The JavaCC FAQ

Maintained by Theodore S. Norvell
Computer and Electrical Engineering
Memorial University of Newfoundland
Email: theo at engr.mun.ca

Typeset on July 14, 2003.

Contents

1 General Information on JavaCC and Parsing **5**

1.1 Latest changes to the FAQ . . . 5
1.2 What is JavaCC? . . . 5
1.3 Could you explain that in more detail? . . . 6
1.4 What does JavaCC not do? . . . 6
1.5 What can JavaCC be used for? . . . 7
1.6 Where can I get JavaCC? . . . 8
1.7 What legal restrictions are there on JavaCC? . . . 8
1.8 Is the source code for JavaCC publicly available? . . . 8
1.9 Is there any documentation? . . . 8
1.10 Are there books, articles, or tutorials on JavaCC? . . . 9
1.11 Are there books or tutorials on parsing theory? . . . 9
1.12 Is there a newsgroup or mailing list? . . . 10
1.13 Should I send my question to the newsgroup or mailing list? . 10
1.14 Who wrote JavaCC and who maintains it? . . . 10

2 Common Issues **12**

2.1 What files does JavaCC produce? . . . 12
2.2 Can I modify the generated files? . . . 13
2.3 I changed option x; why am I having trouble? . . . 14
2.4 How do I put the generated classes in a package? . . . 14
2.5 How do I use JavaCC with ANT? . . . 14

3 The Token Manager **16**

3.1 What is a token manager? . . . 16
3.2 How do I read from a string instead of a file? . . . 17
3.3 What if more than one regular expression matches a prefix of the remaining input? . . . 17

3.4 What if no regular expression matches a prefix of the remaining input? . . . 19
3.5 How do I make a character sequence match more than one token kind? . . . 19
3.6 How do I match any character? . . . 20
3.7 How do I match exactly n repetitions of a regular expression? 21
3.8 What are **TOKEN**, **SKIP**, and **SPECIAL_TOKEN**? . . . 21
3.9 What are lexical states all about? . . . 22
3.10 Can the parser force a switch to a new lexical state? . . . 24
3.11 Is there a way to make SwitchTo safer? . . . 24
3.12 How do I match an arbitrarily long sequence of characters? . . 25
3.13 What is MORE? . . . 26
3.14 Why do the example Java and C++ parsers report an error when the last line of a file is a single line comment? . . . 27
3.15 What is a lