errors and 2 warnings.
```

This time we see two warnings: one about the obsolete file, and another about the incompatible options. Very handy!

Parser Options

This section will explore options that affect the generated parser.

CACHE_TOKENS

`CACHE_TOKENS` is a boolean valued option that's `false` by default. When set to true, it optimizes parser runtime performance by changing the way the parser requests tokens from the tokenizer. Rather than waiting until each token is requested by the next nonterminal, the parser reads one token ahead from the input stream so that there's always a token ready to be consumed. This avoids the overhead of the various comparisons and method calls needed to ensure there's a token available.

Enabling this option shrinks the generated parser slightly, since a parser field and various usages of that field go away. In the case of the Java grammar, the parser source file size dropped from 138K to 136K.

Several example grammars are included with JavaCC, and benchmarking `CACHE_TOKENS` with those grammars showed a slight runtime performance increase:

Table 3.2. CACHE_TOKENS Benchmark Results

Grammar	CACHE_TOKENS disabled	CACHE_TOKENS enabled	Performance gain
Java 1.5	662 ms	649 ms	2%
IDL	144 ms	142 ms	< 1%
C	948 ms	923 ms	3%

`CACHE_TOKENS` does have a downside. It may cause problems if it's used with interactive parsers—parsers that wait for tokens to become available from the input stream. You can see this problem by enabling `CACHE_TOKENS` in the JavaCC example grammar `examples/GUIParsing/ParserVersion/` and generating the grammar. When you compile and run the generated code, it'll pop up a small window with a simple calculator. When you click the buttons, it appears to ignore the first number that you click, but then it displays that number when you click a second time. That's because `CACHE_TOKENS` causes the parser to wait until a token has been formed from the input stream before calling the grammar's start symbol. You can see this more clearly by regenerating that example parser with the `DEBUG_PARSER` option, then watching the command line. You'll see that the call to the `Input` nonterminal (which is the start symbol for that grammar) doesn't happen until you click a button.

In summary, `CACHE_TOKENS` is worth enabling if you're using JavaCC in the usual "parse the entire input stream" manner. Remember to evaluate this option with your own sample data to see if you get improvements.

CHOICE_AMBIGUITY_CHECK

CHOICE_AMBIGUITY_CHECK is an integer valued option that defaults to 2. It controls, to an extent, how accurate the error message will be when JavaCC finds a choice conflict in a grammar.

For example, consider a grammar that parses out a small but cheery greeting:

Example 3.35. Greetings grammar

```
# examples/parser/choice_ambiguity_check.jj

1    PARSER_BEGIN(ChoiceParser)
2    public class ChoiceParser {}
3    PARSER_END(ChoiceParser)
4    void Start() : {} {
5      Hello1() | Hello2()
6    }
7    void Hello1() : {}
8    { "hello" "there" "happy" "world"}
9    void Hello2() : {}
10   { "hello" "there" "happy"}
```

When JavaCC examines this grammar with the default CHOICE_AMBIGUITY_CHECK value of 2, it warns that there's a choice conflict and that we'll need to use a lookahead of "3 or more":

```
$ javacc choice_ambiguity_check.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file choice_ambiguity_check.jj . . .
Warning: Choice conflict involving two expansions at
         line 5, column 3 and line 5, column 14 respectively.
         A common prefix is: "hello" "there"
         Consider using a lookahead of 3 or more for earlier expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

We can use a CHOICE_AMBIGUITY_CHECK setting of 5 to get a more definitive warning message; with this setting, the warning states that we should try a lookahead of 4:

```
$ javacc -CHOICE_AMBIGUITY_CHECK=5 choice_ambiguity_check.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file choice_ambiguity_check.jj . . .
Warning: Choice conflict involving two expansions at
         line 5, column 3 and line 5, column 14 respectively.
         A common prefix is: "hello" "there" "happy"
         Consider using a lookahead of 4 for earlier expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

This makes sense. We will indeed need to look ahead four tokens to decide whether we're dealing with the Hello1 or the Hello2 nonterminal.

CHOICE_AMBIGUITY_CHECK slows down the parser generation slightly, since it does extra analysis of the grammar to produce the more accurate error messages. But it only affects the speed at which JavaCC generates the parser; it has no effect on the

actual parser code that's generated. So, if you're having trouble pinpointing a complicated lookahead problem, temporarily bumping up CHOICE_AMBIGUITY_CHECK is worth a try.

DEBUG_LOOKAHEAD

DEBUG_LOOKAHEAD is a boolean valued option that's set to false by default. When set to true, it adds output to that already generated by the DEBUG_PARSER option. The additional debug output shows the actions of the lookahead specifications as they resolve choice conflicts.

For example, here's part of the DEBUG_PARSER output for the multiple token lookahead example on page 86:

```
Call:   PhoneNumber
  Call:   LocalNumber
    Call:   AreaCode
      Consumed token: <<THREE_DIGITS>: "123" at line 1 column 1>
```

And now with DEBUG_LOOKAHEAD enabled. Each time the lookahead specification causes the parser to examine a token it's reported, along with the expected token value:

```
Call:   PhoneNumber
  Call:   LocalNumber(LOOKING AHEAD...)
    Visited token: <<THREE_DIGITS>: "123" at line 1 column 1>;
      Expected token: <<THREE_DIGITS>>
    Visited token: <"-" at line 1 column 4>;
      Expected token: <"-">
    Visited token: <FOUR_DIGITS>: "1212" at line 1 column 5>;
      Expected token: <<FOUR_DIGITS>>
      Call:   LocalNumber
        Call:   AreaCode
          Consumed token: <<THREE_DIGITS>: "123" at line 1 column 1>
```

You can see the differences in the lookahead debugging output: when looking ahead, the tokens are being "visited"; with regular productions, they're being "consumed." That's what we'd expect, since the purpose of lookahead is to determine what should be consumed.

Note that DEBUG_LOOKAHEAD overrides DEBUG_PARSER. That is, if you set DEBUG_PARSER to false and DEBUG_LOOKAHEAD to true, JavaCC will detect the conflict and set DEBUG_PARSER to true for you. You'll also get a warning message:

```
$ javacc -nodebug_parser -debug_lookahead phone_lookahead.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file phone_lookahead.jj . . .
Warning: True setting of option DEBUG_LOOKAHEAD overrides false setting of option
 DEBUG_PARSER.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
```

Enabling DEBUG_LOOKAHEAD generates a flood of output, so naturally it results in a performance hit. You should only use this option when you're, well, debugging lookahead.

DEBUG_PARSER

DEBUG_PARSER is a boolean valued option that's set to false by default. When enabled, it adds copious debugging output to the generated parser. The parser size grows by around 15% and runtime performance drops dramatically due to the flood of messages.

Here's a look at the output that DEBUG_PARSER produces when it's enabled for the phone.jj grammar on page 68:

```
Call:   PhoneNumber
  Consumed token: <<THREE_DIGITS>: "555" at line 1 column 1>
  Consumed token: <"-" at line 1 column 4>
  Consumed token: <<THREE_DIGITS>: "555" at line 1 column 5>
  Consumed token: <"-" at line 1 column 8>
  Consumed token: <<FOUR_DIGITS>: "1212" at line 1 column 9>
  Consumed token: <<EOF> at line 1 column 12>
Return: PhoneNumber
```

Internally, JavaCC always inserts two methods to support tracing—enable_tracing and disable_tracing, but these methods are empty by default. If you enable DEBUG_PARSER, the same two methods are generated, but when called they set and unset a boolean field. Since these methods are always present, you can embed calls to them in your code without having to remove them when DEBUG_PARSER is disabled. Thus, you can disable debug output for a particular nonterminal with a syntactic action that calls disable_tracing, e.g.:

Example 3.36. Disabling DEBUG_PARSER output for a particular nonterminal

```
void SomeNonterminal() : {
  disable_tracing();
} {
 Various() Other() Expansions() {
  enable_tracing();
 }
}
```

You'll typically want to reenable tracing when leaving the nonterminal, as shown in the example code above.

ERROR_REPORTING

ERROR_REPORTING is a boolean valued option that defaults to true. If disabled, it results in less detailed runtime error messages from the parser. Here's a parsing error message with ERROR_REPORTING enabled:

```
$ java PhoneParser "888-1212"
ParseException: Encountered " <FOUR_DIGITS> "1212 "" at line 1, column 5.
Was expecting:
    <THREE_DIGITS> ...

 at PhoneParser.generateParseException(PhoneParser.java:209)
 at PhoneParser.jj_consume_token(PhoneParser.java:147)
 at PhoneParser.PhoneNumber(PhoneParser.java:20)
 at PhoneParser.main(PhoneParser.java:8)
```

And with ERROR_REPORTING disabled:

```
$ javacc -noerror_reporting phone_actions.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
```

```
(type "javacc" with no arguments for help)
Reading from file phone_actions.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PhoneParser "888-1212"
ParseException: Parse error at line 1, column 5.  Encountered: 1212
 at PhoneParser.generateParseException(PhoneParser.java:158)
 at PhoneParser.jj_consume_token(PhoneParser.java:124)
 at PhoneParser.PhoneNumber(PhoneParser.java:20)
 at PhoneParser.main(PhoneParser.java:8)
```

Note the "expected token" information isn't presented when `ERROR_REPORTING` is disabled.

This option is `true` by default, for good reason. If you disable it, the parser will produce error messages slightly more quickly, but you'll pay a high price—they'll be less readable.

FORCE_LA_CHECK

`FORCE_LA_CHECK` is a boolean valued option that defaults to `false`. If enabled, it forces JavaCC to perform a check at all choice points. This means that any choice conflict will produce a warning, even if it's being handled by a lookahead specification. It also means that clever tricks—like the one mentioned on page 107 near the end of the `OTHER_AMBIGUITY_CHOICE` option section—won't silence JavaCC's warning messages.

One possible use for this option is when you're starting to work with a grammar and hoping to remove choice conflicts by refactoring it a bit. Using `FORCE_LA_CHECK` can give you a quick rundown of the conflicts, and you can cherry pick the easy ones.

GRAMMAR_ENCODING

`GRAMMAR_ENCODING` is a string valued option that defaults to the empty string. It's used to specify the character encoding of a JavaCC grammar file. For example, if you were working on a Mac where the default encoding was MACROMAN and you had a grammar which was encoded using UTF-8 you could tell JavaCC to use UTF-8 when reading the grammar like this:

```
$ javacc -GRAMMAR_ENCODING=UTF-8 mygrammar.jj
```

This feature was added to JavaCC after the release of v4.2 and as of this writing is not yet included in a stable release. However, that will almost certainly have changed by the time you read this book, so you should be able to use this feature as you work with grammars.

JDK_VERSION

`JDK_VERSION` is a string valued option that defaults to `1.4`. It's used to specify a Java language version for JavaCC to target. If you specify `JDK_VERSION` to be `1.5`, JavaCC can use that setting to emit code that uses Java 1.5 language constructs like generics and enumerations.

JavaCC 4.2 uses `JDK_VERSION` in a number of locations. In several instances, JavaCC will handle exceptions differently for Java 1.4 and higher. Instead of just wrapping the exception message, it wraps the entire exception instance in a `RuntimeException`. There's also a case where JavaCC will produce a generically-typed `Vector` instance (specifically, a `Vector` of `int[]`) if `JDK_VERSION` is set to `1.5`.

Explicitly setting this option is a good idea. Not only does it make it clear which Java version you are targetting, it also leaves the door open for future versions of JavaCC to improve the parser generated from your grammar. Also, now that Java 1.4 has been end-of-lifed[7], look for JavaCC's default language generation to be Java 1.5 in a not-so-distant future JavaCC release.

LOOKAHEAD

The `LOOKAHEAD` option is an integer valued option that's set to `1` by default. It controls the global lookahead setting for the entire grammar.

Setting a global lookahead value can have a dramatic impact on parser performance. Depending on the grammar complexity and how much lookahead is already being used in the grammar, the parser can be up to 60% slower with a global lookahead setting of `3`:

Table 3.3. Global Lookahead Benchmark Results

Grammar	Default Lookahead	LOOKAHEAD:3	Performance loss
Java 1.5	748 ms	1839 ms	60%
IDL	152 ms	200 ms	24%
C	942 ms	1414 ms	33%

This performance degradation continues as you increase the global lookahead setting. If you have to use global lookahead, use the minimum amount possible.

OTHER_AMBIGUITY_CHECK

`OTHER_AMBIGUITY_CHECK` is an integer valued option that defaults to `1`. It's a bit like `CHOICE_AMBIGUITY_CHECK` (see page 101), but instead of controlling the number of tokens to check for alternation choices like `A | B | C`, it checks for quantifier choices like `(A)+`, `(A)*`, and `(A)?`.

For example, consider another greeting grammar. This grammar contains a `Hello1` nonterminal that consists of an optional `hello` followed by a single `hello`, a `happy`, and a `world`. There's also a `Hello2` nonterminal that begins with an optional `hello` followed by a `there` and a `world`:

[7] http://java.sun.com/j2se/1.4.2/

Example 3.37. Another greetings grammar

```
# examples/parser/other_ambiguity_check.jj

1    PARSER_BEGIN(ChoiceParser)
2    public class ChoiceParser {}
3    PARSER_END(ChoiceParser)
4    void Start() : {} {
5      Hello1() | Hello2()
6    }
7    void Hello1() : {} {
8      ("hello")? "hello" "happy" "world"
9    }
10   void Hello2() : {} {
11     ("hello")? "there" "world"
12   }
```

When JavaCC processes this grammar with the default `OTHER_AMBIGUITY_CHECK` setting, it produces a warning for both the alternation choice point in `Start` and the expansion `("hello")?` in `Hello1`:

```
$ javacc other_ambiguity_check.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file other_ambiguity_check.jj . . .
Warning: Choice conflict involving two expansions at
         line 5, column 3 and line 5, column 14 respectively.
         A common prefix is: "hello"
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict in [...] construct at line 8, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "hello"
         Consider using a lookahead of 2 or more for nested expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 2 warnings.
```

Note the "Consider using a lookahead of 2 or more for nested expansions" warning. We can clarify that somewhat by specifying a `OTHER_AMBIGUITY_CHECK` setting of `2`:

```
$ javacc -OTHER_AMBIGUITY_CHECK=2 other_ambiguity_check.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file other_ambiguity_check.jj . . .
Warning: Choice conflict involving two expansions at
         line 5, column 3 and line 5, column 14 respectively.
         A common prefix is: "hello"
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict in [...] construct at line 8, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "hello"
         Consider using a lookahead of 2 for nested expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 2 warnings.
```

Now the last warning states that we should "Consider using a lookahead of 2", which is a more exact suggestion.

Since a grammar like this begs to be cleaned up, let's fix it. We can fix the first warning easily, by specifying a multiple token lookahead specification of `2` in the `Start` production. We need to employ some subterfuge to fix the second warning. The lookahead specification in the `Start` production is sufficient to resolve the choice

conflict, since looking ahead two tokens will encompass the initial expansion in both `Hello1` and `Hello2` as well as the next token. But JavaCC doesn't detect that, so we can add a "we know what we're doing" lookahead specification that uses the default lookahead token value in `Hello1`:

Example 3.38. The greetings grammar fixed

```
# examples/parser/other_ambiguity_check_fixed.jj (lines 15 to 23)

15   void Start() : {} {
16     LOOKAHEAD(2) Hello1() | Hello2()
17   }
18   void Hello1() : {} {
19     (LOOKAHEAD(1) "hello")? "hello" "happy" "world"
20   }
21   void Hello2() : {} {
22     ("hello")? "there" "world"
23   }
```

Now JavaCC processes it without complaint:

```
$ javacc other_ambiguity_check_fixed.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file other_ambiguity_check_fixed.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

As an aside, note that if `Hello1` had started with `("hello")+` rather than `("hello")?` we would have had a problem on our hands. Why? Because `("hello")+` means "one or more occurrences of `hello`", which will gobble up all the `hello`s that it can find. Since the next token after that expansion is `hello`, the parser will consume all the `hello`s and then find that another one is expected, which will result in a lookahead failure. Thus, `Hello1` will never be entered, and `Hello2` will always be chosen.

SANITY_CHECK

`SANITY_CHECK` is a boolean valued option that defaults to `true`. When left at its default value, it causes JavaCC to perform a variety of checks to ensure that the grammar doesn't do anything silly, or even insane. It's the `SANITY_CHECK` option, for example, that controls the checks for left recursion and for choice conflicts.

Disabling `SANITY_CHECK` does result in a small performance increase when generating the parser; some rough benchmarking showed about a 5% savings in parser generation time. There's no runtime performance change at all since `SANITY_CHECK` doesn't affect the generated code.

If you're regenerating the parser frequently but not changing the grammar, you may want to disable `SANITY_CHECK` to save time. On the other hand, if there's any chance you'll make changes to the grammar, leaving `SANITY_CHECK` set to its default value will help ensure that your grammar will be sound. I've never seen a grammar in the wild with `SANITY_CHECK` disabled, and that's probably a good policy.

STATIC

The STATIC option has been covered in the previous chapter; see page 62 for more details on usage and performance. Generally, STATIC has the same effect on the parser as it does on the tokenizer; that is, a static modifier is added to all methods and fields.

SUPPORT_CLASS_VISIBILITY_PUBLIC

SUPPORT_CLASS_VISIBILITY_PUBLIC is a boolean valued option that defaults to true. With this option unspecified, JavaCC will generate various classes (xTokenManager, xCharStream, Token, and TokenMgrError) with an access modifier of public. If set to false, JavaCC will generate these source files with the access modified omitted, resulting in an access level of package private. You may wish to set this option to avoid cluttering your project's package namespace.

TOKEN_MANAGER_USES_PARSER

TOKEN_MANAGER_USES_PARSER is a boolean valued option that defaults to false. When set to true, it inserts various declarations that allow the token manager to access the parser. This can be handy if you want the token manager to pull some sort of application-specific configuration information from the parser. Since the token manager is usually only instantiated by way of creating a parser, it can be convenient to configure any needed context on the parser and let the token manager draw from there.

There are a few wrinkles with TOKEN_MANAGER_USES_PARSER:

- TOKEN_MANAGER_USES_PARSER will have no effect if STATIC is set to true. Since STATIC is true by default, this means that for TOKEN_MANAGER_USES_PARSER to have any effect, you'll need to explicitly set STATIC to false. Of course, disabling STATIC will result in a considerable performance hit; see page 62 for more on the effects of that choice.

- If you enable TOKEN_MANAGER_USES_PARSER but disable STATIC and BUILD_PARSER, JavaCC will generate a token manager that has references to a non-existent parser class, so it will fail to compile. The solution is to make a decision and remove the option setting for either TOKEN_MANAGER_USES_PARSER or BUILD_PARSER, since those options don't make sense together. Arguably, though, JavaCC should reject this set of options.

Summary

This chapter has outlined the JavaCC parser generation component. We've looked at basic parser generation, nonterminals, and expansions, and gotten a good handle on using EBNF to express our syntactic requirements. We've also seen numerous examples of JavaCC's lookahead types: multiple token, syntactic, and semantic. Armed with this information you approach the thorniest parsing problem with confidence!

Chapter 4. JJTree

One day the trees went out to anoint a king for themselves. --Judges 9:8

Why JJTree?

Chapter two explained how JavaCC's tokenizer reads in a data stream and converts characters into tokens. Chapter three showed the way that JavaCC's parser creates a derivation, or a parse tree, consisting of the nonterminals in the syntactic specification. In this chapter, we'll take the process one step further and show how you can use JavaCC's built-in *tree builder*, JJTree, to construct an *Abstract Syntax Tree*, or AST, from the parse tree. We'll also look at how you can customize the tree building process to create an AST that improves upon the parse tree.

Using JJTree to build an explicit tree structure atop a parse tree has its advantages:

- A tree structure makes a grammar's output easier to visualize and understand. If a `PhoneNumber` consists of an `AreaCode` and a `CityCode`, the latter two productions are below the `PhoneNumber` in the parse tree. You can make a parse tree more manageable by building an easily navigable tree structure with well defined types and interfaces from these nonterminals.
- Doing useful work with a parse tree involves embedding numerous syntactic actions within the body of the grammar, making the resulting mix of EBNF notation and Java syntax difficult to read. Just as importantly, most integrated development environments expect a file to contain only one language, so mixing languages in the same file reduces some of the advantage that an IDE offers. A JJTree grammar helps move much of that working code into Java source files that are easily analyzed and modified by a host of development tools.
- You can build a custom AST that improves on the parse tree in some way. For example, if your input data contains a series of letters, each in its own node, you could save memory and clutter by aggregating all those letters in a parent `Word` node and pruning the `Letter` nodes from the tree.

JJTree acts as a preprocessor for JavaCC grammars. That is, when you run JJTree on a grammar, it generates an enhanced JavaCC grammar that has lots of code inserted in the appropriate places—the code necessary to generate a tree structure. When we say that "JJTree builds a tree" what we really mean is that "JJTree inserts code into a JavaCC grammar which causes a tree structure to be created when the parser is generated from the grammar and executed with a data file." This can make talking about JJTree's functionality a bit unwieldy, so when it's clear what's happening, I'll use a phrase like "JJTree builds a tree." When we're discussing something that JJTree is doing explicitly (e.g., generating node source files), I'll try to make it clear whether I'm referring to JJTree in its role as a preprocessor or to JJTree's effect on the runtime behavior of the generated parser.

The rest of this chapter will be more understandable if we define some basic terms to use when talking about trees. A tree is a data structure that is based around a number of *nodes* connected by *branches* or *edges*. A node can be any sort of data structure; in the case of JJTree it's an implementation of a `Node` interface. The top-most node in a tree is called the *root*. A node can have other nodes below it; these are its *children*.

These children all have references extending back up to the higher level node, which is therefore referred to as their *parent*. Since child nodes can have children themselves, any node can be thought of as the root of a *subtree*. Children with the same parent are *siblings*, and continuing the genealogical analogy, a node's childrens' children are its *grandchildren*. Finally, nodes with no children are *leaf* nodes.

With that overview in mind let's plunge ahead with some examples that illustrate JJTree's capabilities.

A Simple Calculator

Let's look at the steps required to build a very simple calculator. An expression for our calculator to parse and solve might be an addition expression like `2+2`. Here's a grammar that models a calculator that handles only a single operator with two operands:

Example 4.1. A grammar for simple expressions

 # examples/jjtree/calculator_parser.jj (lines 15 to 34)

```
15   SKIP : {
16     " "
17   }
18   TOKEN : {
19     <DIGITS : (["0"-"9"])+>
20     | <PLUS : "+">
21   }
22   void Expression() : {} {
23     {System.out.println("Expression starts");}
24     Operator()
25     {System.out.println("Expression ends");}
26   }
27   void Operator() : {} {
28     Operand()
29     "+" {System.out.println("Operator: " + tokenImage[PLUS]);}
30     Operand()
31   }
32   void Operand() : {Token t;} {
33     t=<DIGITS> {System.out.println("Operand: " + t.image);}
34   }
```

When we run the parser, the syntactic actions print out the token images as the parser builds the parse tree:

```
$ java Calculator "4+2"
Expression starts
Operand: 4
Operator: "+"
Operand: 2
Expression ends
```

What's missing in this output? For one thing, there's no answer—it's not solving the expression. But let's leave that for later. More immediately, the output is not really representing the expression in a helpful manner. The `Operand` nonterminals are children of the `Operator` nonterminal, but this parent-child relationship isn't communicated effectively by a flat list of nonterminals and token images. We could add a parser field that incremented and decremented a `tree_depth` value and use that to add leading spaces, but that would clutter each nonterminal even more. Furthermore, in order to get just that list of nonterminals, we'd have to add boilerplate syntactic actions to each nonterminal. But avoiding boilerplate stuff is exactly what parser generators are for!

To crystalize the parse tree that JavaCC is building into a more useful data structure we need JJTree. JJTree will add a layer of Java types and structures to help us manipulate and traverse the input data more effectively.

A Simple Abstract Syntax Tree

Here's a modified version of the calculator grammar that's suitable for processing with JJTree. Notice how little of the existing JavaCC grammar needs to be changed to get a basic JJTree grammar up and running:

Example 4.2. A JJTree expression grammar

```
# examples/jjtree/calculator_tree/calculator.jjt

1    PARSER_BEGIN(Calculator)
2    import java.io.*;
3    public class Calculator {
4      public static void main(String[] args) {
5        Reader sr = new StringReader(args[0]);
6        Calculator p = new Calculator(sr);
7        try {
8          SimpleNode e = p.Expression();
9          e.dump(">");
10       } catch (ParseException pe) {
11         pe.printStackTrace();
12       }
13     }
14   }
15   PARSER_END(Calculator)
16   SKIP : {
17     " "
18   }
19   TOKEN : {
20     <DIGITS : (["0"-"9"])+> | <PLUS : "+">
21   }
22   SimpleNode Expression() : {} {
23     Operator()
24     {return jjtThis;}
25   }
26   void Operator() : {} {
27     Operand() "+" Operand()
28   }
29   void Operand() : {} {
30     <DIGITS>
31   }
```

A couple of new things appear in this grammar:

- The grammar file name has a `.jjt` suffix. This is not required, but it's a useful convention. When you run JJTree on a file, it derives the output file name by snipping off the input file name's suffix and adding `.jj`. So running a JJTree grammar named `ruby.jjt` through JJTree results in the creation of a new JavaCC grammar with a file name of `ruby.jj`. If you want to override this behavior you can use the `OUTPUT_FILE` option documented in detail on page 138.

- The `Expression` production now returns a `SimpleNode` type. `SimpleNode` is an implementation of the `Node` interface and will appear in the AST for each nonterminal. `Expression` uses a syntactic action to return the `SimpleNode`: `return`

`jjtThis;`. `jjtThis` is a reference to the current JJTree node; at runtime `jjtThis` refers to the variable of the appropriate type.

- The parser's `main` method now contains a call to `e.dump()` where `e` is an object of type `SimpleNode`. JJTree provides the `dump` method as a utility for displaying the structure and content of the AST. We're passing in a `>` to use as the prefix to each line that's printed out.

Those are the only changes we'll need to transform a JavaCC grammar into a basic JJTree grammar. In fact, if you don't want anything printed out, you could omit even these changes.

Here are the source files that JJTree generates from this grammar:

Example 4.3. JJTree-generated files

```
$ jjtree -JDK_VERSION=1.5 calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls -l *.java
-rw-r--r--  1 tom  wheel   406 Apr  2 00:39 CalculatorTreeConstants.java
-rw-r--r--  1 tom  wheel  3220 Apr  2 00:39 JJTCalculatorState.java
-rw-r--r--  1 tom  wheel  1282 Apr  2 00:39 Node.java
-rw-r--r--  1 tom  wheel  2195 Apr  2 00:39 SimpleNode.java
```

`CalculatorTreeConstants.java` is analogous to the constants file that JavaCC generates for the token types and names. In this case it contains a number of integers and strings representing the various node types. `JJTCalculatorState.java` contains code for tracking the construction of the AST. `Node.java` contains an interface which AST node classes must implement. Lastly, `SimpleNode.java` contains an implementation of `Node` as well as several convenience methods used to display contents of the AST.

Let's run both JJTree and JavaCC to see how JJTree displays the structure of the tree. Note that we have to pass in `JDK_VERSION` to both JJTree and JavaCC; JJTree won't pass through that option's value to the generated JavaCC grammar. This is a bug in JJTree that (hopefully!) will be fixed in the next release. At any rate, here's our trial run:

```
$ jjtree -JDK_VERSION=1.5 calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ javacc -JDK_VERSION=1.5 calculator.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file calculator.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

```
$ javac *.java
$ java Calculator "2+2"
>Expression
> Operator
>  Operand
>  Operand
```

That's rather nice in some ways, and not so nice in others. We can see the structure of the AST and the type of each of the nodes. While that's quite helpful, none of the token images are included, so we can't see that the `Operand` nodes had a value of `2` or that the `Operator` was a `+`.

We can fix the lack of token images by modifying the code that JJTree generated. First we'll add a new field, `image`, and a `setImage` method to `SimpleNode`. That lets us store the token's image, and then we can modify the `SimpleNode.toString` method to include that new `image` field. Since each AST node is an instance of `SimpleNode` that should cover the bases. We can add syntactic actions to the grammar to set those token images on the `SimpleNode`. Here's how the grammar looks with those syntactic actions:

Example 4.4. A calculator with values

 # examples/jjtree/calculator_tree_image/calculator.jjt (lines 23 to 34)

```
23   SimpleNode Expression() : {} {
24     Operator()
25     {return jjtThis;}
26   }
27   void Operator() : {Token t;} {
28    Operand()
29    t="+" {jjtThis.setImage(t.image);}
30    Operand()
31   }
32   void Operand() : {Token t;} {
33     t=<DIGITS> {jjtThis.setImage(t.image);}
34   }
```

Now when we run it we can see the details of the input data:

```
$ java Calculator "2+2"
Expression:null
 Operator:+
  Operand:2
  Operand:2
```

One thing our calculator doesn't do at this point is calculate. We can get that functionality by adding more code to our syntactic specification. For example, we can further decorate the syntactic action at the end of the `Expression` production by adding code that checks the value of the child nodes and adds those values together. The result can be stored in the image of the `Expression` node and the `main` method can print the result:

Example 4.5. Some unwieldy calculating

 # examples/jjtree/calculator_calculating/calculator.jjt

```
1    PARSER_BEGIN(Calculator)
2    import java.io.*;
3    import java.util.*;
4    public class Calculator {
5      public static void main(String[] args) {
6        Reader sr = new StringReader(args[0]);
7        Calculator p = new Calculator(sr);
8        try {
9          SimpleNode node = p.Expression();
10         System.out.println("The answer is " + node.image);
11       } catch (ParseException pe) {
12         pe.printStackTrace();
13       }
14     }
15   }
16   PARSER_END(Calculator)
17   SKIP : {
18     " "
19   }
20   TOKEN : {
21     <DIGITS : (["0"-"9"])+> | <PLUS : "+">
22   }
23   SimpleNode Expression() : {} {
24     Operator()
25     {
26       SimpleNode first = (SimpleNode)jjtThis.jjtGetChild(0).jjtGetChild(0);
27       int firstValue = Integer.parseInt(first.image);
28       SimpleNode second = (SimpleNode)jjtThis.jjtGetChild(0).jjtGetChild(1);
29       int secondValue = Integer.parseInt(second.image);
30       SimpleNode operator = (SimpleNode)jjtThis.jjtGetChild(0);
31       if (operator.image.equals("+")) {
32         jjtThis.setImage(String.valueOf(firstValue + secondValue));
33       } else {
34         System.out.println("Unknown operator");
35       }
36       return jjtThis;
37     }
38   }
39   void Operator() : {Token t;} {
40     Operand()
41     t="+" {jjtThis.setImage(t.image);}
42     Operand()
43   }
44   void Operand() : {Token t;} {
45     t=<DIGITS> {jjtThis.setImage(t.image);}
46   }
```

And to verify that it works:

```
$ java Calculator "2+2"
The answer is 4
```

A success! But that's a lot of fairly ugly code just to add two numbers. It doesn't quite seem like pushing all that code into a nonterminal is using the tree structure that JJTree is generating to its fullest potential. It's working, but we can do better.

To recap, we've written a very simple expression parser and seen how JJTree can preprocess it to generate code that builds an AST. We've also added some actions to get the value of the expression that we've parsed. In the next section we'll explore some facets of JJTree that will make our calculator both cleaner and more flexible.

Visitors and Visiting

As you think about the AST for the expression `2+2`, you can probably imagine several operations you could perform on that tree. For example, you may want to traverse the tree and print out all the node types and values. You may want to traverse selected parts of the tree and use an algorithm to produce a value, like the sum of all the values that are children of certain operators. Or you may want to scan the tree and collect a list of nodes that meet some criteria.

To a large degree, you can use built-in JJTree functionality to do those things. The primary means that JJTree provides for traversing tree structures is an implementation of the *Visitor pattern*. To quote from the classic book "Design Patterns", the Visitor pattern lets you "represent an operation to be performed on the elements of an object structure." A Visitor implementation is made of several components:

- There needs to be a data structure filled with objects. In JJTree's case, the tree structure of `Node` objects fills this role. Each of these node classes must implement a `jjtAccept` method that takes two arguments: a `Visitor` implementation and an `Object`. When you use the `MULTI` and `VISITOR` options together, JJTree will generate classes that have these methods.

- The first argument that's passed to the `jjtAccept` method mentioned above is an implementation of a `Visitor` interface type generated by JJTree. `Visitor` defines a `visit` method for each type of node. The node class calls this `visit` method and passes itself in. This is the essence of the Visitor pattern; the traversal process calls the node and passes in the `Visitor`, then the node turns around and calls the proper method (sometimes called a *callback method*) on the `Visitor`. Java's function overloading ensures that all these method calls are dispatched to the appropriate objects. With JJTree, this `Visitor` interface is generated only if you set the `VISITOR` option (see page 138) to `true`.

- While not strictly part of the `Visitor` pattern per se, providing an *adapter* is a useful idiom. This adapter avoids forcing each `Visitor` implementation to provide methods for all the AST types. Instead, the adapter provides a stubbed out version of the callback method for each type, and each `Visitor` implementation can extend the adapter and override the callbacks for only those specific node types that it wants to check.

Now let's use what we've learned about the Visitor pattern and JJTree's Visitor implementation details to improve the functionality of our calculator. We'll look at the changes to the grammar as well as the additional source files needed to enable the Visitor behavior. First, the grammar:

Example 4.6. A calculator grammar for use with a Visitor

```
# examples/jjtree/calculator_visitor/calculator.jjt
```

```
1    options {
2      MULTI=true;
3      VISITOR=true;
4      JDK_VERSION="1.5";
5    }
6    PARSER_BEGIN(Calculator)
7    import java.io.*;
8    public class Calculator {
9      public static void main(String[] args) {
10       Reader sr = new StringReader(args[0]);
11       Calculator p = new Calculator(sr);
12       try {
13         SimpleNode e = p.Expression();
14         SumVisitor v = new SumVisitor();
15         e.jjtAccept(v, null);
16         System.out.println("Sum is " + v.sum);
17       } catch (ParseException pe) {
18         pe.printStackTrace();
19       }
20     }
21   }
22   PARSER_END(Calculator)
23   SKIP : {
24     " "
25   }
26   TOKEN : {
27     <DIGITS : (["0"-"9"])+> | <PLUS : "+">
28   }
29
30   SimpleNode Expression() : {} {
31     Operator()
32     {return jjtThis;}
33   }
34   void Operator() : {Token t;} {
35     Operand()
36     t="+" {jjtThis.image = t.image;}
37     Operand()
38   }
39   void Operand() : {Token t;} {
40     t=<DIGITS> {jjtThis.image = t.image;}
41   }
```

Here are the changes we've made to the grammar:

- There's now an `options` section containing (in addition to `JDK_VERSION`) the new `VISITOR` and `MULTI` settings; both are `true`.
- The `PARSER_BEGIN/PARSER_END` section still contains the code to parse the expression, but now it also contains a call to `SumVisitor`, a new class that will traverse the tree and calculate a value. Our `Calculator` object then accesses that value and prints it out.
- We're using `jjtThis` to refer to the current AST node in all the syntactic actions. This is how we store data from the parsing process; we keep it in the AST nodes so that our Visitor implementations can access it later.
- We pass two arguments to `SimpleNode.jjtAccept`: the `Visitor` implementation, and a `null` value. The `null` value can be replaced with any object reference, and you can reference that object in your Visitor implementation. For example, you could pass in a `List` and your Visitor could add items to that `List`. In this case,

since we can get the calculation result by accessing the `sum` field on the `SumVisitor` object, we'll just pass in `null`.

- All the `Node.visit` methods return an `Object` by default, although you can modify that return type using the `VISITOR_RETURN_TYPE`. Usually this is the same object reference that's passed in as the second parameter to the root node's `visit` method, but it could be any type. For example, you could customize the root node to harvest any information it needs, generate the parser with `VISITOR_RETURN_TYPE="java.util.Map"`, return that `Map`, and the Visitor client could then use the `Map` in whatever way it needs. In any case, we're not using that object in this example so we'll just discard the reference that our root node returns.
- Notice how this grammar is much cleaner than previous grammar? This is because all the calculation code has been moved into the `SumVisitor` and all traversal code has moved into the classes generated by JJTree. All that's left in the grammar are the token definitions and the parse tree structure—as it should be.

Here's the `Visitor` interface definition that's generated by JJTree. You can see how a `visit` method has been generated for each node type:

Example 4.7. The Visitor interface

```
$ cat CalculatorVisitor.java
/* Generated By:JavaCC: Do not edit this line. CalculatorVisitor.java Version 4.2
 */
public interface CalculatorVisitor
{
  public Object visit(SimpleNode node, Object data);
  public Object visit(ASTExpression node, Object data);
  public Object visit(ASTOperator node, Object data);
  public Object visit(ASTOperand node, Object data);
}
/* JavaCC - OriginalChecksum=090fbf1a5e0687bca15ad8916031a5a4 (do not edit this
line) */
```

Since we've set `VISITOR` to `true`, here's a `CalculatorVisitorAdaptor` class that will make the `SumVisitor` much shorter. The default behavior for each node is to call `childrenAccept`, which is implemented in `SimpleNode`. That means, by default, that each node will take no action as it's traversed and will instead just recurse down into any child nodes:

Example 4.8. The Visitor adapter

 # examples/jjtree/calculator_visitor/CalculatorVisitorAdapter.java

```
1    public class CalculatorVisitorAdapter implements CalculatorVisitor
2    {
3      public Object visit(SimpleNode node, Object data) {
4        return node.childrenAccept(this, data);
5      }
6      public Object visit(ASTExpression node, Object data) {
7        return node.childrenAccept(this, data);
8      }
9      public Object visit(ASTOperator node, Object data) {
10       return node.childrenAccept(this, data);
11     }
12     public Object visit(ASTOperand node, Object data) {
13       return node.childrenAccept(this, data);
14     }
15   }
```

Now we'll get to the code that does the real work—the `SumVisitor`. This class is quite small since the tree traversal, the default behavior for uninteresting nodes, and the printing of the results are all done elsewhere. The only thing remaining in the `SumVisitor` is the actual aggregation and calculation, that is, the algorithm itself:

Example 4.9. The SumVisitor

 # examples/jjtree/calculator_visitor/SumVisitor.java

```
1    public class SumVisitor extends CalculatorVisitorAdapter {
2      public int sum;
3      public Object visit(ASTOperand operand, Object data) {
4        sum += Integer.parseInt(operand.image);
5        return super.visit(operand, data);
6      }
7    }
```

The `SumVisitor` overrides only one method from `CalculatorVisitorAdapter`: `visit(ASTOperand, Object)`. As mentioned earlier, this method, and all other `visit` methods, declare two parameters: the node type and an Object reference. We passed in `null` when we initiated this traversal (i.e., when we called `SimpleNode.jjtAccept`), so we won't use the second parameter. Instead we'll just check the `ASTOperand` node's image, extract the integer value, and add it to our accumulator.

The last line in the `SumVisitor.visit` method does two things. First, since we're calling the superclass' `visit` method, control passes up to the superclass, `CalculatorVisitorAdapter`. This means that the default node behavior is executed, so the traversal will continue and any child nodes will be visited. Second, we return the result (an `Object`) to the caller (the parent node).

`SumVisitor` performs its work of adding the operand's value to the sum before visiting any child nodes. This method of traversal is called *preorder traversal*. In the alternative *postorder traversal* method, the child nodes are visited first and then the current node is processed. Here's how that would look:

Example 4.10. A Postorder Traversal

 # examples/jjtree/calculator_visitor/PostorderSumVisitor.java

```
1    public class PostorderSumVisitor extends CalculatorVisitorAdapter {
2      public int sum;
3      public Object visit(ASTOperand operand, Object data) {
4        Object obj = super.visit(operand, data);
5        sum += Integer.parseInt(operand.image);
6        return obj;
7      }
8    }
```

You could also perform an operation both before and after any children are visited. To do that, insert code around the call to `super.visit`:

Example 4.11. Visiting Before and After Subtree Traversal

 # examples/jjtree/calculator_visitor/PreAndPostorderVisitor.java

```
1    public class PreAndPostorderVisitor extends CalculatorVisitorAdapter {
2      public Object visit(ASTOperand operand, Object data) {
3        System.out.println("Before visiting subtrees");
4        Object obj = super.visit(operand, data);
5        System.out.println("After visiting subtrees");
6        return obj;
7      }
8    }
```

This can be handy if you want to collect the subtree nodes of a certain type and then act on them. For example, you might initialize a `List` before the subtree is traversed, add each desired node to the `List` as the traversal occurs, then act on that `List` of nodes after control returns to the current node.

Another option you have when a node is being visited is to short-circuit the traversal. Suppose you know that, for the purposes of the visitor you're writing, the `ASTOperand` nodes will never have any interesting subtrees. You could return from the `visit` callback without calling `super.visit`:

Example 4.12. Short-circuiting the Traversal

 # examples/jjtree/calculator_visitor/ShortCircuitVisitor.java

```
1    public class ShortCircuitVisitor extends CalculatorVisitorAdapter {
2      public Object visit(ASTOperand operand, Object data) {
3        System.out.println("Short circuiting");
4        return data;
5      }
6    }
```

By returning immediately, you're avoiding the overhead of traversing selected subtrees resulting in a faster traversal. I use short-circuiting extensively in the Java code analysis tool PMD. For example, when I'm writing a PMD rule that checks method names only (not inner classes), I return immediately without calling `visit` in the `ASTMethodDeclarator` nodes. This skips the method bodies which could contain a considerable number of nodes.

Using VISITOR without MULTI

Back on page 116 we introduced a visitor example and noted that in the `options` section both the `MULTI` and the `VISITOR` options were set to `true`. This may make a bit more sense now that we've seen a couple of visitor examples. If you have `MULTI` enabled but `VISITOR` disabled you'll get a `Visitor` interface with just one method defined:

Example 4.13. Visitor with VISITOR enabled and MULTI disabled

 # examples/jjtree/calculator_no_multi/CalculatorVisitor.java

```
1    /* Generated By:JJTree: Do not edit this line. ./CalculatorVisitor.java */
2
3    public interface CalculatorVisitor
4    {
5      public Object visit(SimpleNode node, Object data);
6    }
```

By default, therefore, your `Visitor` implementation will contain just that one `visit` method. To figure out what node type is actually being visited in the callback you'll need to access the `SimpleNode.id` field. You can compare the `id` value to the integers in the `TreeConstants` interface or you can use the `id` field as an index into the `TreeConstants.jjtNodeName` array. Both techniques are shown in the following example.

Example 4.14. Determining Node Types Without `MULTI`

 # examples/jjtree/calculator_no_multi/NoMultiVisitor.java

```
1    public class NoMultiVisitor implements CalculatorVisitor {
2      public Object visit(SimpleNode node, Object data) {
3        System.out.println("SimpleNode.id = " + node.id);
4        System.out.println("Operand? " + (node.id ==
CalculatorTreeConstants.JJTOPERAND));
5        System.out.println("It's an " + CalculatorTreeConstants.jjtNodeName
[node.id]);
6        return node.childrenAccept(this, data);
7      }
8    }
```

And here's a sample run:

```
$ java Calculator "2+2"
SimpleNode.id = 0
Operand? false
It's an Expression
SimpleNode.id = 1
Operand? false
It's an Operator
SimpleNode.id = 2
Operand? true
It's an Operand
SimpleNode.id = 2
Operand? true
It's an Operand
```

Note that `NoMultiVisitor` doesn't extend an adapter. Since we're not using `MULTI` there's only one method in the interface, so an adapter wouldn't reduce the code size. It's simpler to just implement `CalculatorVisitor` and have our method invoke `childrenAccept` directly.

Tracking tokens

Earlier in this chapter you saw an example of collecting the token images by using syntactic actions to extract a token's image and attach it to the node. Another way to approach this problem is to use the `TRACK_TOKENS` option. Passing this option to JJTree generates additional code that attaches the first and last token associated with a node to that node and makes those tokens available via `SimpleNode.jjtGetFirstToken` and `SimpleNode.jjtGetLastToken`.

Here's an example of this option. We'll use the calculator grammar that we started with back on page 111 and display the `Operand` values, but we'll do so using `TRACK_TOKENS` and a custom `toString` method in `SimpleNode`. Here's the new `toString` method:

 # examples/jjtree/track_tokens/SimpleNode.java (lines 65 to 71)

```
65     public String toString() {
66       if (id == CalculatorTreeConstants.JJTOPERAND) {
67         return CalculatorTreeConstants.jjtNodeName[id] + " " +
```

```
jjtGetFirstToken().image;
68       } else {
69         return CalculatorTreeConstants.jjtNodeName[id];
70       }
71     }
```

Now we generate the parser with the `TRACK_TOKENS` option and compile and run the parser:

```
$ jjtree -TRACK_TOKENS -JDK_VERSION=1.5 calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ javacc -JDK_VERSION=1.5 calculator.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file calculator.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ cp /Users/tom/tom.infoether.com/gpwj/examples/jjtree/track_tokens/
SimpleNode.java .
$ javac *.java
$ java Calculator "2+3"
>Expression
> Operator
>  Operand 2
>  Operand 3
```

We can see that our customized `toString` method is only displaying the value for the `Operand` nodes, not the `Operand` node. Also, note that another way to approach this would be to generate the grammar with the `MULTI` options and only customize the `Operand` class' `toString` method.

The downside of using `TRACK_TOKENS` is that since each token stores a reference to the next token, attaching the tokens to the nodes will keep all those objects in memory. If you instead use syntactic actions to extract the token images as the AST is being constructed you'll save some memory - but at the expense of cluttering the grammar a bit.

Customizing the AST

So far we've seen examples of ASTs which directly reflect the nonterminals—the node names are the same as the nonterminal names, and the tree structure is the same as the parse tree structure. You can override this behavior with JJTree's *node descriptors*. Node descriptors let you change the name of the node, construct a node only under certain conditions, or prevent node construction entirely.

Let's look at a simple case first. To change the name of the class that's generated for a particular nonterminal, place a conditional descriptor of the form `#NewNodeName` after the nonterminal's name. For example, this grammar shortens up our calculator's node names:

Example 4.15. Different Names via Node Descriptors

```
# examples/jjtree/node_descriptor_name_change.jjt (lines 14 to 25)

14   SimpleNode Expression() #Expr : {} {
15     Operator()
16     {return jjtThis;}
17   }
18   void Operator() #Op : {} {
19     Operand()
20     "+"
21     Operand()
22   }
23   void Operand() #Oprnd : {} {
24     <DIGITS>
25   }
```

When we run this grammar through JJTree we can see that the class names are not the same as the nonterminal names:

```
$ jjtree node_descriptor_name_change.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file node_descriptor_name_change.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "ASTExpr.java" does not exist.  Will create one.
File "ASTOp.java" does not exist.  Will create one.
File "ASTOprnd.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "CalculatorVisitor.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./node_descriptor_name_change.jj
$ ls -l AST*
-rw-r--r--  1 tom  wheel  627 Apr  2 00:39 ASTExpr.java
-rw-r--r--  1 tom  wheel  619 Apr  2 00:39 ASTOp.java
-rw-r--r--  1 tom  wheel  631 Apr  2 00:39 ASTOprnd.java
```

As you can see, this is useful if you just want shorter class names. But it's also helpful if you want to coalesce several nonterminals into one node type. For example, the ECMAScript grammar for the Dojo project [1] contains nonterminals for `case` statements. These `case` statements can contain both regular case clauses and default clauses. Keeping these as separate nonterminals lets you perform different syntactic actions for each nonterminal and helps keep the grammar neater. But for the purposes of the AST, they can be combined into one node type. Here's the relevant snippet from that grammar; I've commented out the body of the productions to focus on the node descriptors:

Example 4.16. Combining Nodes with Descriptors

```
void CaseClause()  #CaseGroup : {}
{  /* node definition */ }
void DefaultClause() #CaseGroup : {}
{  /* node definition */ }
```

Thus both the `CaseClause` nonterminal and the `DefaultClause` nonterminal will be represented in the AST as a node of type `CaseGroup`. The AST will still contain the same structure and number of nodes, but this reduces the number of different types of nodes in the tree.

Another use of node descriptors is to suppress node creation entirely. To do this, use a node descriptor of `#void`:

[1] http://dojotoolkit.org/js-linker-dojo

Example 4.17. Suppressing Node Creation with #void

```
# examples/jjtree/node_descriptor_void.jjt (lines 26 to 35)

26   SimpleNode Expression() : {} {
27     Operator()
28     {return jjtThis;}
29   }
30   void Operator() #void : {} {
31     Operand() "+" Operand()
32   }
33   void Operand() : {} {
34     <DIGITS>
35   }
```

When we run this grammar we get an `Expression` with only the two `Operand` nodes as children; the `Operator` nodes are no longer in the AST:

```
$ java Calculator "2+2"
Expression
 Operand
 Operand
```

In other words, using a node description of `void` results in the no-longer-created node's children being attached to their grandparent.

Here's a real world `#void` example. Each Java type declaration has a parent `ASTModifiers` nonterminal that represents the access modifiers (`static`, `public`, `final`, etc) for that type declaration. In a Java grammar I work with, I use `#void` to suppress creation of the `ASTModifiers` node and instead roll that information into the type declaration. This prevents the creation of quite a few nodes since every class, method and field declaration would otherwise have a parent `ASTModifiers` node. The resulting tree is smaller, traversals are faster, and no relevant information is lost.

Definite Node Descriptors

Creating nodes isn't an all or nothing situation. You can also use *definite node descriptors* to cause a node to be created with a certain number of children. For example, consider a small language that consists of strings like `ABCCC`, where only the number of `C`s can vary[2]. If we want to ensure that a `B` node is created with two and only two `C` child nodes we can use a definite node descriptor like this:

Example 4.18. A Definite Node Descriptor

```
# examples/jjtree/definite_node.jjt (lines 19 to 28)

19   SimpleNode A() : {} {
20     "A" B()
21     {return jjtThis;}
22   }
23   void B() #B(2) : {} {
24     "B" (C())+
25   }
26   void C() : {} {
27     "C"
28   }
```

When we pass in `ABCC` we get a `B` with two `C` children:

```
$ java LetterParser "ABCC"
A
 B
```

[2]Contrived? Yes. But stay tuned for real-world node descriptor examples.

```
  C
  C
```

Passing in a `ABCCC` will produce a somewhat odd result, though. `B` is still constructed with two `C` nodes but the last `C` node ends up as a child of the `A` node:

```
$ java LetterParser "ABCCC"
A
 C
 B
  C
  C
```

This may or may not be what we really want. Either way, JJTree is just doing what we've told it to do. Here's how it works. JJTree constructs the tree from the bottom up; that is, it constructs the leaf nodes first and then attaches them to parent nodes. First it constructs the three `C` node objects. When it constructs `B`, it follows the node descriptor's instructions and attaches two of those `C` nodes to `B`. This leaves one `C` node unused or, more accurately, waiting to be used. When `A` is constructed, that `C` is attached as its child and the tree is complete.

If you run this grammar with an input string of `ABC` you'll get a somewhat unhappy result—an `EmptyStackException`. Here's why. The `C` node is constructed as usual. Next JJTree tries to construct the `B` node. It adds the first and only `C` node to `B` and then, per the node descriptor, tries to add another `C` node. This triggers an `EmptyStackException` since there's no additional `C` node to add. Interestingly, that particular `EmptyStackException` is caught by JJTree to give the parser a chance to handle the error. Because the `A` production doesn't do anything to recover from the failure to construct the `B` node, another `EmptyStackException` is triggered when JJTree tries to finish constructing the `A` node. This exception finally halts the process. The moral of the story is that, like the rest of JavaCC, JJTree will give you what you ask for.

A definite node descriptor argument can be any expression that results in an integer, so you could also specify the `2` argument in these ways:

Example 4.19. Specifying a Definite Node Descriptor

```
// You can use an arithmetic expression:
void B() #B(1+1)

// Or a reference to a field; perhaps one that you've declared in
// the grammar's PARSER_BEGIN/PARSER_END section:
void B() #B(this.nodeCount)

// Or a call to a method
void B() #B(this.loadNodeCount())
```

One thing to watch for—since node descriptor expressions can be evaluated multiple times, you shouldn't put anything in a node descriptor that would get confused if repeatedly invoked. Briefly stated, node descriptor expressions should be idempotent.

In my experience, definite node descriptors are rarely used. But they can be useful when you know exactly how many child nodes will be present and you want to group them in a very precise way.

Conditional Node Descriptors

A closely related (and somewhat more frequently used) construct is the *conditional node descriptor*. A conditional node descriptor lets you specify a condition under which a node will be created. This condition can be any expression that evaluates to a boolean value, and JJTree has a few shortcuts that are quite useful as well.

Let's start with the shortcuts since they handle most of the common cases. First, and simplest, the default value for a conditional node descriptor expression is `true`; this value is inserted even if you do not enter it explicitly. This means that rather than having to type `#MyNode(true)` to generate an AST node you can just type `#MyNode`. If you see a grammar littered with node descriptors in the form `#MyNode(true)`; these descriptors can be safely removed.[3]

The second shortcut lets you specify a minimum number of child nodes which must be present for a node to be constructed. Suppose in our `ABC` grammar we only want to create a `B` node if there are more than two child `C` nodes. In this case we can use a conditional node descriptor shortcut: `>2`. JJTree expands this shortcut to call the `nodeArity` method in the `JJTLetterParserState` class and determine if the result is greater than the specified minimum. Here's the grammar:

Example 4.20. A Conditional Node Descriptor

```
# examples/jjtree/conditional_node/conditional_node.jjt (lines 19 to 28)

19   SimpleNode A() : {} {
20     "A" B()
21     {return jjtThis;}
22   }
23   void B() #B(>2) :  {} {
24     "B" (C())+
25   }
26   void C() : {} {
27     "C"
28   }
```

When we run this with the input string `ABCCC` we see that the `B` node is constructed and all three `C` nodes are attached to it. If we pass in only `ABCC`, no `B` node appears; the two `C` nodes are attached to the `A` node instead:

```
$ java LetterParser "ABCCC"
A
 B
  C
  C
  C
$ java LetterParser "ABCC"
A
 C
 C
```

In addition to the `true` default value and the "greater than" shortcuts, almost any Java expression can be used as a conditional node descriptor. The only constraint lies in the way that JJTree generates the tree building code for the node descriptor: the node descriptor expression is placed in a call to the `JJTreeParserState` class' `closeNodeScope` method. If you use a method invocation (e.g., `shouldConstruct()`) for a conditional node descriptor, it will be placed into a method call in the form `closeNodeS-`

[3]See **`org.javacc.jjtree.ASTNodeDescriptor.closeNode()`** for the proof. Yet another advantage of open source!

`cope(node, shouldConstruct())`. This means you can only use Java code that's valid in that context—so, a class declaration would not work. Of course, you can achieve the same effect by calling a method that contains an inner class declaration, so practically speaking this isn't much of a limitation.

Embedded Node Descriptors

You can also embed node descriptors in the body of a nonterminal. This is handy if you want to create additional nodes as the tree is being constructed. For example, suppose we want to create an additional `B` node for the first two `C` nodes. We could add a definite node descriptor after the `(C())+` expansion, like this:

Example 4.21. Embedding a Node Descriptor

```
# examples/jjtree/node_descriptor_in_body/node_descriptor_in_body.jjt (lines 19
to 27)

19   SimpleNode A() : {} {
20     "A" B() {return jjtThis;}
21   }
22   void B() :  {} {
23     "B" (C())+ #B(2)
24   }
25   void C() : {} {
26     "C"
27   }
```

With this descriptor in place we get an extra parent `B` node in the tree when we pass in `ABCC`:

```
$ java LetterParser "ABCC"
A
 B
  B
   C
   C
```

The same behavior we've seen before holds: if we pass in three `C`s, the third `C` is moved up a level and assigned as a child to the topmost `B`:

```
$ java LetterParser "ABCCC"
A
 B
  C
  B
   C
   C
```

Let's combine an embedded definite node descriptor with a production-level conditional node descriptor. Using a node descriptor of `#B(>1)` will create the higher level `B` node only if `B` has more than one child node:

Example 4.22. Combining Node Descriptors

```
# examples/jjtree/node_descriptor_in_body/combined_node_descriptors.jjt (lines 19
 to 27)

19   SimpleNode A() : {} {
20     "A" B() {return jjtThis;}
21   }
22   void B() #B(>1) : {} {
23     "B" (C())+ #B(2)
24   }
25   void C() : {} {
26     "C"
27   }
```

You can see that two `B` nodes are created when we pass in `ABCCC`, but only one `B` node is created when we pass in `ABCC`. That one `B` node is created by the definite node descriptor `#B(2)`, and the other `B` node is suppressed by the production-level node descriptor `#B(>1)`:

```
$ java LetterParser "ABCCC"
A
 B
  C
  B
   C
   C
$ java LetterParser "ABCC"
A
 B
  C
  C
```

You can use definite and conditional node descriptors to do quite a bit of complex tree manipulation as the tree is being constructed. This can save on memory if you create fewer nodes. Just as importantly, the node descriptor code is fairly concise—much more concise than the code that would be required to manipulate the tree after it is built. If you find yourself adding or removing nodes in a sort of post-processing visitor, take a look at your nonterminals and see if you can use node descriptors to your advantage.

JJTree Internals

Now that you've seen JJTree from the outside it's worth taking a closer look at how JJTree works internally as it builds the tree. Let's look at a syntactic specification from the definite node descriptor examples:

Example 4.23. JJTree Internals Example

```
# examples/jjtree/definite_node.jjt (lines 19 to 28)

19   SimpleNode A() : {} {
20     "A" B()
21     {return jjtThis;}
22   }
23   void B() #B(2) : {} {
24     "B" (C())+
25   }
26   void C() : {} {
27     "C"
28   }
```

And here's a sample run with input data of `ABCCC`:

```
$ java LetterParser "ABCCC"
A
 C
 B
  C
  C
```

As mentioned earlier, JJTree constructs the AST from the bottom up. But it creates a `Node` object and opens a new *node scope* for each nonterminal as that nonterminal is expanded. This means that when JJTree expands the first nonterminal, `A`, it creates a new `SimpleNode` object. It uses the `SimpleNode` constructor that accepts a constant from the `TreeConstants` interface; in this case, it's `JJTA`. If we had generated this parser with the `MULTI` option JJTree would have created an `ASTA` object instead. After constructing the `SimpleNode` object, JJTree puts a placeholder `Integer` object on an internal stack to represent the newly opened scope.

After creating the `SimpleNode` object, it calls that object's `jjtOpen` method. The `jjtOpen` method is empty by default in `SimpleNode`, so you'd need to override this method in a particular `Node` implementatation class if you wanted to take some action. This is useful when you only have a few nodes that need this capability and you want to avoid the overhead of using `NODE_SCOPE_HOOK`, which invokes `jjtreeOpenNodeScope` and `jjtreeCloseNodeScope` for every `Node` implementation. If you use both `jjtOpen` and `NODE_SCOPE_HOOK`, `jjtOpen` will be called first.

After opening a new node scope for `A`, JJTree processes the expansions within that production. In the example above, JJTree will open a scope for `A` and then, since `A` contains an expansion of `B`, JJTree will open a new node scope for `B`. Next, since `B` contains an expansion for `C`, JJTree opens a new scope for `C`. At this point, JJTree has an internal stack of open scopes that is three levels deep.

As discussed earlier, conditional nodes default to `true`. Internally, JJTree treats every nonterminal as a conditional node unless otherwise specified. Therefore, `C` is treated as a conditional node with a `true` condition and JJTree adds the `SimpleNode` object to a stack. This happens for each of the three `C` nonterminals, so after processing the three `C` nonterminals we have a stack of three `SimpleNode` objects waiting to be attached to a parent node.

The three `C` nodes are all created within the scope of the `B` node, so normally they would all be attached as children to that `B` node. However, in the grammar we defined `B` as being a definite node with two children (e.g., `#B(2)`), so only the top two `C` nodes get attached to `B`. The third `C` node is left on the stack. JJTree closes the scope for `B`, attaches the children, sets their parent node to `B`, calls `SimpleNode.jjtClose` for the `B` node, and pushes the completed `B` node onto the stack.

The same process happens with the `A` nonterminal. It's a conditional node with a default expression of `true`, so all the nodes within its scope are added as children. In this case that's the `B` node that's on top of the stack, and then a `C` node. This was the first `C` node to have its node scope closed; it was the last `C` in our input data of `ABCCC`. After adding those two child nodes, JJTree calls `jjtClose` on `A`, pushes `A` on the stack, and the tree is done.

An Alternative: Java Tree Builder (JTB)

Introduction to JTB

This chapter focuses on JJTree since that's the tree builder that ships with JavaCC. However, there's another option for tree building: the *Java Tree Builder*, or JTB. JTB was started at Purdue University and it now resides at the University of California at Los Angeles (UCLA).

As you've seen, using JJTree involves taking a JavaCC grammar and annotating it to ensure that JJTree builds the AST structure that you want at runtime. JTB takes a different approach; it requires no grammar changes at all. Instead, you run JTB on a standard JavaCC grammar file and get an annotated grammar as a result. Then you run JavaCC on that annotated grammar file and write visitors to use and manipulate the AST. This keeps your tree building code out of the grammar; in fact, the JTB documentation encourages you to avoid putting even a `main` method in the grammar file.

A Quick Tour

Let's run through an example of JTB usage. First visit the JTB home page[4] and locate the jar file that contains the latest release (1.3.2 as of this writing). Once you download `jtb132.jar` and place it on your `CLASSPATH` you're ready to go.

Here's a small JavaCC grammar we can use for experimentation; just a list of phone numbers:

```
# examples/jjtree/jtb/phone.jj

1    PARSER_BEGIN(PhoneParser)
2    public class PhoneParser {}
3    PARSER_END(PhoneParser)
4    SKIP : { " " }
5    TOKEN : {
6      <FOUR_DIGITS : (<DIGITS><DIGITS><DIGITS><DIGITS>)>
7      | <THREE_DIGITS : (<DIGITS><DIGITS><DIGITS>)>
8      | <#DIGITS : ["0"-"9"]>
9    }
10   void PhoneList() : {} {
11     (PhoneNumber())+
12   }
13   void PhoneNumber() : {} {
14     <THREE_DIGITS> "-" <THREE_DIGITS> "-" <FOUR_DIGITS>
15   }
```

We can process this grammar with JTB by passing it in on the command line. JTB produces an annotated grammar in `jtb.out.jj`, a set of visitors in the `visitor` directory, and syntax tree nodes in the `syntaxtree` directory. In the example below we're passing in two options to JTB. The first, `-scheme`, generates the `SchemeTreeBuilder` visitor implementation which generates an AST in the Scheme programming language[5]. The second, `-printer`, generates the `TreeDumper` visitor, which allows us to print the contents of the AST. Here's the trial run:

[4] http://compilers.cs.ucla.edu/jtb/
[5] http://www-swiss.ai.mit.edu/projects/scheme/

```
$ java -jar /usr/local/jtb/jtb132.jar -scheme -printer phone.jj
JTB version 1.3.2
JTB:  Reading from phone.jj...
JTB:  Input file parsed successfully.
JTB:  "jtb.out.jj" generated to current directory.
JTB:  Syntax tree Java source files generated to directory "syntaxtree".

JTB:  "GJVisitor.java" generated to directory "visitor".
JTB:  "Visitor.java" generated to directory "visitor".
JTB:  "GJNoArguVisitor.java" generated to directory "visitor".
JTB:  "GJVoidVisitor.java" generated to directory "visitor".
JTB:  "GJDepthFirst.java" generated to directory "visitor".
JTB:  "DepthFirstVisitor.java" generated to directory "visitor".
JTB:  "GJNoArguDepthFirst.java" generated to directory "visitor".
JTB:  "GJVoidDepthFirst.java" generated to directory "visitor".

JTB:  "SchemeTreeBuilder.java" generated to directory "visitor".
JTB:  "records.scm" generated to current directory.

JTB:  "TreeDumper.java" generated to directory "visitor".
JTB:  "TreeFormatter.java" generated to directory "visitor".
$ javacc jtb.out.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file jtb.out.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ ls -l *.java *.jj
-rw-r--r--  1 tom  wheel   6147 Apr  2 00:40 ParseException.java
-rw-r--r--  1 tom  wheel   7586 Apr  2 00:40 PhoneParser.java
-rw-r--r--  1 tom  wheel    624 Apr  2 00:40 PhoneParserConstants.java
-rw-r--r--  1 tom  wheel   7974 Apr  2 00:40 PhoneParserTokenManager.java
-rw-r--r--  1 tom  wheel  12144 Apr  2 00:40 SimpleCharStream.java
-rw-r--r--  1 tom  wheel   4055 Apr  2 00:40 Token.java
-rw-r--r--  1 tom  wheel   4399 Apr  2 00:40 TokenMgrError.java
-rw-r--r--  1 tom  wheel   1244 Apr  2 00:40 jtb.out.jj
-rw-r--r--  1 tom  wheel    359 Apr  2 00:40 phone.jj
```

Now that we've generated code with JTB, let's look at a utility class that demonstrates running the `SchemeTreeBuilder` and the `TreeDumper` visitors. Notice that the visitors work the same way they did with JJTree—after creating the visitor you invoke a top-level `accept` method on the object structure and the generated code takes care of the traversal.

 # examples/jjtree/jtb/Runner.java

```
1    import java.io.*;
2    import visitor.*;
3    import syntaxtree.*;
4    public class Runner {
5      public static void main(String[] args) {
6        Reader sr = new StringReader(args[0]);
7        PhoneParser p = new PhoneParser(sr);
8        try {
9          PhoneList pl = p.PhoneList();
10         pl.accept(new TreeDumper());
11         System.out.println("");
12         pl.accept(new SchemeTreeBuilder());
13         System.out.println("");
14       } catch (ParseException pe) {
15         pe.printStackTrace();
16       }
17     }
18   }
```

And the `Runner` class in action with several phone numbers:

```
$ javacc -JDK_VERSION=1.5 jtb.out.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file jtb.out.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac -cp . *.java syntaxtree/*.java visitor/*.java
$ java -cp . Runner "432-789-9876 123-456-7890 888-555-1212"
432-789-9876 123-456-7890 888-555-1212
(define root '(PhoneList ((PhoneNumber "432" "-" "789" "-" "9876" ) (PhoneNumber
"123" "-" "456" "-" "7890" ) (PhoneNumber "888" "-" "555" "-" "1212" ) ) ))
```

If we wanted to customize the output of, say, the `TreeDumper`, we could either edit the `visitor/TreeDumper` and change it to suit our needs or we could write a new visitor. There are a variety of other visitor implementations that are autogenerated into the `visitors` directory.

JTB Summary

You may encounter a few bumps in the road with JTB. It's based on an older version (0.6) of JavaCC's grammar, so it doesn't support all the current JavaCC grammar constructs. For example, JTB can't parse token definitions containing repetition ranges (e.g., `(<SOME_TOKEN>){3}`). In some cases it generates code that JavaCC can't parse, such as productions that contain code like `t=n1=<SOME_TOKEN>`. Finally, any `JAVACODE` nonterminals in a grammar require some manual intervention. With these limitations, you might hesitate to use JTB for new parsing projects—although, by the time you read this, as is always true with open source software, some enterprising JTB developer may well have brought this utility up to date.

You can find more JTB documentation on the JTB web site; it discusses a slightly older version of JTB but is still mostly accurate. There seem to be a fair number of people using JTB for various parsing tasks, and it's definitely worth a look as an alternative to JJTree.

JJTree Options

This section explores the various options that affect JJTree. Options that are not used by JJTree (for example, `CACHE_TOKENS`) as well as options that apply to both JJTree and JavaCC (like `JDK_VERSION` and `STATIC`) are passed through and placed in the generated JavaCC grammar. JJTree checks all options and rejects a setting if the option's value is not appropriate, e.g., `CACHE_TOKENS="hello";`.

You can get a list of all the options which are specific to JJTree by running the `jjtree` shell script or Windows batch file with no arguments.

BUILD_NODE_FILES

`BUILD_NODE_FILES` is a boolean valued option that defaults to `true`. Disabling `BUILD_NODE_FILES` prevents node source files from being generated. In this example, notice that only the `Node` interface file is created and there's no `SimpleNode.java`:

Example 4.24. `BUILD_NODE_FILES` first `true` and then `false`

```
$ jjtree calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls *.java
CalculatorTreeConstants.java
JJTCalculatorState.java
Node.java
SimpleNode.java
$ rm -f *.java *.jj
$ jjtree -NOBUILD_NODE_FILES calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls *.java
CalculatorTreeConstants.java
JJTCalculatorState.java
Node.java
```

This option would be handy if you wanted to roll your own complete implementation of the node classes, or, less dramatically, if you wanted to ensure no one accidentally overwrote node source files that had been modified. Of course, you have those files safely stowed in a version control system such as Subversion[6], right?

JJTREE_OUTPUT_DIRECTORY

`JJTREE_OUTPUT_DIRECTORY` is a string valued option that defaults to the value of the `OUTPUT_DIRECTORY` option (see page 62). You can use this option if you want JJTree to put the files that it generates into one directory and JavaCC to put its files in a different directory.

If you use this option you'll probably want to use the `NODE_PACKAGE` option (page 135) as well to ensure the source files can be compiled.

MULTI

`MULTI` is a boolean valued option that defaults to `false`. When `MULTI` is set to `true`, a separate source file is created for each AST node type. For example, here are the JJTree generated files without `MULTI`:

[6] http://subversion.tigris.org/

Example 4.25. MULTI set to false

```
$ jjtree calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls *.java
CalculatorTreeConstants.java
JJTCalculatorState.java
Node.java
SimpleNode.java
```

And with MULTI set to true. Note that a source file for each node type appears:

Example 4.26. MULTI enabled

```
$ jjtree -multi calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "ASTExpression.java" does not exist.  Will create one.
File "ASTOperator.java" does not exist.  Will create one.
File "ASTOperand.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls *.java
ASTExpression.java
ASTOperand.java
ASTOperator.java
CalculatorTreeConstants.java
JJTCalculatorState.java
Node.java
SimpleNode.java
```

If a nonterminal has a conditional node descriptor of #void, the node source file will not be generated even if MULTI is enabled. This is reasonable since a #void node descriptor means that the node will not be created.

Most of the JJTree grammars I've seen have this option enabled; this is understandable since MULTI is required to make the runtime object tree traversable via the Visitor pattern.

NODE_CLASS

NODE_CLASS is a string valued option that defaults to an empty string. Setting the NODE_CLASS option along with MULTI=falsewill cause the generated parser to create instances of the specified class rather than of SimpleNode. If MULTI=true, the AST node classes will extend the specified class rather than extending SimpleNode. Note that the class you specify will not be generated; you'll need to implement it yourself and move it into the proper package before your parser and node classes will compile.

SimpleNode.java is still generated even if you set a value for NODE_CLASS; the idea is that your custom class will subclass SimpleNode. This allows you to override vari-

ous SimpleNode methods and also gives you access to SimpleNode members with protected access: parent, children, id, value, and parser.

JJTree doesn't do any checking of the NODE_CLASS value; you can set a type name that doesn't exist or even a type name that's not syntactically valid, like MyBase***Class. You'll discover your error in short order, though, when the generated code won't compile.

If you set MULTI=true along with a NODE_CLASS value that's not in the same package as the generated node source code you'll need to edit each node implementation and add an import statement to gain access to the NODE_CLASS type. I've found that I typically generate the node classes only once, so this manual edit should be a one-time event. Alternatively, you can use a fully qualified name like NODE_CLASS="com.company.BaseNode" and then even the one-time edit goes away.

Note that in JavaCC 4.2 this option does not appear in the output when you run jjtree with no options. This is a bug which has been fixed in the source repository and will be included in the next release.

NODE_DEFAULT_VOID

NODE_DEFAULT_VOID is a boolean valued option that defaults to false. When NODE_DEFAULT_VOID is set to true, each nonterminal will have a node descriptor of void by default. This means that instances of the node classes won't be created at tree construction time unless you manually add a node descriptor.

Suppose you want all your node class names to be different than your nonterminal names, or you're doing a great deal of specialized node creation in your grammar (like combining a dozen nonterminals into one tree node type). That's when this option comes in handy. You can eliminate a slew of #void descriptors with NODE_DEFAULT_VOID.

NODE_EXTENDS

NODE_EXTENDS is a string valued option that defaults to an empty string. Setting the NODE_EXTENDS option will cause the generated SimpleNode class to subclass whatever type you give to this option. This option has been deprecated in JavaCC 4.1. You should use NODE_CLASS (page 133) instead as it is more flexible.

NODE_FACTORY

NODE_FACTORY is a boolean valued option that defaults to false. When set to true, JJTree adds two new static factory methods to each node source file and the parser creates node objects by invoking these methods. Although the usual constructors are still generated into the node source files, they won't be called by the parser.

NODE_FACTORY appears to be a leftover from the early days of JJTree; I wasn't able to find any grammars in the wild with this option enabled.

NODE_PACKAGE

NODE_PACKAGE is a string valued option that defaults to an empty string. When NODE_PACKAGE is set to a specified string, the filename for each file that JJTree generates will begin with a package statement with that string as the package name. JJTree will also insert an import statement into the .jj file that will import the node classes. For example, if you have NODE_PACKAGE set to com.acme.nodes, JJTREE_OUTPUT_DIRECTORY set to com/acme/nodes, and OUTPUT_DIRECTORY set to com/acme/parser, you'll end up with a parser in com/acme/parser/ that contains the proper package statements and also imports the node files (via a wildcard import) from com/acme/nodes/.

JJTree does not do any syntax checking on NODE_PACKAGE, so you can set it to an invalid package name and JJTree will happily generate invalid source files for you. Avoid doing this, it annoys the compiler.

NODE_PREFIX

NODE_PREFIX is a string valued option that defaults to AST. This option prefixes each node class name with the option value.

Here's a sample run of JJTree that shows how setting NODE_PREFIX alters the class names:

Example 4.27. Source files with and without a custom NODE_PREFIX

```
$ jjtree -MULTI calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "ASTExpression.java" does not exist.  Will create one.
File "ASTOperator.java" does not exist.  Will create one.
File "ASTOperand.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ rm -f *.java *.jj
$ jjtree -MULTI -NODE_PREFIX=Foo calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "FooExpression.java" does not exist.  Will create one.
File "FooOperator.java" does not exist.  Will create one.
File "FooOperand.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
```

As you can see, this option doesn't affect the Node interface or the SimpleNode base class; those names remain unchanged. As with NODE_CLASS, JJTree doesn't check the NODE_PREFIX setting to ensure it's a syntactically valid Java class name; that's up to you.

NODE_SCOPE_HOOK

NODE_SCOPE_HOOK is a boolean valued option that defaults to false. When NODE_SCOPE_HOOK is set to true, JJTree inserts calls to two predefined methods each time a node scope is entered and exited.

Here's an example of NODE_SCOPE_HOOK in action; this grammar uses hook methods that print a nicely indented list of node types as the scopes are opened and closed:

Example 4.28. Indenting with NODE_SCOPE_HOOK

```
# examples/jjtree/node_scope_hook/node_scope_hook.jjt

1    options {
2      MULTI=true;
3      NODE_SCOPE_HOOK=true;
4    }
5    PARSER_BEGIN(Hook)
6    import java.io.*;
7    public class Hook {
8      public static int depth;
9      public static void main(String[] args) {
10       Reader sr = new StringReader(args[0]);
11       Hook p = new Hook(sr);
12       try {
13         p.Expression();
14       } catch (ParseException pe) {
15         pe.printStackTrace();
16       }
17     }
18     private static String pad() {
19       StringBuffer sb = new StringBuffer();
20       for (int i=0; i<depth; i++) {
21         sb.append(" ");
22       }
23       return sb.toString();
24     }
25     public static void jjtreeOpenNodeScope(Node n) {
26       System.out.println(pad() + "Opening scope for " + n.getClass());
27       depth++;
28     }
29     public static void jjtreeCloseNodeScope(Node n) {
30       depth--;
31       System.out.println(pad() + "Closing scope for " + n.getClass());
32     }
33   }
34   PARSER_END(Hook)
35   SKIP : {
36     " "
37   }
38   TOKEN : {
39     <DIGITS : (["0"-"9"])+> | <PLUS : "+">
40   }
41   void Expression() : {} {
42     Add()
43   }
44   void Add() : {} {
45     Operand() ("+" Operand())*
46   }
47   void Operand() : {Token t;} {
48     <DIGITS>
49   }
```

And in action:

```
$ java Hook "2+2"
Opening scope for class ASTExpression
 Opening scope for class ASTAdd
```

```
  Opening scope for class ASTOperand
  Closing scope for class ASTOperand
  Opening scope for class ASTOperand
  Closing scope for class ASTOperand
 Closing scope for class ASTAdd
Closing scope for class ASTExpression
```

In old grammars you'll see NODE_SCOPE_HOOK used to record the first and last tokens of a particular node so that line and column numbers can be extracted later. For example:

Example 4.29. Recording node start/end with NODE_SCOPE_HOOK

```
void jjtreeOpenNodeScope(Node node) {
  ((SimpleNode)node).setFirstToken(getToken(1));
}
void jjtreeCloseNodeScope(Node node) {
  ((SimpleNode)node).setLastToken(getToken(0));
}
```

For the above code to work you'd also need to define the setFirstToken and setLastToken methods in SimpleNode. However, this technique is out of date since you can get the same results with TRACK_TOKENS.

Incidentally, collecting begin/end tokens with this technique does have a disadvantage. Since the tokenizer creates a linked list of all the tokens, keeping a reference to one token prevents the others from being garbage collected. But in these halycon days of starter workstations shipping with a GB of RAM, maybe keeping those tokens in memory is a non-issue.

NODE_USES_PARSER

NODE_USES_PARSER is a boolean valued option that defaults to false. When NODE_USES_PARSER is enabled, JJTree generates a JavaCC grammar that creates nodes using an alternate constructor—one that accepts a parameter for the parser class.

Both constructors are generated for each node class regardless of the setting of NODE_USES_PARSER, so each node source file contains constructor declarations of this sort:

```
public ASTSomeNode(int id) {
  super(id);
}
public ASTSomeNode(MyParserName p, int id) {
  super(p, id);
}
```

The means that you can generate a grammar with NODE_USES_PARSER set to its default value of false, make various changes to your node classes, then go regenerate the grammar with NODE_USES_PARSER set to true without having to modify your node source code. Very handy!

NODE_USES_PARSER works in conjunction with STATIC; if NODE_USES_PARSER is true and STATIC is false, the calls to the node constructors pass in null. If both NODE_USES_PARSER and STATIC are true, the calls pass in a reference to the current parser object. This makes sense—if you're using a static parser there's no object reference to pass in to the node constructors; you can just reference the parser's static methods from your node classes.

OUTPUT_FILE

`OUTPUT_FILE` is a string valued option that defaults to the input file with the suffix replaced by `.jj`. In other words, here's the default behavior:

```
$ jjtree calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calculator.jj
$ ls *.jj
calculator.jj
```

Setting `OUTPUT_FILE` changes this name as you see fit:

```
$ jjtree -OUTPUT_FILE="calc_tree.jj" calculator.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file calculator.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "CalculatorTreeConstants.java" does not exist.  Will create one.
File "JJTCalculatorState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./calc_tree.jj
$ ls *.jj
calc_tree.jj
```

`OUTPUT_FILE` will accept a name with a directory prefix, e.g., `-OUTPUT_FILE="some_directory/a.jj"`, but the `some_directory` directory needs to exist when JJTree is run. JJTree won't create the directory for you, instead it will fail with an error: `Error setting input: Can't create output file "./some_directory/a.jj"`.

VISITOR

`VISITOR` is a boolean valued option that defaults to `false`. When `VISITOR` is set to `true`, JJTree makes several changes to the generated source code:

- A new source file, `XVisitor.java`, where `X` is the parser class name, is generated. `XVisitor` is an interface that defines callback methods for either `SimpleNode` or, if `MULTI` is set to `true`, each node type.
- The `Node` interface gets a new method, `Object jjtAccept(XVisitor, Object)`.
- The `SimpleNode` type gets two new methods: `Object jjtAccept(XVisitor, Object)` and `Object childrenAccept(XVisitor, Object)`. If the `MULTI` option is set to `true`, any node types are also generated with the `Object jjtAccept(XVisitor, Object)` method.

The net effect of these changes is to enable the AST to be traversed by client code that implements the `XVisitor` interface.

VISITOR_DATA_TYPE

Page 115 discussed the `visit` method signatures; each method accepts a visitor implementation and an `Object`. This allows for all sorts of flexibility since you can pass in anything as that second method argument. Sometimes, however, you know exactly what type the second argument will be in every case. When that's the situation you can use the `VISITOR_DATA_TYPE` option and specify that second argument's type. For example, you might set that option like this:

```
options {
  MULTI=true;
  VISITOR=true;
  VISITOR_DATA_TYPE="java.util.Set";
}
```

This would result in each of your `Node` implementations containing a method like this:

```
/** Accept the visitor. **/
public Object jjtAccept(CalculatorVisitor visitor, java.util.Set data) {
  return visitor.visit(this, data);
}
```

Note that you'll probably want to use a fully qualified type name here to avoid needing to insert an import statement into every generated `Node` implementation. The beauty of this option is that it can remove a lot of runtime casting as your code no longer needs to cast that second argument to the actual type that you're passing around.

VISITOR_EXCEPTION

`VISITOR_EXCEPTION` is a string valued option that defaults to an empty string. `VISITOR_EXCEPTION` works in conjunction with the `VISITOR` option; when the `VISITOR` option is `true`, the `VISITOR_EXCEPTION` option causes the `visit` method signature to change as follows:

```
// Default output
public Object visit(SimpleNode node, Object data);

// With VISITOR_EXCEPTION="MyException"
public Object visit(SimpleNode node, Object data) throws MyException;
```

If you set a value for `VISITOR_EXCEPTION` but don't set `VISITOR` to `true`, the `VISITOR_EXCEPTION` setting is ignored.

VISITOR_RETURN_TYPE

`VISITOR_RETURN_TYPE` is a string valued option that defaults to `Object`. Setting the `VISITOR_RETURN_TYPE` option changes the `visit` method signature as follows:

```
// Default output
public Object visit(SimpleNode node, Object data);

// With VISITOR_RETURN_TYPE="SomeClass"
public SomeClass visit(SimpleNode node, Object data);
```

As with `VISITOR_EXCEPTION`, if you set `VISITOR_RETURN_TYPE` but don't set `VISITOR` to `true`, the `VISITOR_RETURN_TYPE` setting will be ignored. You may want to specify

a fully qualified type name as well to avoid having to add `import` statements after generating the node classes.

Summary

In this chapter we've discussed using JJTree to generate an enhanced JavaCC grammar. This enhanced grammar builds an easily-traversable object tree to allow you to work with the data you're parsing without embedding lots of code in the grammar file itself. We also discussed an alternate tree builder, JTB. Whichever tree builder you choose, you'll find that it makes your grammars much more readable and gives you more control over the output.

Chapter 5. JJDoc

But I'll make it clearer. Perhaps it really is incomprehensible. --Fyodor Dostoevsky, "Crime and Punishment"

Documenting Your Grammar

A long JavaCC grammar can be an intimidating document. Although some IDEs support syntax highlighting the grammar, scrolling up and down through the web of nonterminals and tokens is not always easy. To address this problem, JavaCC includes a documentation utility, *JJDoc*. JJDoc creates text and HTML reports that provide an overview of a grammar's structure. In the case of the HTML report, the output also includes a nicely hyperlinked navigation path through the nonterminals.

JJDoc is a fairly limited utility, so this chapter is one of the shortest in this book. After reviewing a couple of examples of JJDoc's output, we'll go over the various JJDoc command line options.

Running JJDoc

The Java grammar that comes with JavaCC is a complicated one, so let's use that as a testbed. To generate HTML documentation with JJDoc, run the `jjdoc` utility with the grammar as a command line argument. The resulting HTML file will have the same name as the grammar, but the `.jj` will be replaced by `.html`:

Example 5.1. Running JJDoc

```
$ jjdoc Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in Java1.5.html
$ ls -l *.html
-rw-r--r--  1 tom  wheel  58111 Apr  2 00:40 Java1.5.html
```

A couple of notes about the generated documentation:

- The token productions appear first and are followed by the nonterminals. Having the token productions in the output is a recently added feature thanks to Tim Pizey.
- The nonterminal names in each production are hyperlinked; clicking on the link takes you to the production for that nonterminal.
- Running the Java grammar through JJDoc took about half a second on my moderately powered workstation and most of that time was undoubtedly spent waiting for the JVM to start up. So while there are no performance tweaks or options for JJDoc, you probably won't need any.

JJDoc uses the JavaCC parser to parse input data, so it does not work on a JJTree grammar, i.e., a `.jjt` file. To get JJDoc'd output for a JJTree file, first preprocess it with JJTree and then run JJDoc on the resulting `.jj` file. This means, of course, that all the JJTree-specific items, such as node descriptors, will be stripped out and won't appear in the generated documentation.

You can add a link to a Cascading Style Sheet (CSS) to the JJDoc generated HTML file. This lets you customize the properties of the HTML without having to hack the JJDoc code or postprocess the HTML. Here's an example of a very simple stylesheet; it sets the background color to light blue and italicizes the top line of the JJDoc HTML:

```
# examples/jjdoc/blue_background.css

1    BODY {background-color: lightblue}
2    H1 {font-style: italic}
```

Here's how you would use the `CSS` option:

Example 5.2. Running JJDoc with the CSS option

```
$ jjdoc -CSS=blue_background.css Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in Java1.5.html
```

If you look at the output of this run you'll be able to see that the `BNF for Java1.5.jj` text has been italicized.

There are many other possibilities for customizing the HTML using CSS—you could highlight alternate rows, add a background image, change the font size, and more. CSS is a vast subject in its own right, so for those who want more detail, I've listed an excellent resource in the bibliography.

Passing Javadoc Through a Grammar

On a more general documentation note, Javadoc comments that precede a production are passed on to the generated parser. So in the following production definition, the Javadoc describing the `Hello` production will appear in the parser:

```
/**
 * A <b>Hello</b> production simply consists of a HI token.
 */
void Hello() : {} {
  <HI>
}
```

This means that later, when someone generates Javadoc for the project that contains this parser, these documentation comments will be included in the Javadoc output.

JJDoc Options

Here's a summary of the options available for JJDoc. Note that all these options must all be passed in via the command line. Unlike JavaCC and JJTree, JJDoc options cannot be placed in an `options` header section of a grammar.

CSS

CSS is a string valued option that defaults to the empty string. If CSS is set, JJDoc will insert an HTML LINK element in which the href attribute points to the specified stylesheet. For example:

```
$ jjdoc -CSS=http://mysite.com/styles/javacc.css Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in Java1.5.html
$ grep LINK Java1.5.html
<LINK REL="stylesheet" type="text/css" href="http://mysite.com/styles/
javacc.css"/>
```

Note that JJDoc makes no attempt to validate the existence or format of the argument you pass to CSS option.

OUTPUT_FILE

OUTPUT_FILE is a string valued option which defaults to the grammar name with the .jj suffix replaced with either .html or .txt. It's .txt if the TEXT option (more on that option on page 144) is used. The file specification that you supply can include directory names.

Here's a run of JJDoc on the Java example grammar with the default output file name:

```
$ jjdoc Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in Java1.5.html
$ ls -l *.html
-rw-r--r--  1 tom  wheel  58111 Apr  2 00:40 Java1.5.html
```

And with a customized output file name:

```
$ jjdoc -OUTPUT_FILE="java_grammar.html" Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in java_grammar.html
$ ls -l *.html
-rw-r--r--  1 tom  wheel  58111 Apr  2 00:40 java_grammar.html
```

If JJDoc cannot open the output file, it will fall back to writing the output to standard out. This might happen if, for example, you specify an OUTPUT_FILE option with a file name that includes a nonexistent directory. Note that there's no equivalent of the parser's OUTPUT_DIRECTORY option; so creating a directory and then supplying that directory path and a filename to JJDoc is the way to get JJDoc to write its output to a particular directory.

ONE_TABLE

ONE_TABLE is a boolean valued option that defaults to true. If ONE_TABLE is set to false, JJDoc generates HTML output that differs from the default output:

- Each nonterminal is packaged in its own HTML `TABLE` element, so the output is about a third larger in size. Each of these tables is centered, giving the resulting HTML a "down the middle of the page" appearance.
- Each nonterminal is preceded by a header containing the nonterminal name, and nonterminals are separated by a horizontal rule.
- The token production output looks about the same whether `ONE_TABLE` is enabled or disabled.

I find that this option results in a rather cluttered page, but your mileage may vary.

TEXT

`TEXT` is a boolean valued option that defaults to `false`. If `TEXT` is set to `true`, JJDoc will generate the documentation in a text format that looks a bit like EBNF. Here are the first and last few lines of the Java grammar documentation generated by setting the `TEXT` option:

Example 5.3. Enabling the `TEXT` option

```
$ jjdoc -TEXT Java1.5.jj
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)
(type "jjdoc" with no arguments for help)
Reading from file Java1.5.jj . . .
Grammar documentation generated successfully in Java1.5.txt
$ head -5 Java1.5.txt
DOCUMENT START
TOKENS
/* WHITE SPACE */<DEFAULT> SKIP : {
" "
$ tail -5 Java1.5.txt
AnnotationTypeMemberDeclaration := Modifiers ( Type <IDENTIFIER> "("
")" ( DefaultValue )? ";" | ClassOrInterfaceDeclaration | EnumDeclaration |
AnnotationTypeDeclaration | FieldDeclaration )
  | ( ";" )
 DefaultValue := "default" MemberValue

DOCUMENT END
```

Notice that the `TEXT` output, like that created by the `ONE_TABLE` option, contains any comments that precede productions.

Summary

That's all for JJDoc. It's a straightforward little utility that can generate a bit of documentation and possibly make navigating your grammar a bit easier; it's definitely worth a run as part of your documentation process.

Chapter 6. JavaCC and Unicode

For concreteness, we suggest the use of standard 7-bit ASCII embedded in an 8 bit byte whose high order bit is always 0. --Vint Cerf, RFC 20: ASCII format for Network Interchange

What's Unicode?

This chapter focuses on parsing Unicode data with JavaCC. After a brief review of the history of character encodings, we'll look at various techniques that you can use within JavaCC to parse Unicode and other non-ASCII data. This includes exploring several of JavaCC's built-in options and utility classes that make most Unicode parsing jobs simple. By the end of this chapter, you should be able to write parsers that can easily handle Unicode data.

Early computers were expected to handle simple character data only: upper and lowercase letters, digits, and a few operators. This worked for a while and the American Standard Code for Information Interchange (ASCII) *character encoding* was (and still is) considered a decent solution even though it can represent only 128 characters. Since the high order bit is always 0, it uses only 7 of the 8 available bits in a byte; for example, the uppercase "a" `A` is encoded as `01000001`, or the decimal value 65. Various organizations noticed that if all 8 bits were used, an additional 128 slots would be available, and so each began to define a different standard for how those additional slots were used. Confusion has ensued as documents are exchanged between and usually corrupted by systems that use different encoding schemes.

A simple character encoding like ASCII can be used as long as the characters involved are, as the standard is named, "American." This encoding breaks down when we need to encode something more complicated. The French dish involving chopped liver can be written incorrectly as "pate", or correctly with diacritical marks as pâté. For processing such words, let alone languages like Chinese with thousands of different characters, we need a more comprehensive solution.

The best solution to this problem at present is *Unicode*. Unicode, to quote from the web site[1], "provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language." We usually don't need quite that much horsepower to use JavaCC effectively, but it's nice to work with a solution that provides room to grow.

Rather than mapping a character to a single byte as ASCII does, Unicode maps each character to a unique number (sometimes called a *code point*). These numbers are usually written as a `U+` followed by the hexadecimal value of the number. For example, the Latin uppercase A has a Unicode number of 65 (hex equivalent 41), so it's written as `U+0041`.

Once you know the code point for a particular character you can use a specific character encoding scheme, such as `UTF-8`, `UTF-16`, or `ISO-8859-1` to write that character

[1] http://unicode.org/

out in a well defined way. Once it's been written to a file or a socket, another program can read in the data and decode it using the same character encoding scheme.

To summarize, Unicode provides a unique number for each character, and a character encoding scheme provides a standard way to turn each number into a sequence of bytes. With that background in mind, let's see how JavaCC can use Unicode to properly handle data.

Escaping Unicode Characters

The JavaCC tokenizer converts bytes to characters and characters to tokens, so we'll be working primarily with that part of JavaCC in this chapter. Once we've got the input rolled up into tokens we can use the syntactic specification and the tree building parts of JavaCC as usual.

If you only want to use ASCII characters in your grammar, the standard way[2] to deal with Unicode characters in JavaCC (and in Java in general) is to encode those characters using an escape sequence. This escape sequence consists of the code point preceded by a `\u`. For example the code point `U+00FC`, also known as "the Latin small letter u with diaeresis", or more simply, ü, can be encoded as `\u00FC`. Here's a tokenizer that can handle a list of space-separated words that can consist of upper and lowercase ASCII characters as well as the code point `U+00FC`.

Example 6.1. Java Unicode Escaping

 # examples/unicode/words_encoded.jj

```
1    options {
2      BUILD_PARSER=false;
3      JAVA_UNICODE_ESCAPE=true;
4    }
5    PARSER_BEGIN(Words)
6    import java.io.*;
7    public class Words {}
8    PARSER_END(Words)
9    TOKEN_MGR_DECLS: {
10     public static void main(String[] args) throws Exception {
11       StringReader sr = new StringReader(args[0]);
12       JavaCharStream jcs = new JavaCharStream(sr);
13       WordsTokenManager mgr = new WordsTokenManager(jcs);
14       for (Token t = mgr.getNextToken(); t.kind != EOF;
15         t = mgr.getNextToken()) {
16           debugStream.println("Found token:" + t.image);
17       }
18     }
19   }
20   SKIP : {
21     " " | "\n"
22   }
23   TOKEN : {
24     <WORD : (["A"-"Z","a"-"z","\u00FC"])+>
25   }
```

Notice the new items in this grammar:

- We've added our escaped character definition (`\u00FO`) to the `WORD` token definition. The escape sequence makes for a long character definition, but it's still only a single character and so we can place it in a character class.

[2]Java Language Specification, 3rd Edition, §3.3

- In the `options` section, we've set the `JAVA_UNICODE_ESCAPE` option to `true`. This causes JavaCC to generate a new character stream reader, `JavaCharStream`, which can read in the escaped sequences and translate them into a Java `char` primitive.
- Since we're telling JavaCC to generate `JavaCharStream`, we're also using that class in our `main` method and passing it to the `TokenManager`.

Here's the heart of the `JavaCharStream` unescaping code:

```
buffer[bufpos] = c = (char)
  (hexval(c)         << 12 |
  hexval(ReadByte()) << 8 |
  hexval(ReadByte()) << 4 |
  hexval(ReadByte()));
```

Here's how it works. Suppose we have an escaped Unicode code point like `U+2158`[3], or, in the Java escaping parlance, `\u2158`. We need to decode this into a Java `char` primitive. The `hexval` method called in the code snippet above converts digits and upper and lowercase letters `a` to `f` to their decimal integer value. When the first character, `2`, is converted to a four byte `int` containing the value `2`, it looks like this in binary:

```
00000000 00000000 00000000 00000010
```

The integer value `2` is first left-shifted twelve times so that it occupies the upper four bits (also known as a *nibble*) of the third byte of the `int`. So now we have this:

```
00000000 00000000 00100000 00000000
```

The next byte, `1`, is read, converted to an `int`, and left shifted eight times so that it occupies the lower nibble of an integer's third byte. This new integer is combined with the first integer via a bitwise `OR` operation. So now we've decoded two of the four characters, both of which have been fit into the third byte of a single Java `int`. The `int` now looks like this:

```
00000000 00000000 00100001 00000000
```

The next byte, `5` is decoded into an integer and shifted only four bits to the left. When combined with the current integer via another bitwise OR we have this:

```
00000000 00000000 00100001 01010000
```

The last character, `8`, is decoded into an integer, but it's not shifted. Instead, it stays in the lowest nibble, the bitwise OR operation is repeated, and the last character is moved into its place:

```
00000000 00000000 00100001 01011000
```

This integer is then cast into a `char` primitive type. The cast operation truncates the top two bytes, which is fine since they were empty anyway. We're left with a Java `char` with the binary value `00100001 01011000`. This binary equivalent of the decimal value `8536` produces the appropriate character when printed in a Java program:

```
# examples/unicode/Char.java

1    public class Char {
2      public static void main(String[] args) {
3        System.out.println((char)8536);
4      }
5    }
$ javac Char.java
```

[3]The Unicode Character 'VULGAR FRACTION FOUR FIFTHS': ⅘

```
$ java -Dfile.encoding=UTF-8 Char
⅘
```

Notice that in this case I needed to specify a file encoding of UTF-8. That's because I'm writing this book on a Mac and the default file encoding is a single-byte encoding format, MacRoman, so it can't display this particular code point. Incidentally, MacRoman does indeed provide a character encoding for ü, but using a MacRoman-specific encoding wouldn't be very portable.

Let's circle back to our JavaCC grammar. We wanted to include escaped Unicode characters in our input, and now we see how JavaCC handles those. Here's a run with some sample data:

```
$ java -Dfile.encoding=UTF-8 WordsTokenManager "F\u00FCr Elise"
Found token:Für
Found token:Elise
```

Notice that when we printed out the ü it appeared in its unescaped form. Once the character was converted to Java's internal Unicode representation it stayed properly encoded, and was correctly displayed using the specified encoding, UTF-8.

Specifying characters with the Java escaping mechanism is probably the most straightforward way to use Unicode characters with JavaCC. But since it's not the only situation you'll encounter, let's explore some other techniques as well.

Unicode Characters and Encodings

In the real world it's unlikely that you'll be asked to write grammars that process data that uses the Java-specific Unicode escapes. Most of the time the input will simply contain Unicode characters, and your grammar will need to handle them. For example, rather than receiving `F\u00FCr Elise`, your tokenizer will instead see `Für Elise`.

Here's a starting solution to this problem that may be a bit surprising. Consider our earlier `words.jj` grammar. It had the `JAVA_UNICODE_ESCAPE` option off and used a `SimpleCharStream` to read in the characters. Let's modify this grammar very slightly by adding[4] a ü to the `WORD` token definition:

[4]You can quickly enter some sample Unicode data using **vim**. Open a document, hit **i** to get into insert mode, then hit **Ctrl-V**, then **u**, then the four characters of the hexadecimal value of the code point, e.g., **00FC**. Voilá! A Unicode character in a **vim** session.

Example 6.2. StringReader Simplifies Decoding

```
# examples/unicode/words_simple.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Words)
5    import java.io.*;
6    public class Words {}
7    PARSER_END(Words)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       Reader r = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(r);
12       WordsTokenManager mgr = new WordsTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found token:" + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " " | "\n"
21   }
22   TOKEN : {
23     <WORD : (["A"-"Z","a"-"z","ü"])+>
24   }
```

When we generate this tokenizer and hand it some Unicode input, that input is tokenized with no problems:

```
$ javacc -GRAMMAR_ENCODING=UTF-8 words_simple.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file words_simple.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java -Dfile.encoding=UTF-8 WordsTokenManager "Für Elise"
Found token:Für
Found token:Elise
```

Why did this succeed even though we used a `SimpleCharStream`? The reason it worked is related to the fact that we're passing in the data on the command line and processing it using a `StringReader`. Java reads in these command line arguments and decodes them into its internal Unicode character representation. So by the time the data gets to the `SimpleCharStream` via the `StringReader` it has already been decoded and is ready to pass off to the tokenizer. Furthermore, we used the `GRAMMAR_ENCODING` option to JavaCC with a value of `UTF-8`; this option tells JavaCC to use the specified encoding when reading the grammar file. If you're using a Linux workstation the `GRAMMAR_ENCODING` may not be necessary as many Linux-based operating systems default to a UTF-8 file encoding.

The same thing happens when we place the Unicode characters in a file and use a `FileReader` to pass characters to a `SimpleCharStream`. Here's the grammar:

Example 6.3. FileReader Decoding the Default Encoding

```
# examples/unicode/words_filereader.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Words)
5    import java.io.*;
6    public class Words {}
7    PARSER_END(Words)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws IOException {
10       Reader r = new FileReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(r);
12       WordsTokenManager mgr = new WordsTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != WordsConstants.EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found token:" + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " " | "\n"
21   }
22   TOKEN : {
23     <WORD : (["A"-"Z","a"-"z","ü"])+>
24   }
```

And the demonstration:

```
$ javacc -GRAMMAR_ENCODING=UTF-8 words_filereader.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file words_filereader.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java -Dfile.encoding=UTF-8 WordsTokenManager music_utf8.txt
Found token:Für
Found token:elise
```

Note the filename: `music_utf8.txt`. I chose that name because the file is encoded using UTF-8, which is not my workstation's default file encoding. Since it's not the default encoding, the `FileReader` can decode it only because first we told JavaCC that the grammar was UTF-8 encoded via the `GRAMMAR_ENCODING` option and next we passed in `-Dfile.encoding=UTF-8` to the JVM. If we convert[5] this file to an incompatible eight bit encoding, ISO 8859-1, and then run it again, the `FileReader` can't decode it:

```
$ javacc -GRAMMAR_ENCODING=UTF-8 words_filereader.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file words_filereader.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java WordsTokenManager music_iso8859_1.txt
Found token:F
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2.
Encountered: "\u00b8" (184), after : ""
```

[5]To do this with the Unix **`iconv`** utility, try this command: iconv -f utf8 -t iso_8859-1 -o music_iso8859_1.txt music_utf8.txt

```
 at WordsTokenManager.getNextToken(WordsTokenManager.java:279)
 at WordsTokenManager.main(WordsTokenManager.java:12)
```

To fix this, we need to use a `Reader` object that's configured for the input file's encoding. Java supplies an `InputStreamReader` that serves as a bridge between bytes and characters. We can create an `InputStreamReader` with the character encoding ISO 8859-1 and the `Reader` can wrap that to supply the `SimpleCharStream` with characters:

Example 6.4. FileReader Decoding Non-Default Encoding(ISO 8859-1)

```
# examples/unicode/words_filereader_iso8859_1.jj

1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Words)
5    import java.io.*;
6    public class Words {}
7    PARSER_END(Words)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws IOException {
10       InputStream is = new FileInputStream(args[0]);
11       Reader r = new InputStreamReader(is, "ISO_8859-1");
12       SimpleCharStream scs = new SimpleCharStream(r);
13       WordsTokenManager mgr = new WordsTokenManager(scs);
14       for (Token t = mgr.getNextToken(); t.kind != WordsConstants.EOF;
15         t = mgr.getNextToken()) {
16           System.out.println("Found token:" + t.image);
17       }
18     }
19   }
20   SKIP : {
21     " " | "\n"
22   }
23   TOKEN : {
24     <WORD : (["A"-"Z","a"-"z","ü"])+>
25   }
```

Here's our ISO 8859-1-aware grammar in action:

```
$ java -Dfile.encoding=UTF-8 WordsTokenManager music_iso8859_1.txt
Found token:Für
Found token:elise
```

A success! Of course, in order for this to work you need to know the encoding of the input data, and it probably won't be part of the filename. Such are the requirements of handling multiple encodings.

To combine a few scenarios, suppose you have a file containing both escaped and unescaped Unicode characters:

```
# examples/unicode/music_utf8_and_escaped.txt

1    Für elise
2    Die Himmel r\u00FChmen
```

You can combine `JAVA_UNICODE_ESCAPE` with a `Reader` to handle both cases. Here's a slightly modified version of `words_filereader.jj`; it still uses the UTF-8 `Reader` but now we're generating it with the `JAVA_UNICODE_ESCAPE` flag and thus it uses the `JavaCharStream`:

```
$ javacc -GRAMMAR_ENCODING=UTF-8 -JAVA_UNICODE_ESCAPE
words_filereader_escaped_and_utf8.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
```

```
Reading from file words_filereader_escaped_and_utf8.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "JavaCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java -Dfile.encoding=UTF-8 WordsTokenManager music_utf8_and_escaped.txt
Found token:Für
Found token:elise
Found token:Die
Found token:Himmel
Found token:rühmen
```

Another success! I wouldn't expect documents with mixed encodings such as this to surface frequently, but if they do, now you know how to handle them.

Skipping Unicode Characters

Most of the time you'll need your tokenizers to recognize Unicode input data—that is, you'll need to include acceptable Unicode characters and character ranges in your token definitions and then do something with those characters. In some applications, however, you may want to simply skip or discard Unicode characters. One way to do this is by defining a `SKIP` token that includes all non-ASCII characters.

Example 6.5. Skipping Unicode Characters with SKIP

```
# examples/unicode/skip_unicode.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Skipper)
5    import java.io.*;
6    public class Skipper {}
7    PARSER_END(Skipper)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       SkipperTokenManager mgr = new SkipperTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found token:" + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " " | "\n"
21   }
22   SKIP : {
23     <UNICODE : ["\u0080"-"\uFFFF"]>
24   }
25   TOKEN : {
26     <WORD : (["A"-"Z","a"-"z"])+>
27   }
```

Note that this character range starts at `0080` hexadecimal, which is 128 decimal. You may want to start this range at `007F` hexadecimal instead so that you also skip the last character in the ASCII character set—the delete character.

The problem with using `SKIP` in this way is that it results in tokens being broken around the Unicode characters. This is probably not what you want:

```
$ java SkipperTokenManager "Please play Für Elise"
Found token:Please
Found token:play
Found token:F
Found token:r
Found token:Elise
```

A better solution might be to replace the Unicode characters with some prearranged ASCII character. You can do this with a lexical action that operates on the token's image. In the example below we're using a regular expression that replaces all non-ASCII characters with a question mark:

Example 6.6. Normalizing Unicode Characters

```
# examples/unicode/replace_unicode.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Replacer)
5    import java.io.*;
6    public class Replacer {}
7    PARSER_END(Replacer)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) {
10       StringReader sr = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(sr);
12       ReplacerTokenManager mgr = new ReplacerTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found token:" + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " " | "\n"
21   }
22   TOKEN : {
23     <WORD : (["A"-"Z","a"-"z","\u0080"-"\uFFFF"])+>  {
24       matchedToken.image = matchedToken.image.replaceAll("[\u0080-\uFFFF]",
"?");
25     }
26   }
```

And now the ü is replaced, making the input uglier but retaining the original token structure:

```
$ java ReplacerTokenManager "Please play Für Elise"
Found token:Please
Found token:play
Found token:F?r
Found token:Elise
```

Another option would be to use a lexical action to collect all non-ASCII characters and store them in an error file. You could then check this error file and decide if the characters should be handled by your tokenizer or if they are data errors and someone else can be blamed. Either way, if your input data consistently uses just a few Unicode characters, adding them to your token manager's répertoire may be worth the effort.

Cleaner Error Messages

Suppose you've got a lexical specification that accepts some Unicode characters but not others. The default error message that the generated token manager produces con-

tains the escaped version of an unknown character. For example, in this run the lexical specification didn't recognize the Euro character (`U+20AC`, or €), and so it prints out the escaped code point in the error message:

```
$ java WordsTokenManager "That costs €5"
Found token:That
Found token:costs
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 12.
Encountered: "\u20ac" (8364), after : ""
 at WordsTokenManager.getNextToken(WordsTokenManager.java:269)
 at WordsTokenManager.main(WordsTokenManager.java:12)
```

You can make the error code a bit clearer by editing the `LexicalError` method in the generated `TokenMgrError` class. Here's the default version:

Example 6.7. Default Token Manager Error

 # examples/unicode/token_mgr_error_message/TokenMgrError.java.default (lines 93 to 111)

```
93     /**
94      * Returns a detailed message for the Error when it is thrown by the
95      * token manager to indicate a lexical error.
96      * Parameters :
97      *    EOFSeen     : indicates if EOF caused the lexical error
98      *    curLexState : lexical state in which this error occurred
99      *    errorLine   : line number when the error occurred
100     *    errorColumn : column number when the error occurred
101     *    errorAfter  : prefix that was seen before this error occurred
102     *    curchar     : the offending character
103     * Note: You can customize the lexical error message by modifying this
method.
104     */
105    protected static String LexicalError(boolean EOFSeen, int lexState, int
errorLine, int errorColumn, String errorAfter, char curChar) {
106      return("Lexical error at line " +
107            errorLine + ", column " +
108            errorColumn + ".  Encountered: " +
109            (EOFSeen ? "<EOF> " : ("\"" + addEscapes(String.valueOf(curChar))
+ "\"") + " (" + (int)curChar + "), ") +
110            "after : \"" + addEscapes(errorAfter) + "\"");
111    }
```

Note the last line of the comment above the `LexicalError` method definition; we can edit this method to customize the error message. In this case, we can modify line 109 so that it displays the Unicode character that was encountered by simply printing the current character. We can still display the escaped code point as well in case that's helpful. Here's the new version (without the comment header):

Example 6.8. Unicode Token Manager Error

```
# examples/unicode/token_mgr_error_message/TokenMgrError.java.new (lines 93 to
111)

93     /**
94      * Returns a detailed message for the Error when it is thrown by the
95      * token manager to indicate a lexical error.
96      * Parameters :
97      *    EOFSeen     : indicates if EOF caused the lexical error
98      *    curLexState : lexical state in which this error occurred
99      *    errorLine   : line number when the error occurred
100     *    errorColumn : column number when the error occurred
101     *    errorAfter  : prefix that was seen before this error occurred
102     *    curchar     : the offending character
103     * Note: You can customize the lexical error message by modifying this
method.
104     */
105    protected static String LexicalError(boolean EOFSeen, int lexState, int
errorLine, int errorColumn, String errorAfter, char curChar) {
106      return("Lexical error at line " +
107            errorLine + ", column " +
108            errorColumn + ".  Encountered: " +
109            (EOFSeen ? "<EOF> " : ( curChar + " " + ("(code point " +
addEscapes(String.valueOf(curChar)) + ") ") ) ) +
110            "after : \"" + addEscapes(errorAfter) + "\"");
111    }
```

And here's how the new error message appears:

```
$ java -Dfile.encoding=UTF-8 WordsTokenManager "That costs €5"
Found token:That
Found token:costs
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 12.
Encountered: € (code point \u20ac) after : ""
 at WordsTokenManager.getNextToken(WordsTokenManager.java:269)
 at WordsTokenManager.main(WordsTokenManager.java:12)
```

There's a lot of information in that error message; feel free to trim or expand it to fit the needs of your grammar.

Avoid UNICODE_INPUT

One of the options JavaCC accepts is `UNICODE_INPUT`. The current documentation suggests that this option causes JavaCC to "use an input stream object that reads Unicode files." The best advice I can give concerning this option is to avoid it and instead use the techniques described earlier in this chapter, i.e., use a `Reader` for translating bytes to characters and a `JAVA_UNICODE_ESCAPE` for translating escaped Unicode characters into Java's internal Unicode character representation.

Beyond the BMP

Unicode characters that fit into 16 bits represent what's called the *Basic Multilingual Plane*. The BMP contains 65536 characters, which seems like it should be plenty. However, as of Unicode 3.1 there are other *supplementary characters* that do not fit into the BMP. These require 21 bits of space, so they don't fit inside a 16 bit Java `char` primitive.

Java 1.5 has various API improvements to handle supplementary characters. For example, the Java 1.5 `Character` class now includes various methods that accept an `int` rather than a `char`. At 32 bits, an `int` can contain all the current Unicode charac-

ters. Similarly, the `StringBuilder` class contains an `appendCodePoint` method that accepts an `int` so it can handle supplementary characters.

JavaCC does not yet handle characters outside the BMP. There's been some discussion of this issue on the JavaCC mailing list, and there's an open bug[6], so the team is aware of the problem. Stay tuned for future developments. In the meantime, you can work around this limitation by specifying characters outside the BMP using a surrogate pair.

Summary

We've covered a variety of issues that you may encounter when processing Unicode data with JavaCC. Hopefully you now feel confident about writing grammars that can handle alternative character encodings and you may have even learned a thing or two about how Java itself processes Unicode data. May you no longer be baffled by "those funny characters"!

[6]https://javacc.dev.java.net/issues/show_bug.cgi?id=151

Chapter 7. Error Handling

...out of this nettle, danger, we pluck this flower, safety. --William Shakespeare, "King Henry the Fourth"

Parsing Problem Data

Most of the challenges we've looked at so far have involved describing the data that we want to parse. We've assumed that the data is well formed, meaning that it conforms to some predetermined structure and format. We've focused on ways to use JavaCC to formally specify that format and produce tokens, a parse tree, and an abstract syntax tree from that specification.

But in the real world, bad input data is common. It may be as simple as a typo entered by a user, or a whole new scenario—like the sudden inclusion of Unicode characters in the input source. Whatever the cause, we need to have a strategy for dealing with it. In this chapter we'll look at various mechanisms which JavaCC and JJTree provide for reporting and recovering from problems. We'll look at error handling techniques at the tokenizer, parser, and AST levels and discuss the issues involved with applying these techniques.

Tokenizing Errors

As we've seen, the tokenizing process transforms a stream of incoming characters into a series of tokens that can be passed off to the parser, so the lexical specification needs to be able to recognize valid sequences of characters and turn them into `Token` objects.

What happens when there's an unexpected series of characters? Let's experiment with the LOGO grammar example that appears in chapter 11 of this book. Here's the lexical specification for that tokenizer:

Example 7.1. The Logo Tokenizer

```
# examples/error_handling/original_logo_tokenizer/logo_tokenizer.jj (lines 19 to
26)

19   SKIP : {
20     " " | "\n" | "\r" | "\r\n"
21   }
22   TOKEN : {
23     <FORWARD : "FORWARD">
24     | <RIGHT : "RIGHT">
25     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
26   }
```

Here's an example of some valid input data:

```
# examples/error_handling/good_data.logo

1    FORWARD 20
2    RIGHT 120
3    FORWARD 20
```

And some bad data—note the mispelled command name `RIHGT`:

```
# examples/error_handling/bad_token.logo

1    FORWARD 20
2    RIHGT 120
3    FORWARD 20
```

First let's look at how the tokenizer handles errors out of the box, then take a look at ways we might do better.

Default Error Handling

When presented with a mispelled command (such as `RIHGT`), the default tokenizer raises a `TokenMgrError` and quits:

```
$ java LogoTokenManager bad_token.logo
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
Exception in thread "main" TokenMgrError: Lexical error at line 2, column 3.
Encountered: "H" (72), after : "RI"
 at LogoTokenManager.getNextToken(LogoTokenManager.java:407)
 at LogoTokenManager.main(LogoTokenManager.java:12)
```

JavaCC raises an exception that subclasses `java.lang.Error` rather than the usual `java.lang.Exception`. Other `Error` subclasses include such serious problems as `OutOfMemoryError`, so this puts it in a different light than the run of the mill exceptions such as `IOException` and `ClassNotFoundException`. This intentional choice on the part of JavaCC's authors conveys just what they intended—input that cannot be tokenized is a serious error.[1]

Although it's primitive, the technique of raising an exception and halting is not necessarily a bad idea. If the input data is completely garbled, the sooner your program discovers it the better. Besides, some sort of user intervention may well be necessary to get things back on track. Of course, you probably won't want to show the actual stacktrace; instead, you'll wrap the error up in some friendly way for display to the user. You may also want to store the input data somewhere so the user can repair it later. For example, if the program is processing a list of incoming files you may want to move the malformed file off into an "alibis" directory.

A Catch-All Token

One way to handle unexpected character sequences is to add a catch-all token definition to the end of lexical specification. This definition would match any character not matched by the previous token definitions. Here's our Logo lexical specification with a catch-all `ANYTHING` token definition:

[1]This decision has been debated over the years, and **`TokenMgrError`** may well turn into **`TokenMgrException`** at some point. But the point still stands; input that cannot be tokenized is considered a showstopper.

Example 7.2. Defining a Catch-All Token

```
# examples/error_handling/catch_all/logo_tokenizer.jj (lines 22 to 27)

22   TOKEN : {
23     <FORWARD : "FORWARD">
24     | <RIGHT : "RIGHT">
25     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
26     | <ANYTHING: ~[]>
27   }
```

With the `ANYTHING` token definition in place we don't get the `TokenMgrError` when we pass in bad input. Note that each character in what was intended to be a single `RIGHT` token is now stored in its own token:

```
$ java LogoTokenManager bad_token.logo
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
Found a <ANYTHING>: R
Found a <ANYTHING>: I
Found a <ANYTHING>: H
Found a <ANYTHING>: G
Found a <ANYTHING>: T
Found a <DIGITS>: 120
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
```

This technique solves the problem as far as the tokenizer itself is concerned—we can now pass in gibberish and the tokenizer will accept it without complaint. A parser using this tokenizer, however, will run into several problems. First, it would need to either skip the `ANYTHING` tokens or store them and report an error. To skip the `ANYTHING` tokens the parser would need to insert an expansion such as `(<ANYTHING>)*` at every point where the `ANYTHING` token could occur—in other words, almost everywhere. This would clutter the syntactic specification to the point of making it unusable. Second, in the case of a Logo parser, the syntactic specification will be expecting a structured stream of tokens that follow a pattern of "first a command, then some digits." If we pass in `RIHGT 120` we may be able to ignore the mispelled command, but we'll still be in a bad state since those digits don't constitute a valid command on their own.

So, while defining a catch-all token definition prevents an exception, it presents several other issues. We can do better!

Panic Mode

In our next technique for handling tokenizer errors we won't use a catch-all token. Instead, w'll catch the `TokenMgrError`, then scan forward in the input stream until we get to a place where the next token might begin. This is sometimes referred to as *panic mode* error recovery, and although this term usually refers to recovery in a parsing context it does describe what we're doing here in the tokenizing stage as well.

Here's the `TOKEN_MGR_DECLS` section for our tokenizer with this error handling mechanism in place:

Example 7.3. Skipping a Bad Token

```
# examples/error_handling/skip_chars/logo_tokenizer.jj (lines 7 to 41)
```

```
7     PARSER_END(Logo)
8     TOKEN_MGR_DECLS: {
9       public static void main(String[] args) throws Exception {
10        Reader r = new FileReader(args[0]);
11        SimpleCharStream scs = new SimpleCharStream(r);
12        LogoTokenManager mgr = new LogoTokenManager(scs);
13        while (true) {
14          try {
15            if (readAllTokens(mgr).kind == EOF) {
16              break;
17            }
18          } catch (TokenMgrError tme) {
19            debugStream.println("TokenMgrError: " + tme.getMessage());
20            recoverTo(' ');
21          }
22        }
23      }
24      private static Token readAllTokens(LogoTokenManager mgr) {
25        Token t;
26        for (t = mgr.getNextToken(); t.kind != EOF;
27          t = mgr.getNextToken()) {
28            String location = " starting at " + t.beginLine + "," +
t.beginColumn;
29            location += " and ending at " + t.endLine + "," + t.endColumn;
30            debugStream.println("Found a " + LogoConstants.tokenImage[t.kind] +
": " + t.image + location);
31        }
32        return t;
33      }
34      private static void recoverTo(char delimiter) throws IOException {
35        debugStream.println("Skipping ahead...");
36        int skipped = 0;
37        while (input_stream.readChar() != delimiter) {
38          skipped++;
39        }
40        debugStream.println("Skipped " + skipped + " characters");
41      }
```

The `main` method begins reading tokens as usual. If the tokenizer raises a `TokenMgrError`, control is passed over to the `recoverTo` method. `recoverTo` uses the `readChar` method to scan forward until it finds a space character. The intervening characters are discarded and the tokenizer is given control again at a place where it can carry on with decoding characters into tokens. Here's a sample run:

```
$ java LogoTokenManager bad_token.logo
Found a "FORWARD": FORWARD starting at 1,1 and ending at 1,7
Found a <DIGITS>: 20 starting at 1,9 and ending at 1,10
TokenMgrError: Lexical error at line 2, column 3.  Encountered: "H" (72), after :
 "RI"
Skipping ahead...
Skipped 3 characters
Found a <DIGITS>: 120 starting at 2,7 and ending at 2,9
Found a "FORWARD": FORWARD starting at 3,1 and ending at 3,7
Found a <DIGITS>: 20 starting at 3,9 and ending at 3,10
```

This is a useful technique because we aren't cluttering the token stream with `ANYTHING` tokens. But the syntactic problem remains; our parser will still choke when presented with a `DIGITS` token without a command name.

Suggesting a Fix

Another possibility is to record the bad characters and suggest a fix. This code is quite similar to the previous example except now we're saving the characters, checking them, and displaying an alternative:

Example 7.4. Skipping and Suggesting a Fix

```
# examples/error_handling/skip_and_retain/logo_tokenizer.jj (lines 7 to 44)

7     PARSER_END(Logo)
8     TOKEN_MGR_DECLS: {
9       public static void main(String[] args) throws Exception {
10        Reader r = new FileReader(args[0]);
11        SimpleCharStream scs = new SimpleCharStream(r);
12        LogoTokenManager mgr = new LogoTokenManager(scs);
13        while (true) {
14          try {
15            if (readAllTokens(mgr).kind == EOF) {
16              break;
17            }
18          } catch (TokenMgrError tme) {
19            debugStream.println("TokenMgrError: " + tme.getMessage());
20            String skipped = recoverTo(' ');
21            debugStream.println("Skipped bad input: " + skipped);
22            if (skipped.equals("RIHGT")) {
23              debugStream.println("Did you mean 'RIGHT'?");
24            }
25          }
26        }
27      }
28      private static Token readAllTokens(LogoTokenManager mgr) {
29        Token t = mgr.getNextToken();
30        for (; t.kind != EOF; t = mgr.getNextToken()) {
31          debugStream.println("Found a " + LogoConstants.tokenImage[t.kind]);
32
33        }
34        return t;
35      }
36      private static String recoverTo(char delimiter) throws IOException {
37        debugStream.println("Skipping ahead...");
38        String skipped = input_stream.GetImage();
39        char c;
40        while ((c = input_stream.readChar()) != delimiter) {
41          skipped += c;
42        }
43        return skipped;
44      }
```

And in action:

```
$ java LogoTokenManager bad_token.logo
Found a "FORWARD"
Found a <DIGITS>
TokenMgrError: Lexical error at line 2, column 3.  Encountered: "H" (72), after :
 "RI"
Skipping ahead...
Skipped bad input: RIHGT
Did you mean 'RIGHT'?
Found a <DIGITS>
Found a "FORWARD"
Found a <DIGITS>
```

These simple examples hide some of the difficulties with this technique. It's easy to call `recoverTo` with a space character since we know that's the next delimiter. If we had several types of delimiters, we would need to pass a collection of characters to `recoverTo` and check for each of them in turn. Another possibility would be to associ-

ate each lexical state (or even each token definition) with a specific delimiter. If something goes wrong in the middle of a quoted string, for example, you'd want to skip ahead to the next unescaped quote.

Repairing the Character Stream

So far we've seen how to skip errors; now we'll look at a way to look back at the erroneous characters in the input stream and actually repair the problem. We can get the current attempt at a match with the character stream class' `GetImage` method, then append the rest of the characters (up to the delimiter) to that image. Once we know that the input is something close to a valid token (like `RIHGT`), we can use that information to repair the character stream.

In the following implementation of this technique we're checking for only one possible mispelling, but you can imagine having a table of common typos or perhaps a soundex[2] algorithm for plugging in the proper token. The rest of the tokenizer is the same as in the previous examples, so we'll show only the `main` method here:

Example 7.5. Repairing a Mispelled Token

```
# examples/error_handling/skip_and_replace/logo_tokenizer.jj (lines 9 to 30)

9       public static void main(String[] args) throws Exception {
10        Reader r = new FileReader(args[0]);
11        SimpleCharStream scs = new SimpleCharStream(r);
12        LogoTokenManager mgr = new LogoTokenManager(scs);
13        while (true) {
14          try {
15            if (readAllTokens(mgr).kind == EOF) {
16              break;
17            }
18          } catch (TokenMgrError tme) {
19            debugStream.println("TokenMgrError: " + tme.getMessage());
20            String skipped = recoverTo(' ');
21            debugStream.println("Saw bad input: " + skipped);
22            if (skipped.equals("RIHGT")) {
23              mgr.input_stream.backup("RIHGT".length()+1);
24              mgr.input_stream.buffer[mgr.input_stream.bufpos+3] = 'G';
25              mgr.input_stream.buffer[mgr.input_stream.bufpos+4] = 'H';
26              debugStream.println("Repaired with RIGHT");
27            }
28          }
29        }
30      }
```

Here's a sample run:

```
$ java LogoTokenManager bad_token.logo
Found a "FORWARD": FORWARD starting at 1,1 and ending at 1,7
Found a <DIGITS>: 20 starting at 1,9 and ending at 1,10
TokenMgrError: Lexical error at line 2, column 3.  Encountered: "H" (72), after :
 "RI"
Skipping ahead...
Saw bad input: RIHGT
Repaired with RIGHT
Found a "RIGHT": RIGHT starting at 2,1 and ending at 2,5
Found a <DIGITS>: 120 starting at 2,7 and ending at 2,9
Found a "FORWARD": FORWARD starting at 3,1 and ending at 3,7
Found a <DIGITS>: 20 starting at 3,9 and ending at 3,10
```

Repairing the input stream in this way is a delicate job. A couple of points to consider when implementing this technique:

[2] http://en.wikipedia.org/wiki/Soundex

- We want to repair obvious errors only, that is, errors where it's clear that the user mistyped a command or a keyword. Attempting to repair more complex problems could lead to incorrect "fixes" which would probably be worse than the more conservative approach of halting the tokenizing process.
- In the above example we repair the input stream by overwriting only the two characters that need to be fixed to replace `RIHGT` with `RIGHT`. This could be refactored into a class that would overwrite the input stream buffer based on the token that needed to be inserted.
- An alternative to repairing the character stream is to recognize the error and then create a new token. To do this, we calculate the line and column numbers and image, populate the token with that data, and insert it into the token stream. This is a viable technique, but I prefer to set up the character stream and let the JavaCC-generated tokenizer do the work of creating the token.
- This example was also simplified since there were just a few letters transposed. If the error involved extra characters being read or consumned, repairing the input stream would be tricky since you might need to push some characters back into the input buffer. This can quickly get messy.
- The tokenizer error recovery code has to know what the tokenizer is expected to produce in order to repair the character stream properly. In other words, we were able to turn `RIHGT` into `RIGHT` only because we understood the expected structure of the token stream and knew that `RIHGT` was in a spot where a command should have been. Things get much more complicated when a programming language has variables; it may be impossible for the tokenizer to know whether `RIHGT` is a valid variable name declaration or an invalid command name. The tokenizer could communicate with the parser (via the `TOKEN_MANAGER_USES_PARSER` options discussed on page 64), but that introduces a host of new complications.

The advantages of this technique are obvious—any program using this tokenizer will now receive an uninterrupted stream of tokens in the expected sequence. It's a difficult technique to get right, though, and it gets more convoluted as the grammar gets more complicated.

To summarize, a tokenizing error is a bit of a disaster in the JavaCC world. Although the error handling techniques presented here work in some limited cases, the most pragmatic approach may be to accept the `TokenMgrError` and bail out when one occurs.

Parsing Errors

Let's assume that the tokenizing stage was completed successfully. Next up is the parsing stage, where the tokens are expected to be in a certain order to form a valid parse tree. Let's continue with the Logo grammar and see what's involved in recovering from an error at this stage.

Here's a set of input data for the Logo grammar. Line two has consecutive `RIGHT` tokens where only one movement token is allowed, so it can be tokenized but not parsed:

```
# examples/error_handling/original_logo_parser/bad_data.logo
```

```
1    FORWARD 20
2    RIGHT RIGHT 120
3    FORWARD 20
```

When we hand this to our Logo parser we get the expected `ParseException`:

```
$ java Logo bad_data.logo
Exception in thread "main" ParseException: Encountered " "RIGHT" "RIGHT "" at
line 2, column 7.
Was expecting:
    <DIGITS> ...

 at Logo.generateParseException(Logo.java:226)
 at Logo.jj_consume_token(Logo.java:164)
 at Logo.Turn(Logo.java:44)
 at Logo.Program(Logo.java:18)
 at Logo.main(Logo.java:7)
```

As with the tokenizer, halting the program at this point isn't necessarily a terrible solution. It lets us stop without doing anything dangerous, and the user can then fix the input data and rerun the program. But if there were several errors of this sort, it'd be nice to detect and report them all so the user could fix them in one fell swoop.

To report multiple errors, we need to be able to skip the current logical section of data that's causing problems and move on to the next block of data. The definition of a "logical section" of data depends on the domain in which your parser is operating. In our Logo example, each line of the input is a new section of data. In a C or Java program, each statement, or perhaps each function declaration, might be considered a separate data chunk. In a weather forecast, each day's data might be considered a separate section. We'll look at several techniques that work regardless of how the data is partitioned.

Shallow Error Recovery

One possibility for handling parsing errors is to use what JavaCC's documentation refers to as *shallow error recovery*. This technique works by adding a Javacode production that skips tokens until it gets to a place where parsing can resume. Here's a variant on our Logo grammar that applies this technique in the `Program` nonterminal:

Example 7.6. Shallow Error Recovery

 # examples/error_handling/logo_skip_command/logo_parser.jj (lines 14 to 43)

```
14   TOKEN : {
15     <FORWARD : "FORWARD">
16     | <RIGHT : "RIGHT">
17     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
18     | <EOL : "\n" | "\r" | "\r\n">
19   }
20   void Program() : {} {
21     (
22     Move() {System.out.println("Processed a Move");}
23     | LOOKAHEAD(2) Turn() {System.out.println("Processed a Turn");}
24     | <EOF> {return;}
25     | skipThisCommand()
26     )+
27   }
28   void Move() : {} {
29     <FORWARD> <DIGITS> <EOL>
30   }
31   void Turn() : {} {
32     <RIGHT> <DIGITS> <EOL>
33   }
34   JAVACODE
35   String skipThisCommand() {
36     ParseException pe = generateParseException();
37     System.out.println("Skipping line " + (pe.currentToken.beginLine + 1) + "
due to a parsing problem");
38     Token t;
39     do {
40       t = getNextToken();
41     } while (t.kind != EOL);
42     return t.image;
43   }
```

Here's a sample run:

```
$ java Logo bad_data.logo
Processed a Move
Skipping line 2 due to a parsing problem
Processed a Move
```

A couple of notes on this technique:

- We moved the end-of-line tokens out of the `SKIP` token and into their own token definition. That's so that we can use them as a "sentinel token", a marker for the place where skipping tokens should stop. We could use `DIGITS` as our sentinel, but that would fail if the erroneous input data contained two consecutive `DIGITS` tokens.

- The `skipThisCommand` production is the heart of this technique. It's the last alternative in `Program`'s main expansion, so all the other possibilities will be checked first. `skipThisCommand` calls `generateParseException` to get information on the current token location, although we have to add one to the line number because we're generating the exception before skipping forward. `skipThisCommand` then consumes tokens until it gets to the end of the erroneous line.

- There's a lookahead of 2 for the `Turn` nonterminal. This may seem unnecessary since the `Move` and `Turn` nonterminals can be disambiguated by their first token: it's either a `FORWARD` or a `RIGHT`. But our example input data consists of two consecutive `RIGHT` tokens. Without a lookahead directive, JavaCC will see the first `RIGHT` token and choose the `Turn` nonterminal. The next token that JavaCC sees will be

another `RIGHT`, which would result in an exception. JavaCC doesn't backtrack when it encounters unexpected tokens, so we have to look ahead instead.

Shallow error recovery works fairly well if the input data has delimiter tokens that you can skip to, or if you can modify the grammar to supply those (as we did here). But it also requires a somewhat shallow parse tree; deep trees will cause failures if you're unable to look ahead far enough to detect possible problems. JavaCC will make a choice, follow it, and then throw an exception that will bypass the `skipThisCommand` alternative. In the next section we'll discuss a possible solution.

Deep Error Recovery

A more advanced technique, *deep error recovery*, lets us handle errors that occur further down in the parse tree. Rather than supplying an error-skipping alternative, we catch any exceptions that occur and then recover at that point.

To implement deep error recovery, we add a `try`/`catch` block around the expansions that might raise an exception. In our Logo example, these are the `Move` and `Turn` nonterminals. If an exception occurs, we catch it and call `skipThisCommand` just as we did in the previous example. This solution also lets us remove the lookahead directive in the previous example; since we're going to catch the exception there's no need to use a lookahead directive to avoid it. Note that this `try`/`catch` (with an optional `finally`) block is one of the few Java statements that can be simply inserted into a JavaCC grammar.

Here's how the grammar looks with this error handling technique:

Example 7.7. Deep Error Recovery

 # examples/error_handling/logo_deep_skip/logo_parser.jj (lines 11 to 44)

```
11   SKIP : {
12     " "
13   }
14   TOKEN : {
15     <FORWARD : "FORWARD">
16     | <RIGHT : "RIGHT">
17     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
18     | <EOL : "\r" | "\n" | "\r\n">
19   }
20   void Program() : {} {
21     (
22       try {
23         Move() {System.out.println("Processed a Move()");}
24         | Turn() {System.out.println("Processed a Turn()");}
25       } catch (ParseException pe) {
26         System.out.println("Skipping line " + pe.currentToken.beginLine + "
due to a parsing problem");
27         skipThisCommand();
28       }
29     )+
30   }
31   void Move() : {} {
32     <FORWARD> <DIGITS> <EOL>
33   }
34   void Turn() : {} {
35     <RIGHT> <DIGITS> <EOL>
36   }
37   JAVACODE
38   String skipThisCommand() {
39     Token t;
40     do {
41       t = getNextToken();
42     } while (t.kind != EOL);
43     return t.image;
44   }
```

And here's a demonstration run:

```
$ java Logo bad_data.logo
Processed a Move()
Skipping line 2 due to a parsing problem
Processed a Move()
```

Notice that we left the new `EOL` token in place; `skipThisCommand` still needs to know where to stop skipping tokens. We could stop skipping tokens when we encounter either `FORWARD` or `RIGHT`, but that would leave us open to skipping past subsequent erroneous data such as a line containing only `120 120`. Our goal is to gather each faulty line of input data into its own error message, so we need to process each line separately.

This technique can handle bad input data like `RIGHT RIGHT 120` and `RIGHT 120 120` since both result in a `ParseException`. Even though JavaCC chooses the wrong branch, the fact that it fails with an exception if the input data is not well formed serves our purposes.

Tree Building Errors

Clearing the Stack

Although handling errors with JJTree is closely related to handling errors in the parsing stage, we need to do something different after catching the exception. We still need to skip tokens until we get to the next section of input, but we also need to clear the node stack.

Here's our Logo grammar with a few new additions. It's now a JJTree grammar, so the filename ends in `.jjt` and there's an `options` section with `VISITOR` and `MULTI` set to `true`. The most significant change is in the `Program` nonterminal in the `catch` block: we call `jjtree.popNode` (where `jjtree` is an instance of `JJTreeLogoState`) to clear the partially-constructed node off the stack.

Example 7.8. Recovering from an Error with JJTree

 # examples/error_handling/logo_tree_recover/logo_tree.jjt

```
1    options {
2      VISITOR=true;
3      MULTI=true;
4    }
5    PARSER_BEGIN(Logo)
6    import java.io.*;
7    public class Logo {
8      public static void main(String[] args) throws Exception {
9        Reader r = new FileReader(args[0]);
10       Logo parser = new Logo(r);
11       ASTProgram program = parser.Program();
12       program.dump("");
13     }
14   }
15   PARSER_END(Logo)
16   SKIP : {
17     " "
18   }
19   TOKEN : {
20     <FORWARD : "FORWARD">
21     | <RIGHT : "RIGHT">
22     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
23     | <EOL : "\r" | "\n" | "\r\n">
24   }
25   ASTProgram Program() : {} {
26     (
27       try {
28         Move()
29         | Turn()
30       } catch (ParseException pe) {
31         System.out.println("Skipping line " + pe.currentToken.beginLine + "
due to a parsing problem");
32         skipThisCommand();
33         jjtree.popNode();
34       }
35     )+
36     {return jjtThis;}
37   }
38   void Move() : {} {
39     <FORWARD> <DIGITS> <EOL>
40   }
41   void Turn() : {} {
42     <RIGHT> <DIGITS> <EOL>
43   }
44   JAVACODE
45   String skipThisCommand() #void {
46     Token t;
47     do {
48       t = getNextToken();
49     } while (t.kind != EOL);
50     return t.image;
51   }
```

Notice that we also mark `skipThisCommand` as `#void` to prevent JJTree from creating an AST node for it.

Here's some malformed data to pass to the grammar; the second line contains consecutive `RIGHT` tokens, so we'll want to discard it:

 # examples/error_handling/logo_tree_recover/bad_data.logo

```
1    FORWARD 20
2    RIGHT RIGHT 120
3    FORWARD 20
```

And here's a sample run of this grammar. As we expect, the resultant AST has only two `Move` nodes beneath `Program`—the malformed `Turn` has been discarded:

```
$ java Logo bad_data.logo
Skipping line 2 due to a parsing problem
Program
 Move
 Move
```

Summary

In this chapter you've learned how to write JavaCC tokenizers and grammars that handle errors in a variety of ways. While the default error handling mechanism of "report the error and exit" is good enough in many situations, you now have the tools to collect more than one error on each run and even recover from some types of errors.

Chapter 8. Case Study: The JavaCC Grammar

Brothers, let me take an example from everyday life. --Galatians 3:15

Examining the JavaCC Grammar

This chapter is devoted to analyzing the JavaCC grammar. First, let's recall how JavaCC works: you define a grammar and JavaCC generates a parser from it. But how does JavaCC parse the grammar file that you write? Enter the parser, generated by JavaCC, defined by—that's right—another JavaCC grammar. This chapter will help you understand the grammar that defines JavaCC grammars.

In the next pages I'll move through the JavaCC grammar, explaining the interesting bits. Sometimes I'll point you to more in-depth explanations in other chapters and sometimes I'll summarize a section—like the six hundred lines at the end of the grammar that defines a letter.

Why the JavaCC grammar for a case study, you ask? I picked it for several reasons:

- It's large and complex, so many of the more interesting facets of JavaCC are put to good use. Semantic lookahead, lexical states that use `MORE` and `SPECIAL_TOKEN`, and parser fields that track state changes all come into play.
- Studying the JavaCC grammar will help you better understand JavaCC. The grammar is the definitive source for all sorts of information about what JavaCC will handle; production kinds, keywords, and regular expression formats are all spelled out here.
- If you want to customize the JavaCC input format, you'd need to tweak this grammar. Want to add a new type of JJDoc option? You can do this if you're familiar with the grammar.
- A JavaCC grammar can contain Java code, so the JavaCC grammar contains most of the Java language grammar. You're reading this book, so I bet you're working with Java code! Seeing and understanding a Java syntactic specification is a great way to learn about Java corner cases.

The grammar that we'll study comes from the file `src/org/javacc/parser/JavaCC.jj` in the JavaCC source code repository. There's another grammar in `src/org/javacc/jjtree/JJTree.jjt`; this grammar is used to parse JJTree files. The two grammars are similar, although the JJTree grammar includes additional JJTree-specific constructs such as node descriptors. If you can read the JavaCC grammar, the JJTree grammar should be a lay-up.

This chapter has two main sections. First we discuss the lexical specification and then we examine the syntactic specification. Some parts of the lexical specification

(e.g., the handling of "greater than" tokens) are closely linked to the syntactic specification, but we'll point out that sort of thing as we go along.

Options and Headers

The JavaCC grammar starts with just two options:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 37 to 40)

37   options {
38     JAVA_UNICODE_ESCAPE = true;
39     STATIC=false;
40   }
```

`JAVA_UNICODE_ESCAPE` is set, which explains why you can have escaped Unicode characters embedded in a grammar. Also, the generated parser will be non-`STATIC`. This causes a performance hit when generating other parsers, but it's easier to manage.

Next comes the `PARSER_BEGIN/PARSER_END` section which contains the definition for the `JavaCCParser` class. There's no `main` method, so the parser can't be invoked directly from the command line. This is understandable; if you're processing JavaCC grammars, you can probably manage to knock out a supporting class or two. `JavaCCParser` extends the abstract class `JavaCCParserInternals`, which provides a number of static utility methods.

Next comes a short `TOKEN_MGR_DECLS` section. It contains a few fields and methods to track the line and column numbers of code inserted by JJTree or any other grammar preprocessing tool. We'll revisit those methods as we move further down through the grammar.

Lexical Specification

JavaCC Keywords

The actual lexical specification begins next. It's similar to the Java language lexical specification, except that it starts with a series of JavaCC-specific keyword token definitions, such as `MORE`, `SKIP`, and `LOOKAHEAD`. This list prevents you from using JavaCC keywords within a grammar. Suppose you tried to use `IGNORE_CASE` as a field name, as shown below:

Example 8.1. Using a JavaCC Keyword in a Grammar

```
# examples/case_study_javacc_grammar/reserved.jj

1    PARSER_BEGIN(Reserved)
2    public class Reserved {
3      private static final boolean IGNORE_CASE = true;
4    }
5    PARSER_END(Reserved)
6    TOKEN : {
7      <HELLO : "hello">
8    }
```

You'd get a parsing exception when JavaCC processed the grammar:

```
$ javacc reserved.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file reserved.jj . . .
org.javacc.parser.ParseException: Encountered " "IGNORE_CASE" "IGNORE_CASE "" at
line 3, column 32.
Was expecting one of:
    <IDENTIFIER> ...
    "[" ...

Detected 1 errors and 0 warnings.
```

There are just a dozen such keywords, so it's not a big limitation. Along similar lines, JavaCC used to generate parsers containing variable names that matched Java 1.5 keywords - e.g., `enum`. That particular bug has been fixed in a recent JavaCC release.

Generated Comments

Next comes a `SKIP` token definition which discards various bits of whitespace: tabs, spaces, carriage returns, and so forth. It includes `\f`, a rarely seen escape code for a "form feed.":

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 258 to 267)

```
258  SKIP :
259  {
260    " "
261  | "\t"
262  | "\n"
263  | "\r"
264  | "\f"
265  | "/*@egen*/" : AFTER_EGEN
266
267  }
```

This `SKIP` definition also includes `/*@egen*/`. `/*@egen*/`, short for "end generated content", is a comment used to mark the end of a range of code that has been inserted by another tool. Since JJTree is currently the only tool which is generating JavaCC grammars and using this feature, the corresponding "begin generated content" marker is `/*bgen(jjtree*/`. Here's a simple JJTree grammar:

 # examples/case_study_javacc_grammar/egen.jjt

```
1    PARSER_BEGIN(Egen)
2    public class Egen {}
3    PARSER_END(Egen)
4    TOKEN : { <A : "A">}
```

Running this grammar through JJTree adds several `bgen/egen` pairs with the JJTree-inserted code between them:

```
$ jjtree egen.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file egen.jjt . . .
File "EgenTreeConstants.java" does not exist.  Will create one.
File "JJTEgenState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./egen.jj
$ cat egen.jj
/*@bgen(jjtree) Generated By:JJTree: Do not edit this line. egen.jj */
/*@egen*/PARSER_BEGIN(Egen)
public class Egen/*@bgen(jjtree)*/implements EgenTreeConstants/*@egen*/ {/*@bgen
(jjtree)*/
  protected static JJTEgenState jjtree = new JJTEgenState();

/*@egen*/}
```

```
PARSER_END(Egen)
TOKEN : { <A : "A">}
```

Processing that grammar with JavaCC removes the `egen` comments, but not the `bgen` comments:

```
$ head -5 Egen.java
/* Generated By:JJTree&JavaCC: Do not edit this line. Egen.java */
public class Egen/*@bgen(jjtree)*/implements EgenTreeConstants, EgenConstants {/
*@bgen(jjtree)*/
  protected static JJTEgenState jjtree = new JJTEgenState();

  static private boolean jj_initialized_once = false;
```

When the tokenizer encounters a `/*bgen(jjtree*/`[1], it saves the current line and column by calling `saveBeginLineCol()`. This method, defined in `TOKEN_MGR_DECLS`, pushes the line and column information onto a stack[2]. When the tokenizer sees a `egen` comment, it pops the topmost item off the stack. This lets you use nested `bgen`/`egen` directives.

You could enhance this section of the grammar to handle content inserted by other tools. You'd need to add a new `MORE` token definition that included the name of the tool that was inserting the code. Alternatively, you could just reuse the current setup by just using `/*bgen(jjtree*/` to insert your code. As long as the inserted content ended with `/*egen*/`, it should Just Work.

Literals and Operators

Next come a series of token definitions for the Java keywords, literals, separators and operators. These definitions are (mostly) straightforward, though some of the regular expressions used in the literals are a bit complex. You may want to refer back to page 55 in the tokenizer chapter for a thorough explanation of one of the more complicated definitions, `STRING_LITERAL`.

One exception to the straight forward approach in this section is the handling of angle brackets. Before Java 1.5, angle brackets were defined as a couple of simple tokens for the comparison, shift, and "shift and assign" operators:

```
< GT: ">" >
< LT: "<" >
< LSHIFT: "<<"  >
< RSIGNEDSHIFT: ">>"  >
< RUNSIGNEDSHIFT: ">>>"  >
< LSHIFTASSIGN: "<<="  >
< RSIGNEDSHIFTASSIGN: ">>="  >
< RUNSIGNEDSHIFTASSIGN: ">>>=" >
```

With Java 1.5, you can use angle brackets to indicate a generic type; for example, the `Generic` class below declares an instance of `List` that can only contain `Sets` of `String` objects:

 # examples/case_study_javacc_grammar/Generic.java

```
1    import java.util.*;
2    public class Generic {
3      List<Set<String>> names = new ArrayList<Set<String>>();
4    }
```

[1] Why the missing close parenthesis? Because **`/*bgen(jjtree*/`** is the shortest match needed.
[2] It's actually implemented using an array and a **`depth`** integer that points to the end of the array.

To accommodate this, the JavaCC grammar tokenizes right shift operators as if they were simple greater than (>) tokens. You can see this happening in the lexical action for `RSIGNEDSHIFT`; the matched token is set to a `GT` by manually setting the `kind` and `image` fields, while the fact that it's really a right shift is preserved in the `Token.GTToken`'s `realKind` field. `Token.GTToken` is an odd bird: it's an inner class that subclasses its outer class.

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 506 to 512)

```
506  | < RSIGNEDSHIFT: ">>" >
507    {
508       matchedToken.kind = GT;
509       ((Token.GTToken)matchedToken).realKind = RSIGNEDSHIFT;
510       input_stream.backup(1);
511       matchedToken.image = ">";
512    }
```

Note the `input_stream.backup(1)` call. This causes the character stream to move back one character. So, a `RSIGNEDSHIFT` token is created when two `>` characters appear, but only one character is consumed. Later, in the syntactic specification, the `RSIGNEDSHIFT` nonterminal is defined:

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 2599 to 2606)

```
2599 void RSIGNEDSHIFT():
2600 {}
2601 {
2602   ( LOOKAHEAD({ getToken(1).kind == GT &&
2603                 ((Token.GTToken)getToken(1)).realKind == RSIGNEDSHIFT} )
2604   ">" ">"
2605   )
2606 }
```

This nonterminal does a semantic lookahead, detects that the token's `realkind` field is set to `RSIGNEDSHIFT`, and then consumes the next two `>` tokens. So we get the proper results—a `RSIGNEDSHIFT` nonterminal appears in the parse tree. Of course, anyone using just the tokenizer for this grammar needs to be aware of the way it's handling angle brackets. If they don't, they won't know to check the `realKind` field and will get some unexpected token sequences.

Identifiers

The rest of the lexical specification appears at the bottom of the grammar file and defines the valid identifiers. These are handled using two private token definitions: `LETTER` and `PART_LETTER`. `LETTER` handles all the characters which the Java Language Specification defines as being a valid "start of identifier" character, and `PART_LETTER` handles all the characters which are valid "part of identifier" characters. These definitions consist (mostly) of ranges of Unicode characters such as:

```
"\u0388"-"\u038a"
```

The above character range indicates that Έ, Ή, and K are valid characters for starting an identifier. The complete set of character ranges are based on the corresponding code in the `java.lang.Character` class, namely, the methods `isJavaIdentifierStart` and `isJavaIdentifierPart`.

Syntactic Specification

The JavaCC syntactic specification consists of two parts: the JavaCC-specific productions, like `bnf_production` and `character_descriptor`, and the Java grammar productions, like `ClassOrInterfaceDeclaration` and `EnumDeclaration`. We'll look at both parts and, rather than examine every production, we'll hit the high points.

The JavaCC Grammar

`javacc_input` is the start symbol for the JavaCC grammar and consists of three main sections: the options section, the parser class definition, and the productions for the lexical and syntactic specifications.

The options section is defined by the `javacc_options` production. It's a "one or more" expansion of the `option_binding` nonterminal, which explains why you can't have an empty `options` section (i.e., `options {}`). The production ends with a call to `Options.normalize()`. The `normalize` method does housekeeping chores, such as issuing a warning when command line options are overriding grammar options.

One clever bit about the `option_binding` production: it uses some of the nonterminals in the Java grammar to parse out option values. Here's the expansion that handles boolean option values; it uses the `Boolean` nonterminal to verify the input and then uses `getToken(0)` to get the actual value:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 579 to 582)

579      bool_val = BooleanLiteral()
580    {
581      Options.setInputFileOption(t, getToken(0), option_name, new Boolean
(bool_val));
582    }
```

Next comes the parser class definition. This definition is, more or less, the start symbol for the Java grammar wrapped in `PARSER_BEGIN/PARSER_END` delimiters:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 521 to 548)

521  void javacc_input() :
522   {
523     String id1, id2;
524     initialize();
525   }
526  {
527    javacc_options()
528    "PARSER_BEGIN" "(" id1=identifier()
529   {
530     addcuname(id1);
531   }
532                    ")"
533   {
534     processing_cu = true;
535     parser_class_name = id1;
536   }
537    CompilationUnit()
538   {
539     processing_cu = false;
540   }
541    "PARSER_END" "(" id2=identifier()
542   {
543     compare(getToken(0), id1, id2);
544   }
```

```
545                 ")"
546    ( production() )+
547    <EOF>
548  }
```

Note that it ends with a call to a `compare` method (which is defined in a supporting class, `JavaCCParserInternals`) to ensure the class name in `PARSER_END` matches the name in `PARSER_BEGIN`. If they don't match, JavaCC exits without building a parse tree.

Next come the four production definitions: `token_manager_decls`, `regular_expr_production`, `javacode_production`, and `bnf_production`. The grammar doesn't mandate an order for these types of productions; they're listed as alternatives. As a consequence, you can put a `TOKEN_MANAGER_DECLS` section in the midst of your nonterminal definitions. That's probably not a great idea, though, since it will obfuscate your grammar.

The `token_manager_decls` production, which you may recognize by its start token, `TOKEN_MGR_DECLS`, is a bit different than the other three productions. It's more like the `PARSER_BEGIN/PARSER_END` section in that, with a nod to the movie "Highlander", there can be only one. This constraint isn't enforced by the grammar, so two `token_manager_decls` productions could be parsed. But after each `token_mgr_decls` production is parsed, a syntactic action calls `add_token_manager_decls` and this method throws an exception if a `TOKEN_MGR_DECLS` section has already been encountered. So two `token_manager_decls` sections can be parsed but the process will halt after the second one is parsed. Here's `token_manager_decls`:

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 788 to 799)

```
788  void token_manager_decls() :
789   {
790     java.util.List decls = new java.util.ArrayList();
791     Token t;
792   }
793  {
794    t="TOKEN_MGR_DECLS" ":"
795    ClassOrInterfaceBody(false, decls)
796   {
797     add_token_manager_decls(t, decls);
798   }
799  }
```

The three other types of productions (`javacode_production`, `regular_expr_production`, and `bnf_production`) are the heart of the grammar. We'll look closely at each of these.

The Javacode Production

Recall from chapter 2 the general structure of a Javacode production. It starts with a `JAVACODE` token, but then looks more or less like a Java method. There's an optional `AccessModifier`; this is typically not included since most of the time these are called from within a parser. Next comes a `ResultType`, then the name (an `identifier`), followed by parameters, an optional `throws` clause and, finally, the body of the production, which is a `Block` containing Java code. The actual Java code is optional, so this empty javacode production is valid:

```
JAVACODE void bar() { }
```

Defining an empty Javacode production could be useful if you plan on subclassing the generated parser and overriding this production.

The `AccessModifier` production isn't used to return a result. Instead, a `NormalProduction` is passed to the method. Once the access modifier is determined, the `accessMod` field on the `NormalProduction` object is set to that value. This technique of using method parameters for communicating between productions was discussed in more detail on page 75.

The Regular Expression Production

Next we see the `regular_expr_production`, which is used to define elements (tokens, `SKIP`/`MORE`/`SPECIAL_TOKEN`) of the lexical specification. This production begins with a couple of lines to handle the various ways to specify the lexical states to which a token definition applies: all states (`<*>`), one state (`state_a`), or multiple states (`state_a, state_b`). These are collected in a `List`, copied to an array and then attached to the `TokenProduction` object which encapsulates the entire production.

There can be a series of `regexpr_spec` productions separated by the pipe symbol (`|`) to combine multiple token definitions in a single production:

```
TOKEN : {
  <A : "A">
| <B : "B">
}
```

Each `regexpr_spec` production is composed of a `regular_expression` followed by optional lexical actions and lexical state change directives. `regexpr_spec` also enforces some limits on private regular expressions; namely, they can't have lexical actions or lexical state changes.

Each individual token definition is handled by the `regular_expression` nonterminal. A token definition can consist of just a string literal, so one of the alternatives calls the `StringLiteral` nonterminal which returns a `String`. This lets us handle simple cases like:

```
TOKEN : {
   "hi" | "there"
}
```

Another offbeat case handled by `regular_expression` is a free-standing regular expression:

```
TOKEN : {
  <HELLO>
}
```

These are useful when they appear in a nonterminal, e.g.:

```
void A() : {} {
  <HELLO>
}
```

Although it's accepted by the grammar, placing one of these in a token definition doesn't make sense. It results in a warning from the tokenizer, and the definition is discarded before the tokenizer is produced.

After a few more levels of organizational productions, we get down to the `character_list` nonterminal. A `character_list` contains `character_descriptor` objects, which are either single characters or character ranges. At this point, we're at the end of the regular expression parse tree, and only terminals remain.

To make the structure more concrete, consider a regular expression to define one or more digits: `(["0"-"9"])+`. Here are the productions that are used to parse out this expression and the tokens that each production consumns:

```
complex_regular_expression_unit "("
 complex_regular_expression_choices
   complex_regular_expression
     complex_regular_expression_unit
       character_list "["
         character_descriptor
           StringLiteral "0"
           StringLiteral "9"
         character_descriptor
       character_list "]"
     complex_regular_expression_unit
   complex_regular_expression
 complex_regular_expression_choices
complex_regular_expression_unit ")" "+"
```

The `character_descriptor` production also consumes the "-" token that separates the characters in the range definition. Note that although some nonterminals, such as `complex_regular_expression_unit`, don't consume anything in this example. But they would come into play in more complicated cases.

The BNF Production

The last production type is the `bnf_production`, which is used to define the syntactic specification elements. A `bnf_production` looks much like a Javacode production, but without the `JAVACODE` keyword; it has an `AccessModifer`, a `ResultType`, an optional `throws` clause, and so forth. Here's the production:

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 664 to 709)

```
664  void bnf_production() :
665   {
666     BNFProduction p = new BNFProduction();
667     Container c = new Container();
668     Token t = getToken(1);
669     p.setFirstToken(t);
670     java.util.List excName;
671     String lhs;
672     p.setThrowsList(new java.util.ArrayList());
673     p.setLine(t.beginLine);
674     p.setColumn(t.beginColumn);
675     jumpPatched = false;
676   }
677  {
678    AccessModifier(p)
679    ResultType(p.getReturnTypeTokens())
680    lhs=identifier()  { p.setLhs(lhs); }
681    FormalParameters(p.getParameterListTokens())
682    [ "throws"
683   {
684     excName = new ArrayList();
685   }
686      Name(excName)
687   {
688     p.getThrowsList().add(excName);
689   }
690      (
```

```
691   {
692     excName = new ArrayList();
693   }
694         "," Name(excName)
695   {
696     p.getThrowsList().add(excName);
697   }
698       )*
699    ]
700    ":"
701    Block(p.getDeclarationTokens())
702    "{" expansion_choices(c) t="}"
703   {
704     p.setLastToken(t);
705     p.setJumpPatched(jumpPatched);
706     production_addexpansion(p, (Expansion)(c.member));
707     addproduction(p);
708   }
709  }
```

However, rather than containing a `Block` as its central element like a Javacode production does, a `bnf_production` contains an `expansion_choices` nonterminal. Like `regular_expression`, `expansion_choices` contains a flurry of related nonterminals that can be (and usually are) nested recursively. However, the basic unit is an `expansion_unit` nonterminal. An `expansion_unit` can contain other `expansion_choices`.

For example, the `Location` nonterminal below has a production body that contains two `expansion_unit`s, `State` and `[Zip() | LongZip()]`:

```
void Location() {} {
  State() [Zip() | LongZip()]
}
```

The latter expansion unit contains another `expansion_choices`, each of which consists of another `expansion_unit`.

Expansion and Semantic Lookahead

The `expansion` production uses an interesting technique to read in all the elements of an `expansion_unit`. Once you've started an `expansion_unit`, almost any token with the exception of a "closing token" type (such as a right parenthesis or a right curly brace) can be part of it. `expansion` enables this by using zero-length local lookahead to force a call to the subsequent semantic lookahead production which checks for a closing token.

Here's the lookahead directive; note the semantic lookahead call to `notTailOfExpansionUnit`, which was defined earlier in the `PARSER_BEGIN/PARSER_END` section:

 # examples/case_study_javacc_grammar/JavaCC.jj (line 924)

```
924    ( LOOKAHEAD(0, { notTailOfExpansionUnit() } )
```

This lookahead directive is contained in a "one or more" expansion, so the `expansion` production continues to consume tokens until the semantic lookahead fails, that is, until `notTailOfExpansionUnit` encounters a closing token and returns `false`.

For completeness, here's the `notTailOfExpansionUnit` method that's called by the semantic lookahead:

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 98 to 103)

```
98     private boolean notTailOfExpansionUnit() {
99       Token t;
100      t = getToken(1);
101      if (t.kind == BIT_OR || t.kind == COMMA || t.kind == RPAREN || t.kind ==
 RBRACE || t.kind == RBRACKET) return false;
102      return true;
103    }
```

We could probably get a tiny performance gain by either placing these options in a genericized `Set` or doing some analysis to see which closing token is likely to be encountered most often and placing the check for that token first.

The Java Grammar

A JavaCC grammar contains Java code in various places: in the `PARSER_BEGIN/PARSER_END` blocks, in semantic lookahead directives, in javacode productions, in lexical actions, and so forth. Thus, the JavaCC grammar needs to contain a Java grammar so that it can parse and at least partially validate that code. As with the previous sections, we'll look at some of the interesting bits in this part of the grammar.

Annotation Syntactic Lookahead

The start symbol for the Java grammar is the `CompilationUnit`. This is only called from `PARSER_BEGIN/PARSER_END`; all the other Java grammar usages employ other nonterminals as the start symbol. A `CompilationUnit` consists of a series of optional elements: annotations, a `package` statement, and `import` statements, and then a type declaration, such as a class or an interface.

The annotation check has to use local syntactic lookahead to determine whether any leading annotations are annotating a package statement or whether they instead apply to the first type declaration:

 # examples/case_study_javacc_grammar/JavaCC.jj (lines 1449 to 1464)

```
1449 void CompilationUnit() :
1450 /*
1451  * The <EOF> is deleted since the compilation unit is embedded
1452  * within grammar code.
1453  */
1454  {
1455    set_initial_cu_token(getToken(1));
1456  }
1457 {
1458   [ LOOKAHEAD( ( Annotation() )* "package" ) PackageDeclaration() ]
1459   ( ImportDeclaration() )*
1460   ( TypeDeclaration() )*
1461  {
1462    insertionpointerrors(getToken(1));
1463  }
1464 }
```

In practice, not many `package` statements have annotations[3], and those that do probably have small ones. But if there were a large annotation, that could mean a lot of lookahead, so it might be worth defining a lookahead-only production to avoid scanning forward through the entire annotation.

[3]Per JLS 7.4.1.1, "...at most one annotated package declaration is permitted for a given package."

Modifiers

A Java type declaration can have modifiers such as `public`, `static`, `final`, and so forth. The `Modifiers` nonterminal handles these in an efficient way; it records the occurrence of each modifier using a bitwise OR operation on an `int` and a set of predefined constants. For example, when it encounters the `public` modifier, it sets the appropriate `modifiers` bit as follows:

```
"public" { modifiers |= ModifierSet.PUBLIC; }
```

The modifiers are collected in the `int` and that value is returned to the caller. The `Modifiers` production does this collecting within a "zero or more" expansion, so a modifier sequence like `public public static` will be accepted by the parser. This is invalid Java code, but it will be caught later by the Java compiler so it's not crucial here. If you wanted to fix this, you could augment the syntactic action by having it raise an exception if it encountered a modifier whose bit was already set.

ExtendsList and Syntactic Actions

Moving down the grammar a bit, there's the `ExtendsList` nonterminal, which parses the list of types which a class or interface extends. A syntactic action ensures that a Java class declaration can only extend one type:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 1574 to 1585)

1574 void ExtendsList(boolean isInterface):
1575 {
1576    boolean extendsMoreThanOne = false;
1577 }
1578 {
1579    "extends" ClassOrInterfaceType()
1580    ( "," ClassOrInterfaceType() { extendsMoreThanOne = true; } )*
1581    {
1582       if (extendsMoreThanOne && !isInterface)
1583          throw new ParseException("A class cannot extend more than one other
 class");
1584    }
1585 }
```

There's a similar check in `ImplementsList` to ensure that an interface type declaration doesn't include an `implements` clause.

Token Gathering

Various nonterminals—`Block`, `Name`, `ResultType`—use a simple technique for gathering up all tokens that appear within their scope. They declare a local variable in the nonterminal's initialization block and set that to the first token in the nonterminal:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 1890 to 1894)

1890  {
1891    Token first = getToken(1);
1892    if (tokens == null)
1893      tokens = new ArrayList();
1894  }
```

Then, in the last syntactic action, all the tokens between the first and the last terminal are swept up into a `List` that is either passed in as a parameter or initialized on the spot:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 1901 to 1909)

1901  {
1902    Token last = getToken(0);
1903    Token t = first;
1904    while (true) {
1905      tokens.add(t);
1906      if (t == last) break;
1907      t = t.next;
1908    }
1909  }
```

The caller can then do whatever it needs with this collection of tokens. This grammar does different things with it; in some cases, such as in `PackageDeclaration`, the `List` gets discarded. In other cases, such as in a Javacode production `throws` clause, it's stored in another `List` that contains the names of all the types that can be thrown from that production. Incidentally, it's entirely possible that these syntactic actions may be removed at some point in favor of the built-in `TRACK_TOKENS` option.

Java Statements

The `Statement` nonterminal serves to group all the different types of Java statements—`break`, `continue`, and so forth:

```
void Statement(): {} {
  LOOKAHEAD(2)
  LabeledStatement()
|
  AssertStatement()
|
  Block(null)
  [ .... and several more ... ]
```

The `ForStatement` nonterminal is included in the above list; it handles both the old style `for` loop syntax and the Java 1.5 "for in" statement syntax. Here's an example of both:

```
# examples/case_study_javacc_grammar/ForExample.java

1    public class ForExample {
2      void for_old(String[] strings) {
3        for (int i = 0; i < strings.length; i++) {
4          // do something with strings[i]
5        }
6      }
7      void for_in(String[] strings) {
8        for (String string : strings) {
9          // do something with string
10       }
11     }
12   }
```

The grammar decides style is being used via syntactic lookahead. If there's a type declaration followed by an identifier and a colon, it's the "for in" style loop:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 2470 to 2483)

2470 void ForStatement():
2471 {}
2472 {
2473   "for" "("
2474
2475   (
2476       LOOKAHEAD(Modifiers() Type() <IDENTIFIER> ":")
2477       Modifiers() Type() <IDENTIFIER> ":" Expression(null)
2478     |
```

```
2479       [ ForInit() ] ";" [ Expression(null) ] ";" [ ForUpdate() ]
2480    )
2481
2482    ")" Statement()
2483 }
```

A possible optimization would take advantage of the fact that the "for in" loop style cannot use an existing variable as the loop control variable. The following is not valid Java code:

```
void test(String[] strings)
  String string;
  for (string : strings) {
    // do something
  }
}
```

With that language limitation in mind, we could optimize the production slightly by doing a lookahead:

```
LOOKAHEAD(<IDENTIFIER> "=")
[ ForInit() ] ";" [ Expression() ] ";" [ ForUpdate() ]
```

This is a micro-optimization that probably wouldn't make much difference since `for` loops with existing variables in the `ForInit` block may not be common enough to make this worthwhile. Nevertheless, this technique of exploiting language rules can be valuable when you're working on your own grammars. If there's a restriction in the input data specification, you can usually write your grammar to take advantage of it.

Java Expressions

Java expressions can range from simple ones like `2+2` to complex combinations with different operators and parentheses, like `((8<<1)*3)/6*2`. They can also include identifiers, generic types, array dereferences, and more. To give you a good feel for the general principles, we'll look at the basic Java arithmetic expression grammar.

No surprise, the start symbol is the `Expression` nonterminal. `Expression` begins with a comment explaining that it was written the way it was to improve performance. In other words, you can write a nonsensical expression like `2=1` and `Expression` and its children will happily generate a parse tree for it. Given the usage model of JavaCC, this isn't a critical failure. If someone is generating a parser, chances are they'll compile it shortly afterwards and the Java compiler will catch this faulty expression.

Let's trace through a simple expression: `2+2`. Here's the heart of the `Expression` nonterminal:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 1965 to 1969)

1965   ConditionalExpression()
1966   [
1967  LOOKAHEAD(2)
1968     AssignmentOperator() Expression(null)
1969   ]
```

The first token that `Expression` sees is the literal `2`. Since `ConditionalExpression` is the first expansion in the `Expression` production, JavaCC passes control down to `ConditionalExpression`. Here's `ConditionalExpression`:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 1988 to 1992)

1988 void ConditionalExpression():
1989 {}
1990 {
1991   ConditionalOrExpression() [ "?" Expression(null) ":" Expression(null) ]
1992 }
```

It's quite similar to the `Expression` production. There's an expansion for another expression type, `ConditionalOrExpression`, followed by an optional expansion that defines the current nonterminal. In this case, a conditional expression, also known as a ternary expression, consists of a `?` followed by an `Expression` followed by `:` and another `Expression`.

This pattern continues, as `ConditionalOrExpression`'s first expansion is `ConditionalAndExpression`, which in turn has `InclusiveOrExpression` as its first expansion. If we continue down this chain of nonterminals, we pass through the entire list of Java operators: `EqualityExpression`, `ShiftExpression`, `PostfixExpression`, and so forth, until we finally get down to the `Literal` nonterminal. `Literal` is defined as a series of token alternatives including `<INTEGER_LITERAL>`, the one we're looking for. So that `INTEGER_LITERAL` token is consumed.

Now that the token has been consumed, control passes back up from `Literal` to the parent nonterminal, which in this case is `PrimaryPrefix`. `PrimaryPrefix` can't consume the next token (the `+`), so control continues to rise back up the call stack until it hits `AdditiveExpression`. Here's the `AdditiveExpression` production:

```
# examples/case_study_javacc_grammar/JavaCC.jj (lines 2048 to 2052)

2048 void AdditiveExpression():
2049 {}
2050 {
2051   MultiplicativeExpression() ( ( "+" | "-" ) MultiplicativeExpression() )*
2052 }
```

Since `AdditiveExpression`'s second expansion provides a match for either the plus or the minus sign, the `+` token is consumed and control passes back down to `MultiplicativeExpression`. `MultiplicativeExpression` is structured the same way, so control once again passes step by step all the way down to `Literal`, and the second `2` is consumed.

This descending nonterminal order is how the grammar implements the Java operator precedence. The operators with lower precedence, like conditional "or" and equality, are at the top, and the higher precedence operators, like postfix, are at the bottom. Since control passes all the way down to the bottom of the chain, higher precedence operators get the first opportunity to consume tokens.

One way to visualize this is to build JavaCC with `DEBUG_PARSER` enabled and then feed it our sample expression. The output starts with a trip down the nonterminal chain, ending with a consumed `Literal`:

```
Call:   Expression
 Call:   ConditionalExpression
  [ ... lots of nonterminals skipped ... ]
   Call:   AdditiveExpression
    Call:   MultiplicativeExpression
     [ ... ]
      Call:   Literal
```

```
      Consumed token: <INTEGER_LITERAL>: "2">
     Return:   Literal
```

Next we go up a few levels until the `AdditiveExpression` consumes the `+`:

```
    [ ... ]
   Return:   MultiplicativeExpression
   Consumed token: <"+">
```

Then back down again to consume the second literal:

```
   Call:   MultiplicativeExpression
    [ ... ]
     Call:   Literal
      Consumed token: <INTEGER_LITERAL>: "2">
     Return:   Literal
```

Finally returning back up the chain to the root node, `Expression`:

```
     [ ... ]
   Return:   MultiplicativeExpression
  Call:   AdditiveExpression
 [ ... ]
Return:   ConditionalExpression
Return:   Expression
```

That's all there is to the expression grammar. The depth of the parse tree can seem intimidating, but it's necessary in order to handle the wide variety of operators that Java provides.

Summary

The JavaCC grammar is large and complicated, but perhaps this chapter has helped to clarify it. More generally, you can defang any complex grammar by taking it apart piece by piece and isolating and examining each production and token definition. If there's a particularly confusing section, remember that the JavaCC debug options like `DEBUG_TOKEN_MANAGER` and `DEBUG_PARSER` can be very helpful. If JavaCC can process a grammar, you can find a way to understand it too!

Chapter 9. Testing JavaCC Parsers

The universe rings true wherever you fairly test it. --C. S. Lewis

Benefits of Automated Testing

Writing a JavaCC grammar is fun—what's not so much fun is debugging it. Finding all the small corners and edge cases that you forgot to handle takes some of the joy out of life. Even worse, a small change to a grammar can have unintended consequences. That little tweak to one lookahead directive can cascade into a slew of choice conflict warnings, or, worse yet, incorrect parser behavior. Frequently the only way we find these bugs is when the parser is run on some large and tangled data file that (we hope) contains all the possible input permutations or post-implementation, when they're reported by a user.

Fortunately, there's a better way to ensure that parser bugs get fixed and stay fixed. That better way is called *unit testing*. A unit test is a small program that exercises a particular bit of functionality and ensures it behaves as expected. For example, a simple test, or a *test case*, for a game might ensure that when the game is started, the score is set to 0. A collection of such test cases is a called *test suite*. Once you've got a solid unit test suite you can tweak your program with impunity knowing that your safety net of tests will notify you if you break something.

Another advantage to writing unit tests is that they provide documentation on how you expect the code to behave. Perhaps you have a tricky syntactic lookahead, or a lexical action that is only used in a few cases. Writing a unit test to exercise this functionality also lets the next programmer who works on the code see the input data and the expected results. Even if there turns out to be a bug in the way the grammar is defined, you will at least have identified the way you expected it to work. You can then change the test to expect the proper results and tweak the grammar to produce them.

If you've already got lots of code (e.g., a large grammar file that you've inherited) and no tests, don't despair; you can build your test suite incrementally. You can begin building up a safety net by writing one test case to cover a section of the grammar about which you have concerns. The next time someone reports a bug or you need to make a change to the grammar, write another test case. If you can bring yourself to do just a test case or two per day you'll eventually build up a good test suite, and every test you add will improve your understanding of the grammar. More generally, this type of "exploratory testing" is a great way to build confidence with unfamiliar code.

Introduction to JUnit

You could write unit tests for JavaCC by coding up a small test driver program that creates a parser, feeds it some input, and validates the results. If you did that, you'd find yourself writing some of the same code over and over, and adding a lot of test cases would result in a long and tangled driver. This would lead you to refactor some of the functionality out into a library, which you could then use to structure your test cases and reduce duplication.

At this point, you'll be happy to hear that Kent Beck and Erich Gamma have already gone through that process of extracting commonly used testing idioms and data structures into a library. Their efforts resulted in JUnit[1], which they released as open source and which is now the most popular Java unit testing tool. JUnit continues to be actively developed and the newest release, JUnit 4.5, was released in mid-2008.

Let's do a quick introduction to JUnit before we dive into JavaCC testing. Suppose you wanted to learn about this newfangled `java.util.ArrayList` class that everyone's blogging about. What better way to study it than to write a test program that uses it?

First, download JUnit and unzip it to a directory on your filesystem. In my case I put it in `/usr/local/junit4.5` and created a symlink so that I could get to it with `/usr/local/junit`. `junit-4.5.jar` is the jar file you'll need on your classpath, and that's located in the top level directory.

Next, here's a small JUnit test that ensures that the `ArrayList.size` method behaves as expected:

Example 9.1. A Unit Test for ArrayList

 # examples/testing/arraylist/TestArrayList.java

```
1    import static org.junit.Assert.assertEquals;
2    import org.junit.Test;
3    import java.util.ArrayList;
4
5    public class TestArrayList {
6        @Test public void size() {
7            ArrayList<String> list = new ArrayList<String>();
8            assertEquals("Initial list size should be 0", 0, list.size());
9            list.add("Hello");
10           assertEquals("After adding an item, size should be 1", 1, list.size
());
11       }
12   }
```

A few notes on this source code are in order:

- Line 1 statically imports the `assertEquals` method from the JUnit `Assert` class. This is handy since we can then invoke this method without prefixing it with the class name.
- Our test class doesn't extend any JUnit framework base class or implement any particular interface. It does, however, contain a method declaration (on line 6) that has a `Test` annotation. This tells JUnit that it's a test method. Older versions of JUnit required that test method names be prefixed with `test`.

[1] http://junit.org/

- Inside the body of the test method we create an `ArrayList` object that's generically typed so that it can only contain `String` objects[2]. Then we make an *assertion*; we call `assertEquals` to ensure that a newly created `ArrayList` object's size is 0. This is a fundamental JUnit pattern; each assertion exercises a particular bit of behavior that we want to verify. We then add an object to the `ArrayList` and use another assertion to verify that `size` returns the expected value of 1.
- `assertEquals` is not the only assertion we can use; JUnit has a bunch of others including `assertNotNull`, `assertSame`, and `assertTrue`. It also has a variety of `assertEquals` variants that accept different argument types, such as one that ensures that two `Object` arrays are equal.
- Notice that we preceded each assertion with an error message. This error message will be displayed if this test ever fails, so it's usually worthwhile to fill this argument with some informative text. After all, you may be the one who reads it!

Here's our test case in action. In order to compile and to run the test I'm placing the JUnit jar file on the classpath:

```
$ javac -classpath /usr/local/junit/junit-4.5.jar TestArrayList.java
$ java -classpath /usr/local/junit/junit-4.5.jar:. org.junit.runner.JUnitCore
TestArrayList
JUnit version 4.5
.
Time: 0.004

OK (1 test)
```

As you can see, JUnit's output is terse when all the tests pass. We get a dot for each test that passes, followed by a time at the end. When this line of dots begins to get wrap around the terminal window a few times you know you've built up a decent test suite.

Let's intentionally make a test fail to see what happens. The test below asserts that a newly created `ArrayList` object has a size of 1:

Example 9.2. A Broken Unit Test

 # examples/testing/arraylist/BadTestArrayList.java

```
1    import static org.junit.Assert.assertEquals;
2    import org.junit.Test;
3    import java.util.ArrayList;
4
5    public class BadTestArrayList {
6        @Test public void improperSize() {
7            ArrayList<String> list = new ArrayList<String>();
8            assertEquals("Initial list size should be 1 (?)", 1, list.size());
9        }
10   }
```

If we compile and run this flawed test, here's what we'll see:

```
$ java -classpath /usr/local/junit/junit-4.5.jar:. org.junit.runner.JUnitCore
BadTestArrayList
JUnit version 4.5
.E
Time: 0.004
There was 1 failure:
1) improperSize(BadTestArrayList)
java.lang.AssertionError: Initial list size should be 1 (?) expected:<1> but
was:<0>
```

[2]Well, at compile-time anyhow, since Java implements generics via type erasure.

```
 at org.junit.Assert.fail(Assert.java:91)
 at org.junit.Assert.failNotEquals(Assert.java:618)
 at org.junit.Assert.assertEquals(Assert.java:126)
 at org.junit.Assert.assertEquals(Assert.java:443)
 at BadTestArrayList.improperSize(BadTestArrayList.java:8)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke
(DelegatingMethodAccessorImpl.java:25)
 at java.lang.reflect.Method.invoke(Method.java:597)
 at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall
(FrameworkMethod.java:44)
 at org.junit.internal.runners.model.ReflectiveCallable.run
(ReflectiveCallable.java:15)
 at org.junit.runners.model.FrameworkMethod.invokeExplosively
(FrameworkMethod.java:41)
 at org.junit.internal.runners.statements.InvokeMethod.evaluate
(InvokeMethod.java:20)
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
 at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
 at org.junit.runners.BlockJUnit4ClassRunner.runChild
(BlockJUnit4ClassRunner.java:73)
 at org.junit.runners.BlockJUnit4ClassRunner.runChild
(BlockJUnit4ClassRunner.java:46)
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:180)
 at org.junit.runners.ParentRunner.access$000(ParentRunner.java:41)
 at org.junit.runners.ParentRunner$1.evaluate(ParentRunner.java:173)
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
 at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
 at org.junit.runners.ParentRunner.run(ParentRunner.java:220)
 at org.junit.runners.Suite.runChild(Suite.java:115)
 at org.junit.runners.Suite.runChild(Suite.java:23)
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:180)
 at org.junit.runners.ParentRunner.access$000(ParentRunner.java:41)
 at org.junit.runners.ParentRunner$1.evaluate(ParentRunner.java:173)
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
 at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
 at org.junit.runners.ParentRunner.run(ParentRunner.java:220)
 at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
 at org.junit.runner.JUnitCore.run(JUnitCore.java:116)
 at org.junit.runner.JUnitCore.run(JUnitCore.java:107)
 at org.junit.runner.JUnitCore.runMain(JUnitCore.java:88)
 at org.junit.runner.JUnitCore.runMainAndExit(JUnitCore.java:54)
 at org.junit.runner.JUnitCore.main(JUnitCore.java:46)

FAILURES!!!
Tests run: 1,  Failures: 1
```

Yikes! A stacktrace like that is punishment enough in itself. JUnit reports the name and line number of the test that failed, the failure message that we placed in the call to `assertEquals`, and prints the full stacktrace. Of course, in this case the problem is with the test, not the class being tested.

Now that you've seen JUnit in action, let's move onward to testing JavaCC-generated code.

Testing a Tokenizer

Let's start our JavaCC testing by seeing how we can test our tokenizer. What sorts of things do we want to test about a tokenizer? Here are a couple of possibilities:

- We want to know that the characters are being grouped into tokens of the proper kind.

- If we have any `SKIP`, `MORE`, and `SPECIAL_TOKEN` regular expression productions, we want to ensure that they are consuming the expected characters.
- We want to be certain the tokenizer fails when given a series of characters for which we haven't defined a token.
- We want to know that our lexical state transitions are happening at the expected times.

Chapter 11 walks through a few small examples of JavaCC grammars including one for parsing programs in the Logo programming language. I'll use that Logo grammar here as well to demonstrate how to test a tokenizer.

Here's the lexical specification portion of that grammar; it consists of a `SKIP` regular expression production and three token definitions. I've also set `STATIC` to `false` for this grammar to make the test code a bit cleaner:

Example 9.3. Logo Lexical Specification

 # examples/testing/testlogo/logo_tokenizer.jj (lines 20 to 27)

```
20   SKIP : {
21     " " | "\n" | "\r" | "\r\n"
22   }
23   TOKEN : {
24     <FORWARD : "FORWARD">
25     | <RIGHT : "RIGHT">
26     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
27   }
```

Consider the following test case for this tokenizer. It contains only two tests, but it's a good start:

Example 9.4. Logo Tokenizer Test

 # examples/testing/testlogo/src/TestLogo.java

```
1    import static org.junit.Assert.*;
2    import org.junit.Test;
3    import java.io.StringReader;
4
5        public class TestLogo {
6        @Test public void tokenizeMoveCommand() {
7            String cmd = "FORWARD 10";
8            SimpleCharStream cs = new SimpleCharStream(new StringReader(cmd));
9            LogoTokenManager ltm = new LogoTokenManager(cs);
10           Token t = ltm.getNextToken();
11           assertTrue("Wrong token kind for FORWARD", LogoConstants.FORWARD ==
t.kind);
12           t = ltm.getNextToken();
13           assertTrue("Wrong token kind for DIGITS", LogoConstants.DIGITS ==
t.kind);
14       }
15       @Test(expected=TokenMgrError.class) public void tokenizeFailure() {
16           String cmd = "FORWERD 10";
17           SimpleCharStream cs = new SimpleCharStream(new StringReader(cmd));
18           LogoTokenManager ltm = new LogoTokenManager(cs);
19           ltm.getNextToken();
20       }
21   }
```

The first test, `tokenizeMoveCommand`, tests a small piece of input data: `FORWARD 10`. It ensures this input data is properly separated into a `FORWARD` token and a `DIGITS` token. This implicitly tests the `SKIP` regular expression production; if the `SKIP` is not

properly skipping the whitespace in the input stream our series of tokens will be different.

The second test, `tokenizeFailure`, is a bit different. The `@test` annotation preceding this test contains a parameter, `expected=TokenMgrError.class`. This tells JUnit that the test should throw an exception of type `TokenMgrError`. If this exception is not thrown, JUnit will consider the test to be a failure. Since the characters that we're testing, `FORWERD`, has a spelling error, the tokenizer shouldn't recognize this as a valid token and should indeed throw this exception.

Notice that in the test we're doing comparisons like `LogoConstants.FORWARD == t.kind`. A shorter way to write this would be to statically import `LogoConstants` and then simply use `FORWARD == t.kind`. We can't do that in this case since types can't be imported from the unnamed package[3], but you may want to do that in your test cases to make them more readable.

Here's a successful run of our two tests:

```
$ java -classpath .:/usr/local/junit/junit-4.5.jar:. org.junit.runner.JUnitCore
TestLogo
JUnit version 4.5
..
Time: 0.014

OK (2 tests)
```

In these tests we've embedded the input data (the Logo code) directly in the test rather than reading them in from a file. This works well with tokenizer testing since many of the code snippets needed to test the tokenizer can be quite short. But if you find yourself concatenating a series of strings in order to build up suitable data for a test case, you may want to externalize the code into a file and avoid having to escape newlines and quotes.

Although these tests are short, they're exercising quite a bit of code; the entire tokenizer will be called into action to lex the input data. Since this goes against the usual strategy for unit tests, which is to write tests that exercise a small amount of functionality, it might be more proper to call them *functional tests* or perhaps *integration tests*. But the concept is the same; we're writing code to test our program and we can run these tests whenever we choose to ensure that nothing's broken.

Testing a tokenizer is not difficult, but it can be tedious. If you try to get complete code coverage, at some point you'll find that the test code begins to repeat the lexical specification. A good strategy may be to write a couple of initial tests to ensure that tokens are being created as expected from some basic input strings. Next you may want to write tests that nail down any custom behavior, such as lexical actions that transform a token. It's also a good idea to write tests to cover any state transitions since their behavior is not always obvious.

Testing a Parser

Now let's look at testing a JavaCC-generated parser. If you only have time to write a few tests, writing them for the parser may be the bigger win since parser tests will implicitly test the tokenizer as well.

[3]Per JLS 7.5, "It is a compile time error to import a type from the unnamed package."

In this section we'll show how we can develop some tests for the BeanShell parser. To quote from the BeanShell web site [4], BeanShell is "a small, free, embeddable Java source interpreter with object scripting language features, written in Java." BeanShell is embedded in a variety of products and is currently the topic of a Java Specification Request (JSR-274). The interpreter uses a JavaCC-generated parser, so it fits our purposes. Besides, it's always nicer to see testing techniques demonstrated on a real-world example.

One way to get started testing a parser is similar to the way we tested a tokenizer. We'll construct a string with simple input data and pass that to the parser for validation. If we don't get any exceptions, we know that the parser was at least able to build a parse tree from the data. Here's a simple test case that exercises the Beanshell parser with two unit tests. The first test, `parseGibberish`, ensures an exception is thrown when we feed in some nonsense data, and the second test, `parseGoodCode`, ensures a valid statement is accepted:

Example 9.5. Simple BeanShell Parser Test

 # examples/testing/beanshell/src/bsh/TestValidateCode.java

```
1    package bsh;
2    import org.junit.Test;
3    import java.io.StringReader;
4    public class TestValidateCode {
5        @Test(expected=ParseException.class) public void parseGibberish() throws
 ParseException {
6            String code = "fizzle";
7            Parser p = new Parser(new StringReader(code));
8            p.Line();
9        }
10       @Test public void parseGoodCode() throws ParseException {
11           String code = "int x = 42;";
12           Parser p = new Parser(new StringReader(code));
13           p.Line();
14       }
15   }
```

A few notes on this test case:

- When we wrote the Logo tokenizer tests we were testing a class, `LogoTokenManager`, that was in the default package and was in the same directory. The BeanShell parser is in the `bsh-2.0b5.jar` file, so we have to place it on the classpath before compiling and running the tests.
- As you can see on line 1 of the code, we've put the test case in the `bsh` package. Usually I'd put unit tests in a "separate but parallel" test package hierarchy, the package `test.bsh` for example, to avoid cluttering up the main source tree. In this case, though, the BeanShell node files have package-level access modifiers, so in order to invoke methods on them we'll need to have the tests in the same package. The advantage of this is that we don't need any `import` statements to bring in the BeanShell types. Another alternative to this parallel hierarchy would be to use the Java reflection API to access the type features.
- We're starting a parse by instantiating the `Parser` object with a `StringReader` and calling the parser's start symbol, `Line`. We're not doing any particular testing of the results; this is just a sanity check.

[4] http://beanshell.org

- As mentioned at the beginning of this section, this test exercises the tokenizer as well as the parser; the input data has to be separated into tokens before the parser can build a parse tree. While doing a few tests of this sort may lessen the need for general tokenizer tests, you'll probably still want specific tests to verify and provide examples of any complicated tokenizer behavior.

Here's this test in action:

```
$ java -classpath lib/junit-4.5.jar:lib/bsh-2.0b5.jar:src/
org.junit.runner.JUnitCore bsh.TestValidateCode
JUnit version 4.5
..
Time: 0.04

OK (2 tests)
```

The test above exercises the entire parser; that is, it calls the start symbol, `Line`, and lets the parser figure out which nonterminals are contained in the input data. Sometimes, however, we might want to write a more focused test case that examines just one nonterminal. JavaCC lets us do this by directly invoking a specific nonterminal.

For example, suppose we wanted to ensure that the `Literal` production correctly parsed the input value `void`. The section of the `Literal` production that handles `void` values does so by calling a `VoidLiteral` production and then using a syntactic action to set a field on `jjtThis`[5]; it's shown below:

```
# examples/testing/beanshell/bsh.jjt (lines 1034 to 1035)

1034  VoidLiteral() {
1035      jjtThis.value = Primitive.VOID; }
```

We could write a test to call only this nonterminal:

Example 9.6. Testing One Nonterminal

```
# examples/testing/beanshell/src/bsh/TestLiteral.java

1    package bsh;
2
3    import static org.junit.Assert.*;
4    import org.junit.Test;
5    import java.io.StringReader;
6
7    public class TestLiteral {
8        @Test public void nullValueSet() throws ParseException {
9            String code = "void";
10           Parser p = new Parser(new StringReader(code));
11           p.Literal();
12           Node n = p.popNode();
13           assertTrue("First node should be a Literal", n instanceof
BSHLiteral);
14           assertEquals("Type should be void", ((BSHLiteral)n).value,
Primitive.VOID);
15       }
16   }
```

This test feeds in a single token of input, checks the type of the node that's produced and ensures that the `value` field is set to the proper value.

Our input data consists of a valid BeanShell literal, but it wouldn't be considered a valid BeanShell script because BeanShell requires a semicolon as a statement terminator. By invoking `Literal` alone, we can test just one nonterminal with very precise input and limit the amount of grammar code that gets exercised.

[5]This touches on testing the AST; we'll cover that in more detail in the next section.

Testing the AST

So far we've seen how to validate a tokenizer and how to ensure the parser is building a parse tree and performing syntactic actions. We can also write tests to verify the structure of the AST. The BeanShell grammar is a JJTree grammar, so we'll continue to use it as a testbed.

Consider this small snippet of Beanshell code: `x = 42;`. The AST for this code snippet consists of an `Assignment`[6] which has two `PrimaryExpression` children. Each of the `PrimaryExpression` nodes in turn has one child node. Here's what it looks like:

```
Assignment
 PrimaryExpression
  AmbiguousName
 PrimaryExpression
  Literal
```

We can write a small unit test that verifies this structure by poking around the AST using various JJTree utility methods. `Parser.popNode()` gives us the root level node, and we can use `jjtGetChild` and `jjtGetNumChildren` to spelunk around from there.

Example 9.7. Testing AST Structure

 # examples/testing/beanshell/src/bsh/TestAST.java

```
1    package bsh;
2    import static org.junit.Assert.assertTrue;
3    import static org.junit.Assert.assertEquals;
4    import org.junit.Test;
5    import java.io.StringReader;
6    public class TestAST {
7        @Test public void nullValueSet() throws ParseException {
8            String code = "x = 42;";
9            Parser p = new Parser(new StringReader(code));
10           p.Line();
11           Node n = p.popNode();
12           assertTrue(
13               "Root should be an assignment",
14               n instanceof BSHAssignment);
15           assertTrue(
16               "First child should be a PrimaryExpression",
17               n.jjtGetChild(0) instanceof BSHPrimaryExpression);
18           assertEquals(
19               "Root should have two children",
20               2, n.jjtGetNumChildren());
21           assertTrue(
22               "Second child should be a PrimaryExpression",
23               n.jjtGetChild(1) instanceof BSHPrimaryExpression);
24           assertTrue(
25               "First grandchild should be an AmbiguousName",
26               n.jjtGetChild(0).jjtGetChild(0) instanceof BSHAmbiguousName);
27           assertTrue(
28               "Second grandchild should be a Literal",
29               n.jjtGetChild(1).jjtGetChild(0) instanceof BSHLiteral);
30           assertEquals(
31               "Second grandchild's value should be 42",
32               ((BSHLiteral)n.jjtGetChild(1).jjtGetChild(0)).value.toString(),
"42");
33       }
34
35   }
```

[6]Since this grammar has **`NODE_PREFIX`** set to "BSH", the class name is actually **`BSHAssignment`**. Hopefully dropping that prefix makes things more readable.

Exploring the AST using the `Node` methods gives you lots of flexibility; for each node you can verify the number of children it has, its type, and so forth. The downside of testing the AST like this is that it's rather tedious. You have to manually walk the AST, locating and checking each node in a rather cumbersome fashion. If only there were a better way...

Testing the AST with XPath

Fortunately, there is another way to navigate an AST. Consider the AST snippet from the previous section:

```
Assignment
 PrimaryExpression
  AmbiguousName
 PrimaryExpression
  Literal
```

Suppose we transform this into an XML document by changing each node into an XML element[7]:

```
<Assignment>
 <PrimaryExpression>
  <AmbiguousName/>
 </PrimaryExpression>
 <PrimaryExpression>
  <Literal/>
 </PrimaryExpression>
</Assignment>
```

Now the structure of the AST is preserved, but it's a XML document. The World Wide Web Consortium (W3C) has defined[8] a standard Domain Specific Language (DSL), XPath, for querying XML documents. For example, you could access the `Literal` element in the XML document above by using the XPath query `//Assignment/PrimaryExpression/Literal`. Using XPath is generally much more concise than using the Document Object Model (DOM) API to locate the same information.

Here's where the fun begins. There's an excellent open source XPath engine, Jaxen[9], that provides full support for interpreting and executing XPath queries within Java programs. As it executes an XPath query, Jaxen uses a `DocumentNavigator` interface to traverse the XML document—to iterate over the attributes of an element, to move up and down the node tree, to get the element names, and so forth. By writing an implementation of `DocumentNavigator` that understands how to traverse an AST rather than an XML document, we can use XPath to query an AST.

This idea may be a little easier to understand if we see an example of a unit test that uses this technique. The test below uses a custom `DocumentNavigator` to query the AST and verify various things. For example, the root `Assignment` node should have two `PrimaryExpression` children. The `checkNodeCounts` test verifies that by running the XPath query `//PrimaryExpression`, which translates to "find all the `PrimaryExpression` nodes under the root node." This query should return two nodes, and as you can see below, it does. The `checkNameValue` test queries for `AmbiguousName` nodes where the `text` field's value is `x`; you can see that the proper value is being returned there as well.

[7]True, this isn't valid XML since there's no header.
[8]http://www.w3.org/TR/xpath
[9]http://jaxen.org/

Example 9.8. Testing AST Structure with XPath

 # examples/testing/beanshell/src/bsh/TestWithXPath.java

```
1    package bsh;
2    import static org.junit.Assert.assertEquals;
3    import org.junit.Test;
4    import org.jaxen.BaseXPath;
5    import org.jaxen.JaxenException;
6    import java.io.StringReader;
7    import java.util.List;
8
9    public class TestWithXPath {
10       private static final String BEANSHELL_CODE  = "x = 42;";
11
12       @Test public void checkNodeCounts() throws ParseException,
JaxenException {
13           Parser p = new Parser(new StringReader(BEANSHELL_CODE));
14           p.Line();
15           Node root = p.popNode();
16           String xpath = "//PrimaryExpression";
17           List nodes = query(xpath, root);
18           assertEquals("Unexpected result from " + xpath, 2, nodes.size());
19           xpath = "//PrimaryExpression/Literal";
20           nodes = query(xpath, root);
21           assertEquals("Unexpected result from " + xpath, 1, nodes.size());
22       }
23
24       @Test public void checkNameValue() throws ParseException, JaxenException
 {
25           Parser p = new Parser(new StringReader(BEANSHELL_CODE));
26           p.Line();
27           String xpath = "//PrimaryExpression/AmbiguousName[@text='x']";
28           List nodes = query(xpath, p.popNode());
29           assertEquals("Unexpected result from " + xpath, 1, nodes.size());
30       }
31       private List query(String xpath, Node node) throws JaxenException {
32           BaseXPath base = new BaseXPath(xpath, new DocumentNavigator());
33           return base.selectNodes(node);
34       }
35   }
```

As you can see from the above tests, we can make our AST tests much more readable and concise by harnessing the power of XPath. This technique requires a couple of support classes; read on for details:

- First of all, the four classes necessary to make this work are included in the source code for the book, so you can peruse and reuse them at your leisure.
- The `DocumentNavigator` is the core class that allows client code to navigate around what's known as an XML Information Set, or InfoSet[10]. Technically, it extends `org.jaxen.DefaultNavigator` (and thus implements `org.jaxen.Navigator`) and provides implementations for about 20 methods. Among those are methods to supply "axis iterators" for the various XPath *axes*. For example, if a client program wants to iterate over all the children of a particular node, `getChildAxisIterator` will build a new `NodeIterator` object to do the job.
- `Attribute` is a simple data object that models the attributes of the `bsh.Node` implementations. It has accessors for an attribute name, value, and a parent node object. This allows you to write *predicate*s that include tests for attribute values such as `[@text='hello']`.

[10]http://www.w3.org/TR/xml-infoset/

- `AttributeAxisIterator` is a `java.util.Iterator` implementation that allows client code to loop over the `Attribute` objects contained by a `bsh.Node` implementation. When instantiated, `AttributeAxisIterator` maps all the public fields of type `String` into `Attribute` objects. You may want to change this to include more types, or perhaps to invoke methods rather than reading fields. It all depends on how the data is structured on your node implementations.

- `NodeIterator` is an abstract class. It's a partial `java.util.Iterator` implementation that allows the `DocumentNavigator` to navigate along various axes relating to `Node` objects. It is subclassed by several anonymous inner classes within `DocumentNavigator`; these inner classes override the two methods necessary to make this a concrete class.

- Both `AttributeAxisIterator` and `NodeIterator` are read-only iterators, so calling `remove` on them will result in an `UnsupportedOperationException`. This makes sense because XPath is used to querying a structure, not to change it.

The original credit for this technique goes to Daniel Sheppard, who implemented this for the Java static code analysis utility PMD. It's proved very useful for the PMD project, so perhaps it'll be handy for your projects as well.

As a final note, you can also test an AST by writing a test class that uses a Visitor to receive a callback for the nodes that you want to examine. This is a low-tech approach compared to using XPath expressions, but it's certainly an option if you want to avoid adding a dependency on Jaxen just for testing.

Summary

In this chapter you've learned how to write tests to cover various pieces of JavaCC functionality—the tokenizer, the parser, and the AST. You've also seen how to use JUnit, a popular Java testing tool. Hopefully this chapter has motivated you to write tests and provided some helpful strategies for making your JavaCC grammar tests more effective.

Chapter 10. JavaCC And Eclipse

Nothing that you have not given away will ever be really yours. --C. S. Lewis

Why Use an Integrated Development Environment?

Most of this book shows JavaCC in action from the command line. It's the least-common-denominator approach—everyone has access to a command line, and all the options are available this way. Besides, if you can use JavaCC from the command line you can usually figure out how to use it in higher level tools such as Ant and Maven.

But most Java developers use some sort of Integrated Development Environment (IDE) to work on their code. An IDE such as Eclipse, NetBeans, IDEA, or JDeveloper provides syntax highlighting, code folding, refactoring support, integrated debugging, easy navigation of method invocations, and a variety of other bells and whistles that make developing code faster and more enjoyable.

Although IDEs have a good amount of built in functionality, the IDE developers can't think of everything that you might need. Therefore, most IDEs support some sort of plugin or extension mechanism that allows you to extend the IDE's capabilities. For example, you might want to program in an enhancement so that the IDE closes bugs in your bug tracker, or deploys a product to a server, or works with some new file type—like JavaCC grammars. Thanks to Rémi Koutchérawy, there's an excellent (and free!) JavaCC plugin available for Eclipse[1].

Here's an overview of what the Eclipse JavaCC plugin provides:

- A specialized editor for JavaCC grammars that supports syntax highlighting, brace matching, and grammar reformatting.
- A console for displaying the output from JavaCC and JJTree execution. This console is nicely hyperlinked to locations within the grammar file making navigation quick and easy.
- Support for JavaCC, JJTree, JTB, and JJDoc compilation and options. You can configure your project with a particular set of options (e.g., `STATIC` and `CACHE_TOKENS`) so that JavaCC will be run with these options even though they're not listed in the grammar file itself.
- Integration with standard JavaCC use cases; for example, compiling a JJTree grammar also invokes JavaCC on the generated grammar to produce the parser.

It's all very handy stuff, and the syntax highlighting alone can make using the plugin worthwhile. Let's dive right in and get this plugin up and running.

[1] http://eclipse-javacc.sf.net/

Installation

First, the JavaCC plugin requires Eclipse 3.2 or later to run. As I write this, I'm using Eclipse 3.4.2 and the plugin version 1.5.12 works quite well with that version[2].

We can install the plugin by using Eclipse itself to find and download it. This works because the plugin developers have conveniently structured the web site so that Eclipse can query for plugin versions and data files. You could also install it by downloading a zip file and extracting it, but using Eclipse' built in installer is much easier.

Start Eclipse and select the "Help" menu followed by "Software Updates" option and the "Available Software" tab. We need to tell Eclipse where it can find the plugin, so click the "Add Site" button on the right side of the dialog box. Enter "http://eclipse-javacc.sf.net" in the "Location" field and click "OK". Now, check the box next to the new row labeled "http://eclipse-javacc.sf.net" that appeared in the main list box and deselect the radio button labeled "Show only the latest versions of available software". Eclipse will download a list of all the available versions of the JavaCC plugin. Select the checkbox marked "JavaCC Feature 1.5.12" and click the "Install" button on the upper right side of the dialog box.

Eclipse will grind away for a bit and then display the JavaCC plugin contents. Click "Next" and Eclipse will present you with a license agreement; select the "I accept" radiobutton and click "Finish". Eclipse will download and install the plugin and then suggest that you restart the IDE before continuing. That's probably a good idea, so click "Yes" to initiate the restart.

Once Eclipse restarts you can verify that the plugin is installed by clicking the "Help" menu and selecting the "About Eclipse Platform" menu item. When the dialog box opens, click the "Plug-in Details" button, then click the "Provider" heading to sort the plugins and scroll to the bottom (more or less, depending on what other plugins you have installed) of the list. The JavaCC plugin should be listed with a "Plug-in Name" of "JavaCC Plug-in".

Basic Operation

Let's build a little Java project around a simple JavaCC grammar. From the File menu in Eclipse, select the "New" option, then select the "Java Project" submenu option. Now enter a project name (such as "javacc-test") in the dialog box and click "Finish." After Eclipse puts the project together, the basic project interface should open up, or, if you're still at the default empty view, click the "Go to the Workbench" item on the right side of the window and the project navigation frames should appear.

Creating a Grammar

To create the grammar file, right-click on the "javacc-test" project name and click the "New" menu item. When the submenu opens, select the "Other" option. Expand the "JavaCC" subtree in the resulting dialog box, then click "JavaCC Template File" followed by the "Next" button. This opens a dialog box that allows us to select the

[2]As of this writing 1.5.13 was available, but I encountered a couple of problems with that version and so backed down to 1.5.12. There may well be a .14 release by the time you read this, in which case, give that a whirl.

name, package, and type of grammar file we want. Click the "Browse" button next to the "Folder" field and select "javacc-test/src", leave the "Package" field blank, press the "JJ file" radio button and enter "hello" in the "File name" field, then click the "Finish" button. This creates a grammar file (`hello.jj`) in the "src" directory of the "javacc-test" project and Eclipse will open the grammar automatically.

Let's poke around that `hello.jj` file a bit and see what the plugin gives us:

- The plugin generated a file with an `options` header, a `PARSER_BEGIN/PARSER_END` section with a class definition and a `main` method, and some token and nonterminal definitions. In other words, we've got a good set of standard JavaCC constructs to start with.
- The plugin opens a new Eclipse view: the "JavaCC console." Plunging ahead, the plugin runs JavaCC against the generated grammar and displays the results on this console, so you should see various messages ending with "Parser generated successfully" displayed. Clicking the eraser icon at the top of the view will clear the console.
- The grammar is nicely syntax-highlighted. In other words, JavaCC keywords like `options` and `SKIP` are displayed in green, literals are displayed in blue, Java keywords are red, and so forth. You can customize these colors by clicking the "Window" menu heading, selecting "Preferences", clicking the "JavaCC Preferences" on the left side of the dialog box, and then twiddling the colors as you see fit.[3]
- Click just to the right of the left curly brace character that appears after the `options` keyword. Notice that the corresponding closing brace is now wrapped in a little box. This is called *brace matching*, and it's quite handy when you've got a complex nonterminal or other deeply nested structure. If you click next to the parenthesis after `PARSER_BEGIN`, you'll see that this also works for parentheses. It works the other way too—if you click to the right of a closing parenthesis or curly brace you'll see that the starting character is highlighted.
- You can process the grammar with JavaCC by right-clicking on either the grammar itself or the grammar file name in the "Package Explorer" view, then selecting the "Compile with JavaCC" menu item. As noted before, the output appears in the console view.
- Since we're working in Eclipse we've also got all of Eclipse' fine code navigation capabilities. For example, once you've run the "Compile with JavaCC" menu item, you can click on the "default package" node in the "Package Explorer" view and open one of the files. If you open `eg1.java`, you can click on the `jj_consume_token` method call in the `one_line` method and Eclipse will navigate to the `jj_consume_token` method declaration. To return to the previous location, hit Alt +Left or click the left arrow.

JavaCC Options

To customize the way the plugin is processing the grammar, right-click on the "javacc-test" project node in the "Package Explorer" view (which is usually located on the left side of the Eclipse main window) and select the "Properties" menu

[3] I found that I had to restart Eclipse to get color changes to take effect. Your mileage may vary.

option[4]. Select the "JavaCC options" line from the list on the left side and you'll see a series of tabs for various options.

One handy option on this dialog box enables you to use a custom jar file for JavaCC. This can be useful if you've been using an older version of JavaCC (say, JavaCC 4.0) and want to see how your grammar works with JavaCC 4.2. To do this, just click the "browse" button next to the "Set the JavaCC jar file" text field and locate the new `javacc.jar` file. Once it's selected, click "OK" to dismiss the dialog box, then right-click on your grammar and select "Compile with JavaCC" to process it. The output will appear in the console view with a new JavaCC version number. Be warned—this option is cleared whenever you create a new grammar, so make sure you go back to this dialog box if you want to use an external `javacc.jar` for all your newly-created grammars.

Note that the reason you can select a different `javacc.jar` file is that the plugin doesn't actually load JavaCC into the Eclipse memory space and call methods on the JavaCC classes. Instead, it invokes JavaCC as an external process, feeding it any command line options you've specified and capturing the output. Since this requires starting up a new JVM, it's a little slower than calling the JavaCC classes directly, but it does give the plugin the flexibility to easily switch out the jar file that it's using.

Another interesting option on this page is "Automatically suppress warnings in generated files." JavaCC, like most code generators, creates some rather tangled and obfuscated code. This is usually fine; the generated code isn't really meant for human consumption, so as long as it's correct and it's fast, we're happy. But most IDEs keep a running count of what it considers to be code problems, and JavaCC's output is full of things to complain about. Checking the "Suppress warnings" box will annotate all the class declarations that JavaCC generates with `@SuppressWarnings("all")`. This tells Eclipse (and various other source code checking tools) that there's no need to report any problems found in these classes. Unless you're actually working on JavaCC itself and want to fix the problems at the root, you'll probably want to enable this option.

The rest of the choices on this dialog box are all the standard JavaCC options: `STATIC`, the lookahead value, `DEBUG_PARSER`, and so forth. Toggling these options will affect the generated code next time you select the "Compile with JavaCC" menu option.

Let's modify our `hello.jj` grammar to see what else the plugin does. Edit the file and remove some of the code that the plugin created until you winnow things down to this code:

```
# examples/eclipse/hello.jj

1    options {
2      JDK_VERSION = "1.5";
3    }
4    PARSER_BEGIN(Hello)
5    public class Hello {
6    }
7    PARSER_END(Hello)
8    TOKEN : {
9     <HELLO : "hello">
10   }
11   void hi() : {} {
```

[4]A shortcut is to select the project node and hit Alt-Enter

```
12    <HELLO> there()
13   }
14   void there() : {} {
15    "there"
16   }
```

The JavaCC plugin also finds mistakes in the grammar format as you are working in the file. To see this, delete the colon after the `TOKEN` keyword and hit Ctrl-S to save the grammar file. Two things happen:

- A red "x" appears in the margin to the left of the `TOKEN` keyword. This visual indicator tells you that something is awry with this section of the grammar.
- Since the plugin runs JavaCC on the grammar each time you save it, the console output now shows the error message that JavaCC produced when presented with this malformed grammar. The line and column numbers are highlighted in blue and linked to the grammar file; if you click on those numbers Eclipse will navigate to that point in the grammar. While this is more valuable with a large grammar file, it's quite handy with a small grammar as well.

Re-add the colon and save the grammar again; the "x" goes away and the console now shows a successful JavaCC run. Note that on the main options page there's an option to clear the console before each build; you may want to enable that option to keep the console view from getting cluttered.

Another helpful feature is the ability to navigate around the grammar. Click on the `<HELLO>` token reference in the `hi` nonterminal body. If you then hit the F3 key or press the Ctrl key and click simultaneously, the cursor will hop up to the token declaration. Pressing Alt+Left will return the cursor to the previous location. This also works with nonterminals; click on the `there()` nonterminal reference in the `hi` nonterminal, hit F3, and the cursor will move down to the `there` nonterminal declaration. To get a real appreciation for the value of this feature you need a large grammar, so load up the Java grammar from the JavaCC examples directory[5], scroll down to the parser section and navigate around using F3.

One thing the plugin can't do (yet) is navigate to a token definition when you're only using the literal value of that token. For example, if you have a `HI` token defined as `"hi"`, then use `"hi"` in a nonterminal, clicking on that `"hi"` literal won't move the cursor back to the `HI` token definition. The same applies to navigating to a Java function that's invoked from within the grammar—within a lexical or syntactic action, for example. This is one more reason to follow the advice on page 109—namely, to move as much code outside of the grammar as possible so that you can use your IDE to its fullest advantage.

JJTree Usage

Let's create a JJTree grammar and see what the plugin gives us. Follow the steps on page 200 to create a new grammar file, but this time put `hellotree` in the "Name" field, select the "JJT-file" radio button, and then copy and paste the contents of `hello.jj` into `hellotree.jjt`. Since a valid JavaCC file is also a valid JJTree file this will work swimmingly.

[5]It's also in this book's examples directory at **`examples/jjdoc/Java1.5.jj`**

Now that the grammar is there, hit Ctrl-S to save the `hellotree.jjt` file and look at the console output[6]. The plugin runs JJTree on the grammar file, producing a `hellotree.jj` file, then runs JavaCC on that intermediate grammar file to produce the actual parser and tokenizer. Note that the `hellotree.jj` intermediate file is visible in the "Package Explorer", but that the label next to it indicates that it was derived from `hellotree.jjt`.

From the plugin's perspective, there aren't many other differences between a JJTree grammar and a JavaCC grammar. One nice feature is that if you add a conditional node descriptor to a nonterminal, the plugin will let you navigate from that descriptor to the nonterminal definition. For example, clicking on the `#there(>1)` node descriptor in this production will move you to a `there` nonterminal:

```
# examples/eclipse/hello_conditional.jjt (lines 11 to 13)

11   void hi() : {} {
12    <HELLO> #there(>1)
13   }
```

Of course, there are also a slew of JJTree options that are built into the main options dialog box; just select the project node, hit Alt-Enter and click the "JJTree" tab to see them.

JJDoc Usage

The plugin can also generate JJDoc output. To do this, open a grammar file, right-click on it, and select "Compile with JJDoc" from the menu. This will generate an HTML file in the root directory of the project with the same name as the grammar, so running this on `hello.jj` would give you a `hello.html` in the root directory.

You can tweak the JJDoc options by clicking on the project node in the "Package Explorer", hitting Alt-Enter, and selecting the "JJDoc options" tab. The current version of the plugin appears to have a bug—changing the value of the `OUTPUT_DIRECTORY` text field doesn't cause the JJDoc output to change. There's a bug report for that so the problem may well be fixed by the time you read this.

Odds and Ends

Templates

The default grammar that the plugin generates is helpful because it includes most of the sections that you'd typically want in a grammar. Once you've created a new grammar, you can just delete the parts you don't need. If you generate grammars regularly, though, you may want to tweak this default grammar. The template is located in the plugin's directory structure inside Eclipse in:

```
[your Eclipse root directory]/plugins/sf.eclipse.javacc_1.5.12/templates/
new_file.jj
```

There's a `new_file.jjt` and a `new_file.jtb` as well. After editing any of those template files, you can regenerate a grammar and see your changes. Also, there's a `<?package_declare?>` macro in each of those files that will be replaced with the cor-

[6]If the console is getting cluttered, click the eraser icon at the top of the view to clear it

rect package declaration[7]. Note that these files exist within a specific plugin version's directory, so when the next version of the plugin comes out you'll need to copy over any template file changes you've made.

Options

If you look at the options dialogs you'll see that `JDK_VERSION` defaults to "1.4". This default value is not explicitly passed in on the command line when invoking JavaCC, so if you've got `JDK_VERSION` set to "1.5" in your grammar file, you'll get a parser and tokenizer that have Java 1.5 constructs. To avoid getting Java 1.5 features you can force the plugin to pass in a lower value for `JDK_VERSION` by editing this option and changing it to, say, "1.1".

This brings up a larger issue—should you use the options dialog in the plugin, or list all the options that you want in the grammar file itself? I tend to lean towards listing the options in the grammar file. This has the advantage of still working if you choose to generate the parser via Ant or some means other than Eclipse; your options are all there in the file, you're not dependent on an external tool to pass them in. The same reasoning applies if you're on a project with co-workers who aren't using Eclipse; it may be more useful to keep the options inside the grammar. Even so, being able to override the options in the grammar file by selecting them in the options dialog is quite handy. If you want to turn on `DEBUG_TOKEN_MANAGER` for a minute or two you don't have to edit the grammar file directly. This also means that there's no chance of you deluging your co-workers with a flood of debugging messages the next time they regenerate and run the tokenizer because you accidentally checked in a grammar file with `DEBUG_TOKEN_MANAGER` enabled.

Formatting

The plugin supports automatic reformatting of JavaCC grammars. Just open a grammar and either right-click on the grammar file and select the "Format" option, or just hit Ctrl-Shift-F. You can also select an individual grammar element, such as a token declaration or a nonterminal, and reformat only that section. Note that if there's a typo in the grammar the formatting operation won't work, so run a quick "Compile with JavaCC" to ensure the grammar is valid before formatting.

Token declarations are reformatted so that each goes on a separate line, so a token declaration like this:

```
TOKEN : {
 <A : "A"> | <B : "B"> | <C : "C">
}
```

will be reformatted to look like this:

```
TOKEN : {
 <A : "A">
 | <B : "B">
 | <C : "C">
}
```

More complex token definitions are formatted so that each alternative appears on its own line. Nonterminals are formatted in a similar fashion; alternatives within an expansion are separated into their own lines and most whitespace is stripped away.

[7]Rémi comments that "This <? syntax looks like PHP, but that's where the resemblance ends—don't try calling `preg_replace`!"

I found that reformatting doesn't always produce better-looking code. For example, I tend to format complex token definitions so that some expansions get an extra indentation level for each grouping:

```
# examples/eclipse/string_literal_before.jj (lines 37 to 51)
```

```
37   TOKEN : {
38     < STRING_LITERAL:
39         "\""
40         (   (~["\"","\\","\n","\r"])
41           | ("\\"
42                ( ["n","t","b","r","f","\\","'","\""]
43                | ["0"-"7"] ( ["0"-"7"] )?
44                | ["0"-"3"] ["0"-"7"] ["0"-"7"]
45                )
46             )
47         )*
48         "\""
49     >
50   }
```

Reformatting this token definition flattens things out considerably, but I think it also makes it harder to see where each expansion begins and ends:

```
# examples/eclipse/string_literal_after.jj (lines 37 to 42)
```

```
37   TOKEN:{
38     <STRING_LITERAL:"\""((~["\"", "\\", "\n", "\r"])
39     | ("\\"(["n", "t", "b", "r", "f", "\\", "'", "\""]
40     | ["0"-"7"](["0"-"7"])?
41     | ["0"-"3"]["0"-"7"]["0"-"7"]))*"\"">
42   }
```

The same applies to the way the plugin formats nonterminals; extraneous whitespace is removed and all the nonterminals take on a uniform appearance, but I find that it usually makes long or complex nonterminals harder to read.

That said, this feature can be quite useful for standardizing the look of your simpler tokens and nonterminals, and of course, since this is open source, the output may well be improved by the time you read this. Also, the reformatting is done using the JavaCC-generated tokenizer, so perhaps this book will give you the confidence to work on this code as a learning project! At any rate, give it a try on your grammar and see if you like the results.

Summary

That's a roundup of the Eclipse plugin. If you're using Eclipse and JavaCC together, this plugin is definitely worth a trial run. The time you save is your own!

Chapter 11. Little Grammars

Buturuga mică rastoarnă charul mare (Trans: It's the little log that overturns the carriage). --Romanian proverb

Some Small Examples

This chapter is devoted to a few short JavaCC and JJTree examples that illustrate the thought processes you might go through when you develop your own grammars. In addition, these examples show the various ins and outs of JavaCC's features as well as JavaCC's usefulness even when dealing with a relatively small parsing job.

Web Server Log Files

The world's most popular[1] web server, Apache[2], can be configured to write out a log file entry for each HTTP request it receives. The log file is a gold mine of information on a web site's activity and visitors: it contains referrers, number of bytes transferred, the date of each request, and much more. Since it behooves every webmaster to analyze those files, there are a host of utilities, both commercial and open source, that parse these logs and generate pretty reports chock full of color charts and graphs. Let's join the crowd and implement another one—based on JavaCC of course!

Before we tokenize the log file data, we need to know what it looks like. Here's a sample line from the log file of a web site[3] that I help administer:

```
rubyforge.org 72.36.154.210 - - [09/Dec/2006:21:54:14 -0500] "GET /projects/
rubygems/ HTTP/1.0" 302 - "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://
www.google.com/bot.html)"
```

This series of items consists of the host name, the requestor's IP address, the requestor's user and log name (both of which are blank in this case), request date, URI requested, response code and length, referrer and, finally, the user agent—which is the identifying string that the client program sent with the request. Generally, all these items can be expressed as either a piece of data, a piece of data enclosed in brackets, or a piece of data enclosed in quotes; all the items are separated by spaces. With that in mind, let's knock out an initial tokenizer.

[1] http://news.netcraft.com/archives/2009/03/index.html
[2] http://httpd.apache.org/
[3] http://rubyforge.org

Example 11.1. A Tokenizer for Apache Logs

 # examples/little_grammars/apache/apache_log_tokenizer.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(ApacheLog)
5    import java.io.*;
6    public class ApacheLog {}
7    PARSER_END(ApacheLog)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws Exception {
10       Reader r = new FileReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(r);
12       ApacheLogTokenManager mgr = new ApacheLogTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found a " + ApacheLogConstants.tokenImage
[t.kind] + ": " + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " " | "\n" | "\r" | "\r\n"
21   }
22   TOKEN : {
23     <QUOTED_DATA : "\"" (~["\""])* "\"">
24     | <BRACKETED_DATA : "[" (~["]"])+ "]">
25     | <DATA : (~["\r","\n", " "])+>
26   }
```

Notice how `QUOTED_DATA` and `BRACKETED_DATA` precede `DATA` in the lexical specification. This is to accommodate JavaCC's maximal munch behavior; if `DATA` appeared first, it's so inclusive that it would consume all the data. Other than that, we're not imposing much structure on the input data stream; there's no mention (yet) of any of the actual data contents such as response code, referrer, and so forth. Here's a sample run:

```
$ java ApacheLogTokenManager input.log | head -10
Found a <DATA>: rubyforge.org
Found a <DATA:> 64.62.136.205
Found a <DATA:> -
Found a <DATA:> -
Found a <BRACKETED_DATA>: [09/Dec/2006:21:54:14 -0500]
Found a <QUOTED_DATA>: "GET /pipermail/alexandria-list/2004-June/000107.html
HTTP/1.0"
Found a <DATA>: 200
Found a <DATA>: 2968
Found a <QUOTED_DATA>: "-"
Found a <QUOTED_DATA>: "Twiceler www.cuill.com/twiceler/robot.html"
```

Let's write a parser to put more structure atop our tokenizer. An Apache log file is a series of entries, each of which consists of the data we enumerated earlier. We can add a nonterminal for each of the items so that we know what they are: `ResponseCode`, `Date`, and so on. Once we've delineated each part of the HTTP request, we can gather data about that part. The grammar below contains syntactic actions that call an `AgentCollector` class to report a list of the user agents ordered by frequency. This will give us an idea of the kind of browsers our customers are using.

Example 11.2. Parsing Apache Logs

 # examples/little_grammars/apache/apache_log_parser.jj

```
1    options {
2      STATIC=false;
3    }
4    PARSER_BEGIN(ApacheLog)
5    public class ApacheLog {
6      private AgentCollector collector;
7      public void setAgentCollector(AgentCollector collector) {
8        this.collector = collector;
9      }
10   }
11   PARSER_END(ApacheLog)
12   SKIP : {
13     " " | "\n" | "\r" | "\r\n"
14   }
15   TOKEN : {
16     <QUOTED_DATA : "\"" (~["\""])* "\"">
17     | <BRACKETED_DATA : "[" (~["]"])+ "]">
18     | <DATA : (~["\r","\n", " "])+>
19   }
20   void Log() : {} {
21     (Entry())*
22   }
23   void Entry() : {} {
24     Hostname() RemoteIP() RemoteLogname() RemoteUsername() Date()
RequestMethodURLVersion() ResponseCode() ResponseLength() Referrer() UserAgent()
25   }
26   void Hostname() : {} {<DATA>}
27   void RemoteIP() : {} {<DATA>}
28   void RemoteLogname() : {} {<DATA>}
29   void RemoteUsername() : {} {<DATA>}
30   void Date() : {} {<BRACKETED_DATA>}
31   void RequestMethodURLVersion() : {} {<QUOTED_DATA>}
32   void ResponseCode() : {} {<DATA>}
33   void ResponseLength() : {} {<DATA>}
34   void Referrer() : {} { <QUOTED_DATA> }
35   void UserAgent() : {Token t;} {
36     t=<QUOTED_DATA> { collector.addUserAgentHit(t.image); }
37   }
```

Note how clean the grammar is when all the data collection code is in a separate class. This is a realistic use of JavaCC; it's rare that you'll shove all the processing logic in with the grammar itself as some of the earlier examples in this book have done. We still have to include some accessor methods in the `PARSER_BEGIN/PARSER_END` section, but moving everything else out makes it much tidier.

Here's the `AgentCollector` code:

Example 11.3. Parsing Apache Logs

 # examples/little_grammars/apache/AgentCollector.java

```
1    import java.io.*;
2    import java.util.*;
3    public class AgentCollector {
4
5      private Map<String,Integer> userAgentCount = new TreeMap<String,Integer>
();
6      private File file;
7
8      public AgentCollector(File file) {
9        this.file = file;
10     }
11
12     public void addUserAgentHit(String image) {
13       if (!userAgentCount.containsKey(image)) {
14         userAgentCount.put(image, 0);
15       }
16       userAgentCount.put(image, userAgentCount.get(image).intValue() + 1);
17     }
18
19     public void collect() throws IOException, ParseException {
20       Reader r = new FileReader(file);
21       ApacheLog apacheLog = new ApacheLog(r);
22       apacheLog.setAgentCollector(this);
23       apacheLog.Log();
24     }
25
26     public void printReport() {
27       Set<Map.Entry> sortedAgents = new TreeSet<Map.Entry>(new
Comparator<Map.Entry>() {
28         public int compare(Map.Entry obj1, Map.Entry obj2) {
29           return ((Integer)obj2.getValue()).compareTo((Integer)obj1.getValue
());
30         }
31       });
32       sortedAgents.addAll(userAgentCount.entrySet());
33       for (Map.Entry entry: sortedAgents) {
34         if (((Integer)entry.getValue()).intValue() < 10) {
35         break;
36       }
37       System.out.println(entry.getValue() + " hits from " + entry.getKey());
38       }
39     }
40
41     public static void main(String[] args) throws Exception {
42       AgentCollector collector = new AgentCollector(new File(args[0]));
43       collector.collect();
44       collector.printReport();
45     }
46   }
```

You can see that we've added `TARGET_JDK="1.5"` to the grammar's `options` header; this gives us free reign in our `AgentCollector` class to store the user agent string and the count in a genericized `TreeMap`; lots of tedious casting is eliminated, and the containers are still typesafe. The autoboxing feature also comes in handy, since we can place a bare `int` in the `TreeMap` without wrapping it in an `Integer` object. We could still use the Java 1.5 features without setting the `JDK_VERSION` option, but then we'd miss out on the optimized Java 1.5-specific code that JavaCC can generate.

The `main` method in `AgentCollector` handles the arguments passed in on the command line, while the `collect` method creates and calls the parser. We can create an `ApacheLog` parser object since we disabled the `STATIC` option in the grammar. If we had left `STATIC` at its default value, we would have used the static `ApacheLog.Log()` method instead.

Finally, here's a sample run. Notice most of the traffic consists of bots! Such are the usage patterns of public web sites.

```
$ javacc -JDK_VERSION=1.5 apache_log_parser.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file apache_log_parser.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java AgentCollector input.log
511 hits from "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/
bot.html)"
114 hits from "Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/
us/ysearch/slurp)"
27 hits from "Mozilla/5.0 (X11; U; Linux i686; pt-BR; rv:1.8.1) Gecko/20060601
Firefox/2.0 (Ubuntu-edgy)"
26 hits from "-"
23 hits from "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1) Gecko/20060601
Firefox/2.0 (Ubuntu-edgy)"
18 hits from "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; zh-tw) AppleWebKit/418.9.1
 (KHTML, like Gecko) Safari/419.3"
17 hits from "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
15 hits from "Gigabot/2.0 (http://www.gigablast.com/spider.html)"
14 hits from "Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1)
Gecko/20061010 Firefox/2.0"
11 hits from "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)"
```

This parser is doing a minimum of data validation, which is usually JavaCC's strong point. But in this case, the default values of some of the data items make that hard to do. For example, the response length is usually a number, but if the response body is empty, as is the case in an HTTP 302 response, it can be a dash (-). Since our parser knows what each data item is we can still do data validation a bit further up the stack: check for valid dates, ensure that the response codes conform to those listed in the HTTP specification, and so forth.

We could preprocess our grammar with JJTree and get an AST, but that wouldn't buy us much because the AST is quite flat; just a root `Log` node with a bunch of `Entry` children and various data item grandchildren nonterminals.

On my workstation, the code that JavaCC generated from this grammar is able to tokenize, parse, and gather up the user agent frequencies on a log file at the rate of about 35 thousand lines per second. This may not mean a whole lot (since there's nothing to compare it to) but that's still a decent processing rate. Assuming a site is getting a hit each second, this parser should be able to process a full day's log file in under three seconds; not too shabby.

Lastly, we were only able to process these log files with JavaCC because the format was fixed and was clearly documented. Let this be an encouragement to all daemon writers—if you output machine readable logs, you'll make life much easier on those who wish to programmatically consume those logs.

Temperatures

Suppose we want to parse out a small list of data—a series of temperature readings, each on a separate line. Here's some sample input data:

```
# examples/little_grammars/temperatures/input.temps
```

```
1    21 ℉
2    -2 ℃
3    -4 ℃
4    -1 ℃
5    -8 ℃
6    23 ℉
7    -2 ℃
```

Note the wrinkle? The temperature readings all use the Unicode characters for the "degrees Celsius" (℃) or "degrees Fahrenheit" (℉) indicator. We will engage this enemy on his own ground; we'll embed the Java Unicode escaped versions of those characters in the lexical specification.

We'll use a private regular expression to avoid having to repeat the `DIGITS` definition and this time we'll treat spaces as part of the input (rather than skipping them as we've done in some previous examples).

Example 11.4. Lexical Specification for Unicode Temperatures

```
# examples/little_grammars/temperatures/temperatures.jj
```

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Temperatures)
5    import java.io.*;
6    public class Temperatures {}
7    PARSER_END(Temperatures)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws Exception {
10       Reader r = new FileReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(r);
12       TemperaturesTokenManager mgr = new TemperaturesTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15           System.out.println("Found a " + TemperaturesConstants.tokenImage
[t.kind] + ": " + t.image);
16       }
17     }
18   }
19   SKIP : {
20     "\n" | "\r" | "\r\n"
21   }
22   TOKEN : {
23     <FAHRENHEIT_TEMPERATURE : (["-"])? <DIGITS> " \u2109">
24     | <CELSIUS_TEMPERATURE : (["-"])? <DIGITS> " \u2103">
25     | <#DIGITS : ["0"-"9"](["0"-"9"])*>
26   }
```

Here's our tokenizer in action. Notice the warning about "Non-ASCII characters used" that we get when running JavaCC on the grammar. Using a `FileReader` takes care of that concern for us, since it knows how to decode a file—as long as it's either in the platform's default character encoding or we specify the encoding as we do here. Note how our `DIGITS` token definition keeps the other token definition free of duplicated expansions. Here's a sample run:

```
$ javacc temperatures.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file temperatures.jj . . .
Warning: Line 23, Column 3: Non-ASCII characters used in regular expression.
Please make sure you use the correct Reader when you create the parser, one that
can handle your character set.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
$ javac *.java
$ java -Dfile.encoding=UTF-8 TemperaturesTokenManager input.temps
Found a <FAHRENHEIT_TEMPERATURE>: 21 ℉
Found a <CELSIUS_TEMPERATURE>: -2 ℃
Found a <CELSIUS_TEMPERATURE>: -4 ℃
Found a <CELSIUS_TEMPERATURE>: -1 ℃
Found a <CELSIUS_TEMPERATURE>: -8 ℃
Found a <FAHRENHEIT_TEMPERATURE>: 23 ℉
Found a <CELSIUS_TEMPERATURE>: -2 ℃
```

So, we've shown that we can process Unicode characters as such. We could also take the easy way out, and use the Java Unicode escape sequences for the temperature readings:

 # examples/little_grammars/temperatures/input_java_escape.temps

```
1    21 \u2109
2    -2 \u2103
3    -4 \u2103
4    -1 \u2103
5    -8 \u2103
6    23 \u2109
7    -2 \u2103
```

If the input data took that form we'd need to use the `JAVA_UNICODE_ESCAPE` option to decode those characters, and we'd also need to use the `JavaCharStream` class that `JAVA_UNICODE_ESCAPE` causes JavaCC to generate:

Example 11.5. Escaped Temperatures Grammar

 # examples/little_grammars/temperatures/temperatures_java_escaped.jj

```
1    options {
2      BUILD_PARSER=false;
3      JAVA_UNICODE_ESCAPE=true;
4    }
5    PARSER_BEGIN(Temperatures)
6    import java.io.*;
7    public class Temperatures {}
8    PARSER_END(Temperatures)
9    TOKEN_MGR_DECLS: {
10     public static void main(String[] args) throws Exception {
11       Reader r = new FileReader(args[0]);
12       JavaCharStream jcs = new JavaCharStream(r);
13       TemperaturesTokenManager mgr = new TemperaturesTokenManager(jcs);
14       for (Token t = mgr.getNextToken(); t.kind != EOF;
15         t = mgr.getNextToken()) {
16           System.out.println("Found a " + TemperaturesConstants.tokenImage
[t.kind] + ": " + t.image);
17       }
18     }
19   }
20   SKIP : {
21     "\n" | "\r" | "\r\n"
22   }
23   TOKEN : {
24     <FAHRENHEIT_TEMPERATURE : (["-"])? <DIGITS> " \u2109">
25     | <CELSIUS_TEMPERATURE : (["-"])? <DIGITS> " \u2103">
26     | <#DIGITS : ["0"-"9"](["0"-"9"])*>
27   }
```

Generally, it's easier to work with escaped characters than with the actual characters. The actual characters look nicer though!

Notice that we're packing each entry into a single token. We could add a - token and move the `DIGITS` private regular expression into its own token definition to give us more granularity for each token, but combining each entry into one token may be more useful for this particular usage pattern. If there were a huge number of temperature readings, for example, and you wanted to minimize object creation you might choose this approach. As always, it's up to you to decide the best way to slice and dice your data.

A Little Logo

Logo[4] is a small functional programming language written by Danny Bobrow, Wally Feurzeig and Seymour Papert. Its best known use is to draw graphical images using a series of commands that move a drawing pen—usually called a "turtle"—around the screen. Here's an example of a Logo program:

```
# examples/little_grammars/logo/input.logo

1    FORWARD 20
2    RIGHT 120
3    FORWARD 20
4    RIGHT 120
5    FORWARD 20
```

This program draws a small equilateral triangle: the turtle moves forward 20 units, turns right 120 degrees, moves forward 20 units, turns right 120 degrees, and moves forward another 20 units to arrive back at the origin.

We've got two commands to handle: a `FORWARD` movement and a `RIGHT` turn. The magnitude of each movement, whether the degrees of the turn or the length of the movement, can be any positive integer. The commands are separated by newlines, and whitespace is only significant for separating the "real" data. This is a tiny subset of Logo, but it's enough to get started on a grammar.

Here's a lexical specification that tokenizes the input, along with a sample run. Notice that we specify `DIGITS` so that the first digit is between `1` and `9`; this disallows commands like `FORWARD 05`. We're also using the `Token.kind` as an index into the `LogoConstants.tokenImage` array to make the sample run output more readable:

[4]See http://en.wikipedia.org/wiki/Logo_programming_language for more information on Logo.

Example 11.6. Tokenizing Logo

 # examples/little_grammars/logo/logo_tokenizer.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Logo)
5    public class Logo {}
6    PARSER_END(Logo)
7    TOKEN_MGR_DECLS: {
8      public static void main(String[] args) throws Exception {
9        java.io.Reader r = new java.io.FileReader(args[0]);
10       SimpleCharStream scs = new SimpleCharStream(r);
11       LogoTokenManager mgr = new LogoTokenManager(scs);
12       for (Token t = mgr.getNextToken(); t.kind != EOF;
13         t = mgr.getNextToken()) {
14           System.out.println("Found a " + LogoConstants.tokenImage[t.kind] +
": " + t.image);
15       }
16     }
17   }
18   SKIP : {
19     " " | "\n" | "\r" | "\r\n"
20   }
21   TOKEN : {
22     <FORWARD : "FORWARD">
23     | <RIGHT : "RIGHT">
24     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
25   }
```

And here's how it looks processing our input file:

```
$ javacc logo_tokenizer.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file logo_tokenizer.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java LogoTokenManager input.logo
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
Found a "RIGHT": RIGHT
Found a <DIGITS>: 120
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
Found a "RIGHT": RIGHT
Found a <DIGITS>: 120
Found a "FORWARD": FORWARD
Found a <DIGITS>: 20
```

Let's add some structure with a syntactic specification. Our little program consists of a series of turns and moves, so let's just play "spot the noun" and add nonterminals for `Program`, `Turn`, and `Move`. We can also get rid of the `options` header and the `TOKEN_MGR_DECLS` section and fill in the `PARSER_BEGIN/PARSER_END` block. Notice that we're declaring the `main` method to throw `Exception`. This is not ideal error handling, but in this short program it saves some space. Besides, we aren't going to do much error handling—for this example, seeing a stacktrace on the console is good enough.

Example 11.7. Parsing Logo

 # examples/little_grammars/logo/logo_parser.jj

```
1    PARSER_BEGIN(Logo)
2    import java.io.*;
3    public class Logo {
4      public static void main(String[] args) throws Exception {
5        Reader r = new FileReader(args[0]);
6        Logo p = new Logo(r);
7        p.Program();
8     }
9    }
10   PARSER_END(Logo)
11   SKIP : {
12     " " | "\n" | "\r" | "\r\n"
13   }
14   TOKEN : {
15     <FORWARD : "FORWARD">
16     | <RIGHT : "RIGHT">
17     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
18   }
19   void Program() : {} {
20     (Move() | Turn())+
21   }
22   void Move() : {} {
23     <FORWARD> <DIGITS>
24   }
25   void Turn() : {} {
26     <RIGHT> <DIGITS>
27   }
```

Here's our parser in action:

```
$ javacc -JDK_VERSION=1.5 logo_parser.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file logo_parser.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java Logo input.logo
```

A successful parse! There's no output, of course, since we don't have any syntactic actions. Let's get some output by turning this into a JJTree grammar and dumping the AST. Here's the grammar and a sample run that uses the built-in `SimpleNode.dump` method:

Example 11.8. The Logo Tree

 # examples/little_grammars/logo/logo_tree.jjt

```
1    options {
2      VISITOR=true;
3      MULTI=true;
4    }
5    PARSER_BEGIN(Logo)
6    import java.io.*;
7    public class Logo {
8      public static void main(String[] args) throws Exception {
9        Reader sr = new FileReader(args[0]);
10       Logo logo = new Logo(sr);
11       ASTProgram p = logo.Program();
12       p.dump("");
13     }
14   }
15   PARSER_END(Logo)
16   SKIP : {
17     " " | "\n" | "\r" | "\r\n"
18   }
19   TOKEN : {
20     <FORWARD : "FORWARD">
21     | <RIGHT : "RIGHT">
22     | <DIGITS: (["1"-"9"])+ (["0"-"9"])*>
23   }
24   ASTProgram Program() : {} {
25     (Move() | Turn())+
26     {return jjtThis;}
27   }
28   void Move() : {} {
29     <FORWARD> <DIGITS>
30   }
31   void Turn() : {} {
32     <RIGHT> <DIGITS>
33   }
```

And the tree builder at work:

```
$ jjtree -JDK_VERSION=1.5 logo_tree.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file logo_tree.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "ASTProgram.java" does not exist.  Will create one.
File "ASTMove.java" does not exist.  Will create one.
File "ASTTurn.java" does not exist.  Will create one.
File "LogoTreeConstants.java" does not exist.  Will create one.
File "LogoVisitor.java" does not exist.  Will create one.
File "JJTLogoState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./logo_tree.jj
$ javacc -JDK_VERSION=1.5 logo_tree.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file logo_tree.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java Logo input.logo
Program
 Move
 Turn
 Move
 Turn
 Move
```

If you're so inclined, you could extend this grammar by adding new commands—`LEFT` and `BACKWARD` should be easy to start with. Logo also uses abbreviations, such as `FD` for `FORWARD`; you could implement that by changing the lexical specification. Or try adding another command, such as `REPEAT 2 [FORWARD 5 LEFT 2]`, which repeats the commands inside the brackets the specified number of times.

For the adventurous, try writing a small Swing user interface that allows the user to type in commands and send the turtle here and there. Accept the commands one line at a time, or provide a text area for small programs. Extra bonus points if you handle errors in such a way that the program prompts for a fix to a malformed command.

Postfix Expressions

Postfix notation, or Reverse Polish Notation[5], is an arithmetic notation invented by Charles Hamblin. In postfix notation, the operands precede the operator. For example, adding 2 and 2 would be written in postfix notation as `2 2 +`, as opposed to the more common infix notation of `2 + 2`. More operands and operators can be placed on the end of the expression, so that `2 5 + 3 -` would evaluate to `((2+5)-3)`, or `4`.

As before, we'll do the tokenizer first. With just two operators and some numbers, the lexical specification is straightforward. Here's the tokenizer:

Example 11.9. Tokenizing Postfix

 # examples/little_grammars/postfix/postfix_tokenizer.jj

```
1    options {
2      BUILD_PARSER=false;
3    }
4    PARSER_BEGIN(Postfix)
5    import java.io.*;
6    public class Postfix {}
7    PARSER_END(Postfix)
8    TOKEN_MGR_DECLS: {
9      public static void main(String[] args) throws Exception {
10       Reader r = new StringReader(args[0]);
11       SimpleCharStream scs = new SimpleCharStream(r);
12       PostfixTokenManager mgr = new PostfixTokenManager(scs);
13       for (Token t = mgr.getNextToken(); t.kind != EOF;
14         t = mgr.getNextToken()) {
15         System.out.println("Found a " + PostfixConstants.tokenImage[t.kind] +
": " + t.image);
16       }
17     }
18   }
19   SKIP : {
20     " "
21   }
22   TOKEN : {
23       <PLUS: "+">
24     | <MINUS: "-">
25     | <DIGITS: (["0"-"9"])+>
26   }
```

And a sample run:

```
$ javacc postfix_tokenizer.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file postfix_tokenizer.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
```

[5]http://en.wikipedia.org/wiki/Reverse_Polish_notation

```
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java PostfixTokenManager "2 3 + 4 -"
Found a <DIGITS>: 2
Found a <DIGITS>: 3
Found a "+": +
Found a <DIGITS>: 4
Found a "-": -
```

The syntactic specification is a bit more involved. Given the input `2 3 +`, we can say that the entire input string is an `Expression`. The `Expression` is composed of a `2`, which is a `Factor`, and `2 +`, which is a `Term`. But we want to allow for longer expressions, like `2 3 + 4 -`. To do this, we need to make `Term` recursive so that it can contain other `Expressions`. This introduces a choice conflict, since we don't know whether `2` is a `Factor` or the first token of a `Term`, like `2 +`. We can resolve this conflict by looking ahead an extra token. Here's the grammar:

Example 11.10. Parsing Postfix

 # examples/little_grammars/postfix/postfix_parser.jj

```
1    PARSER_BEGIN(Postfix)
2    import java.io.*;
3    public class Postfix {
4      public static void main(String[] args) throws Exception {
5        Reader r = new StringReader(args[0]);
6        Postfix p = new Postfix(r);
7        p.Expression();
8      }
9    }
10   PARSER_END(Postfix)
11   SKIP : {
12     " "
13   }
14   TOKEN : {
15       <PLUS: "+">
16     | <MINUS: "-">
17     | <DIGITS: (["0"-"9"])+>
18   }
19   void Expression() : {} {
20     Factor() (Term() Operator())*
21   }
22   void Term() : {} {
23     LOOKAHEAD(2) Expression() | Factor()
24   }
25   void Factor() : {} {
26     <DIGITS>
27   }
28   void Operator() : {} {
29     "+" | "-"
30   }
```

And a sample validation run:

```
$ javacc -JDK_VERSION=1.5 postfix_parser.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file postfix_parser.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java Postfix "2 3 + 4 -"
```

We're using right recursion in `Expression` and `Term`. The initial `Factor` "anchors" the parse tree and it grows down and to the right. Also, note that `2` is a valid input string since `Term() Operator()` is contained in a "zero or more" expansion.

We can take things a bit further by actually evaluating the expression. Let's do this using JJTree to construct an AST for the expression and then traverse the tree and do the calculation with a visitor.

Here's the new grammar. We're returning an `ASTExpression` from the root node and we've added code to the `main` method to allow a `CalcVisitor` to walk the tree:

Example 11.11. Evaluating Postfix—The Grammar

 # examples/little_grammars/postfix/postfix_tree.jjt

```
1    options {
2      MULTI=true;
3      VISITOR=true;
4    }
5    PARSER_BEGIN(Postfix)
6    import java.io.*;
7    public class Postfix {
8      public static void main(String[] args) throws Exception {
9        Reader sr = new StringReader(args[0]);
10       Postfix p = new Postfix(sr);
11       ASTExpression e = p.Expression();
12       e.dump("");
13       CalcVisitor c = new CalcVisitor();
14       e.jjtAccept(c, null);
15       System.out.println(c.stack.pop());
16     }
17   }
18   PARSER_END(Postfix)
19   SKIP : {" "}
20   TOKEN : {
21       <PLUS: "+">
22     | <MINUS: "-">
23     | <DIGITS: (["0"-"9"])+>
24   }
25   ASTExpression Expression() : {} {
26     Factor() (Term() Operator())*
27     {return jjtThis;}
28   }
29   void Term() : {} {
30     LOOKAHEAD(2) Expression() | Factor()
31   }
32   void Factor() : {Token t;} {
33     t=<DIGITS> {jjtThis.value = Integer.parseInt(t.image);}
34   }
35   void Operator() : {} {
36     "+" {jjtThis.plus = true;} | "-"
37   }
```

`ASTTerm` and `ASTExpression` are unchanged. But we added a new `value` field to the `ASTFactor` source file that JJTree generated:

 # examples/little_grammars/postfix/ASTFactor.java

```
1    /* Generated By:JJTree: Do not edit this line. ASTFactor.java Version 4.1 */
2    /*
JavaCCOptions:MULTI=true,NODE_USES_PARSER=false,VISITOR=true,TRACK_TOKENS=false,N
ODE_PREFIX=AST,NODE_EXTENDS=,NODE_FACTORY=,SUPPORT_CLASS_VISIBILITY_PUBLIC=true
*/
3    public
4    class ASTFactor extends SimpleNode {
5
6      public int value;
7
```

```
8       public ASTFactor(int id) {
9         super(id);
10      }
11
12      public ASTFactor(Postfix p, int id) {
13        super(p, id);
14      }
15
16
17      /** Accept the visitor. **/
18      public Object jjtAccept(PostfixVisitor visitor, Object data) {
19        return visitor.visit(this, data);
20      }
21    }
22    /* JavaCC - OriginalChecksum=3794971197935ca3815e2c3d1f21866a (do not edit
this line) */
```

and a boolean `plus` field to the `ASTOperator` source code:

 # examples/little_grammars/postfix/ASTOperator.java

```
1     /* Generated By:JJTree: Do not edit this line. ASTOperator.java Version 4.1
*/
2     /*
JavaCCOptions:MULTI=true,NODE_USES_PARSER=false,VISITOR=true,TRACK_TOKENS=false,N
ODE_PREFIX=AST,NODE_EXTENDS=,NODE_FACTORY=,SUPPORT_CLASS_VISIBILITY_PUBLIC=true
*/
3     public
4     class ASTOperator extends SimpleNode {
5
6       public boolean plus;
7
8       public ASTOperator(int id) {
9         super(id);
10      }
11
12      public ASTOperator(Postfix p, int id) {
13        super(p, id);
14      }
15
16
17      /** Accept the visitor. **/
18      public Object jjtAccept(PostfixVisitor visitor, Object data) {
19        return visitor.visit(this, data);
20      }
21    }
22    /* JavaCC - OriginalChecksum=49078abc07c702598a13c707f704e89a (do not edit
this line) */
```

The `CalcVisitor` does the actual calculation. As it visits each `Factor`, it pushes the value onto an internal stack. As it visits each `Operator`, it pops the top two values off the stack, applies the operator, and pushes the result onto the stack:

 # examples/little_grammars/postfix/CalcVisitor.java

```
1     import java.util.*;
2     public class CalcVisitor extends Adapter {
3       public Stack<Integer> stack = new Stack<Integer>();
4       public Object visit(ASTFactor f, Object d) {
5         stack.push(f.value);
6         System.out.println("Pushed on " + f.value);
7         return super.visit(f, d);
8       }
9       public Object visit(ASTOperator o, Object d) {
10        System.out.println("Got an operator: " + (o.plus ? "+" : "-"));
11        int op2 = stack.pop();
12        int op1 = stack.pop();
13        if (o.plus) {
14          stack.push(op1+op2);
15        } else {
16          stack.push(op1-op2);
```

```
17       }
18       System.out.println("Stack top is now " + stack.peek());
19       return super.visit(o, d);
20     }
21   }
```

Here's the `Adapter` that `CalcVisitor` extends. It provides stubbed out versions of all the callbacks so that `CalcVisitor` need only override the two callbacks it needs to receive:

 # examples/little_grammars/postfix/Adapter.java

```
1    public class Adapter implements PostfixVisitor {
2      public Object visit(SimpleNode node, Object data) {
3        return node.childrenAccept(this, data);
4      }
5      public Object visit(ASTExpression node, Object data){
6        return node.childrenAccept(this, data);
7      }
8      public Object visit(ASTFactor node, Object data){
9        return node.childrenAccept(this, data);
10     }
11     public Object visit(ASTTerm node, Object data){
12       return node.childrenAccept(this, data);
13     }
14     public Object visit(ASTOperator node, Object data){
15       return node.childrenAccept(this, data);
16     }
17   }
```

Finally, we put it all together and feed in some sample data:

```
$ jjtree -JDK_VERSION=1.5 postfix_tree.jjt
Java Compiler Compiler Version 4.2 (Tree Builder)
(type "jjtree" with no arguments for help)
Reading from file postfix_tree.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
File "ASTExpression.java" does not exist.  Will create one.
File "PostfixTreeConstants.java" does not exist.  Will create one.
File "PostfixVisitor.java" does not exist.  Will create one.
File "JJTPostfixState.java" does not exist.  Will create one.
Annotated grammar generated successfully in ./postfix_tree.jj
$ javacc -JDK_VERSION=1.5 postfix_tree.jj
Java Compiler Compiler Version 4.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file postfix_tree.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
$ javac *.java
$ java Postfix "2 3 + 4 -"
Expression
 Factor
 Term
  Expression
   Factor
 Operator
 Term
  Expression
   Factor
 Operator
Pushed on 2
Pushed on 3
Got an operator: +
Stack top is now 5
Pushed on 4
Got an operator: -
```

```
Stack top is now 1
1
```

Notice that the `2` and `3` are pushed on the stack first. Then the `+` is processed, producing a `5`. The `5` goes back on the stack, it's topped by the `4`, and then the `-` reduces the stack to a `1`. That's the value we end with.

There are many ways this little grammar could be enhanced. For example, it should fail to parse invalid expressions like `2 3 + -`; instead, it just parses out the first `Expression` that it finds and then exits without checking for an end-of-file marker. Also, it'd be nice to have the `*` and `/` operators available.

Summary

JavaCC is usually presented in the context of complicated parsing tasks, and that's no accident; it's well suited to parsing programming languages and tricky, deeply nested data structures. Hopefully this chapter has shown that JavaCC is good for smaller jobs as well. Give it a try for your next small parsing job; I think you'll be pleased with JavaCC's performance and the ease with which you can build a small parser.

References

[1] *Modern Compiler Design*. Dick Grune, Henri Bal, Ceriel Jacobs, Koen Langendoen. Wiley and Sons, 2000.

[2] *Mastering Regular Expressions, 3rd Edition*. Jeffrey Friedl. O'Reilly and Associates, 2006.

[3] *Cascading Style Sheets: The Definitive Guide, Second Edition*. Eric Meyer. O'Reilly & Associates, 2004.

[4] *Design Patterns*. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley, 1995.

[5] *Java Development with Ant*. Steve Loughran and Eric Hatcher. Manning, 2002.

[6] *The JavaCC Story*. Sriram Sankar. MetaMata, 1997.

[7] *XPath and XPointer*. John Simpson. O'Reilly & Associates, 2002.

Appendix A. JavaCC Directory Layout and Building JavaCC

This appendix will give you an overview of the contents of the JavaCC binary distribution directory and the JavaCC CVS repository directory tree. Then we'll look at what's involved in customizing JavaCC.

Binary Distribution

The following items appear in the binary distribution's top-level directory:

- `javacc-4.2/LICENSE`: The JavaCC license
- `javacc-4.2/bin/`: Contains both Windows and Unix scripts for running JavaCC, JJTree and JJDoc. Also contains the classfiles for all of JavaCC in the `javacc.jar` file in the `lib` subdirectory.
- `javacc-4.2/doc/`: Contains the HTML documentation for JavaCC, JJTree and JJDoc. This is the same as the documentation on the JavaCC web site.
- `javacc-4.2/examples/`: Contains numerous JavaCC and JJTree grammar examples.

Source Distribution

The following items appear in the JavaCC CVS repository's top-level directory:

- Several files appear in the top-level directory: Ant and Maven build scripts (`build.xml` and `pom.xml`), the JavaCC license (`LICENSE`), a shell script for creating a release (`makedist`), and a file with instructions on building the JavaCC source code (`README`).
- `javacc/bin/`: Contains both Windows and Unix scripts for running JavaCC, JJTree and JJDoc.
- `javacc/bootstrap/`: Contains a `javacc.jar` file used to bootstrap JavaCC. In other words, since the JavaCC grammar format is defined by a JavaCC grammar, JavaCC must first generate a parser that's capable of parsing its own grammar file. This chicken and egg situation is resolved by placing a prebuilt JavaCC jar file in this directory, and the classes in that jar file are used to parse the JavaCC grammar file and generate a parser.
- `javacc/examples/`: Contains numerous JavaCC and JJTree grammar examples.
- `javacc/lib/`: Contains a JUnit jar file which is used for running the JavaCC unit test suite.

- `javacc/src/`: Contains the JavaCC source code. There are several top-level classes (`javacc`, `jjtree`, and `jjdoc`) that serve as entry points into each of those application. The code for each utility is contained in respective subdirectories: `org/javacc/jjdoc/`, `org/javacc/parser/` (for JavaCC), and `org/javacc/jjtree/`.
- `javacc/doc/`: Contains the HTML documentation for JavaCC, JJTree and JJDoc.

Building JavaCC From Source

Building JavaCC from source requires a few fairly standard Java tools:

- A CVS client. Any reasonably recent version will do; I use 1.12.13.
- A JDK installation of version 1.4 or higher.
- The Ant build utility. Any recent release will do here as well; I use Ant 1.7.1. You could manage to compile JavaCC without using Ant by gathering up lists of filenames and feeding them to `javac`, but the JavaCC developers use Ant and the scripts are all there for you to use.

To get started, check out the JavaCC code using a CVS client. Note that you need an account on `java.net` to do this:

```
$  cvs -d :pserver:your_account_name@cvs.dev.java.net:/cvs checkout javacc
cvs checkout: Updating javacc
U javacc/LICENSE
U javacc/README
[... lots more output ...]
```

Next, move down into the `javacc` directory that you just checked out and run `ant jar` to compile JavaCC and build a jar file:

```
$ ant jar
Buildfile: build.xml

generated-files:

parser-files-init:

parser-files:
    [java] Java Compiler Compiler Version 4.1d1 (Parser Generator)
    [java] (type "javacc" with no arguments for help)
    [java] Reading from file /home/tom/tmp/foo/javacc/src/org/javacc/parser/
JavaCC.jj . . .
[... lots more output ...]
```

This produces a jar file in the `bin/lib` directory which you can then use to process grammars. Note that you can use the `bin/javacc` script to run JavaCC and it will use the `javacc.jar` file in the `lib/` directory as opposed to any other JavaCC jar files that you may have on your `CLASSPATH`:

```
$ cat test.jj
PARSER_BEGIN(Test)
public class Test {}
PARSER_END(Test)
void Start() : {} {
"HELLO"
}

$ bin/javacc test.jj
```

```
Java Compiler Compiler Version 4.1d1 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file test.jj . . .
Parser generated successfully.
```

Success! You've built JavaCC from source and processed a grammar with it.

Appendix B. Options

What follows is a listing of the options for each major JavaCC application: JavaCC, JJTree, JJDoc, and JTB. The options for each of these applications are covered in much more detail in the chapters, but this can serve as a quick reference.

JavaCC

```
$ javacc
Java Compiler Compiler Version 4.2 (Parser Generator)

Usage:
    javacc option-settings inputfile

"option-settings" is a sequence of settings separated by spaces.
Each option setting must be of one of the following forms:

    -optionname=value (e.g., -STATIC=false)
    -optionname:value (e.g., -STATIC:false)
    -optionname       (equivalent to -optionname=true.  e.g., -STATIC)
    -NOoptionname     (equivalent to -optionname=false. e.g., -NOSTATIC)

Option settings are not case-sensitive, so one can say "-nOsTaTiC" instead
of "-NOSTATIC".  Option values must be appropriate for the corresponding
option, and must be either an integer, a boolean, or a string value.

The integer valued options are:

    LOOKAHEAD              (default 1)
    CHOICE_AMBIGUITY_CHECK (default 2)
    OTHER_AMBIGUITY_CHECK  (default 1)

The boolean valued options are:

    STATIC                 (default true)
    SUPPORT_CLASS_VISIBILITY_PUBLIC (default true)
    DEBUG_PARSER           (default false)
    DEBUG_LOOKAHEAD        (default false)
    DEBUG_TOKEN_MANAGER    (default false)
    ERROR_REPORTING        (default true)
    JAVA_UNICODE_ESCAPE    (default false)
    UNICODE_INPUT          (default false)
    IGNORE_CASE            (default false)
    COMMON_TOKEN_ACTION    (default false)
    USER_TOKEN_MANAGER     (default false)
    USER_CHAR_STREAM       (default false)
    BUILD_PARSER           (default true)
    BUILD_TOKEN_MANAGER    (default true)
    TOKEN_MANAGER_USES_PARSER (default false)
    SANITY_CHECK           (default true)
    FORCE_LA_CHECK         (default false)
    CACHE_TOKENS           (default false)
    KEEP_LINE_COLUMN       (default true)

The string valued options are:

    OUTPUT_DIRECTORY       (default Current Directory)
    TOKEN_EXTENDS          (default java.lang.Object)
    TOKEN_FACTORY          (default none)
    JDK_VERSION            (default 1.5)
    GRAMMAR_ENCODING       (defaults to platform file encoding)

EXAMPLE:
    javacc -STATIC=false -LOOKAHEAD:2 -debug_parser mygrammar.jj
```

JJTree

```
$ jjtree
Java Compiler Compiler Version 4.2 (Tree Builder)

Usage:
    jjtree option-settings inputfile

"option-settings" is a sequence of settings separated by spaces.
Each option setting must be of one of the following forms:

    -optionname=value (e.g., -STATIC=false)
    -optionname:value (e.g., -STATIC:false)
    -optionname       (equivalent to -optionname=true.  e.g., -STATIC)
    -NOoptionname     (equivalent to -optionname=false. e.g., -NOSTATIC)

Option settings are not case-sensitive, so one can say "-nOsTaTiC" instead
of "-NOSTATIC".  Option values must be appropriate for the corresponding
option, and must be either an integer or a string value.

The boolean valued options are:

    STATIC                    (default true)
    MULTI                     (default false)
    NODE_DEFAULT_VOID         (default false)
    NODE_SCOPE_HOOK           (default false)
    NODE_USES_PARSER          (default false)
    BUILD_NODE_FILES          (default true)
    TRACK_TOKENS              (default false)
    VISITOR                   (default false)

The string valued options are:

    JDK_VERSION               (default "1.5")
    NODE_CLASS                (default "")
    NODE_PREFIX               (default "AST")
    NODE_PACKAGE              (default "")
    NODE_EXTENDS              (default "")
    NODE_FACTORY              (default "")
    OUTPUT_FILE               (default remove input file suffix, add .jj)
    OUTPUT_DIRECTORY          (default "")
    JJTREE_OUTPUT_DIRECTORY   (default value of OUTPUT_DIRECTORY option)
    VISITOR_DATA_TYPE         (default "")
    VISITOR_RETURN_TYPE       (default "Object")
    VISITOR_EXCEPTION         (default "")

JJTree also accepts JavaCC options, which it inserts into the generated file.

EXAMPLES:
    jjtree -STATIC=false mygrammar.jjt

ABOUT JJTree:
    JJTree is a preprocessor for JavaCC that inserts actions into a
    JavaCC grammar to build parse trees for the input.

    For more information, see the online JJTree documentation at
    https://javacc.dev.java.net/doc/JJTree.html
```

JJDoc

```
$ jjdoc
Java Compiler Compiler Version 4.2 (Documentation Generator Version 0.1.4)

    jjdoc option-settings - (to read from standard input)
OR
    jjdoc option-settings inputfile (to read from a file)
```

```
WHERE
    "option-settings" is a sequence of settings separated by spaces.

Each option setting must be of one of the following forms:

    -optionname=value (e.g., -TEXT=false)
    -optionname:value (e.g., -TEXT:false)
    -optionname       (equivalent to -optionname=true.  e.g., -TEXT)
    -NOoptionname     (equivalent to -optionname=false. e.g., -NOTEXT)

Option settings are not case-sensitive, so one can say "-nOtExT" instead
of "-NOTEXT".  Option values must be appropriate for the corresponding
option, and must be either an integer, boolean or string value.

The string valued options are:

    OUTPUT_FILE
    CSS

The boolean valued options are:

    ONE_TABLE              (default true)
    TEXT                   (default false)


EXAMPLES:
    jjdoc -ONE_TABLE=false mygrammar.jj
    jjdoc - < mygrammar.jj

ABOUT JJDoc:
    JJDoc generates JavaDoc documentation from JavaCC grammar files.

    For more information, see the online JJDoc documentation at
    https://javacc.dev.java.net/doc/JJDoc.html
```

JTB

This option listing was produced by running JTB 1.3.2.

```
$ java -jar /usr/local/jtb/jtb132.jar
JTB version 1.3.2

Usage: jtb [OPTIONS] [inputfile]

Standard options:
  -h          Displays this help message.
  -o NAME     Uses NAME as the filename for the annotated output grammar.
  -np NAME    Uses NAME as the package for the syntax tree nodes.
  -vp NAME    Uses NAME as the package for the default Visitor class.
  -p NAME     "-p pkg" is short for "-np pkg.syntaxtree -vp pkg.visitor"
  -si         Read from standard input rather than a file.
  -w          Do not overwrite existing files.
  -e          Suppress JTB semantic error checking.
  -jd         Generate JavaDoc-friendly comments in the nodes and visitor.
  -f          Use descriptive node class field names.
  -ns NAME    Uses NAME as the class which all node classes will extend.
  -pp         Generate parent pointers in all node classes.
  -tk         Generate special tokens into the tree.

Toolkit options:
  -scheme     Generate: (1) Scheme records representing the grammar.
                        (2) A Scheme tree building visitor.
  -printer    Generate a syntax tree dumping visitor.
```

Index

Symbols

A

B

C

D

E

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

By the author of "Generating Parsers With JavaCC":

"PMD Applied" is for developers and project managers who use the Java programming language and want to ensure their project's source code is *clean* and *consistent*. Developers working on new projects can use the PMD tool to prevent new problems polluting their code. Those who maintain and enhance legacy Java projects will also benefit from PMD's ability to sniff out problems in existing code. Both groups will benefit from this comprehensive reference guide—covering all PMD's existing capabilities and internal workings.

Available for $19.95, plus shipping.

To order, go to www.centennialbooksonline.com

What experts say about PMD:

"PMD is very customizable and ... is strongest at finding organizational and accident-waiting-to-happen bugs." Joe Walker; JAVA WORLD; November 2003

"PMD is among ... some of the most promising new products on the open source landscape." John Ravella & Rosalyn Lum; SOFTWARE DEVELOPMENT MAGAZINE; March 2004

"PMD is a cheap, easy, fun way to improve your programs. If you haven't used PMD before, you owe it to yourself and your customers to try it." Elliotte Rusty Harold; IBM DEVELOPERWORKS; January 2005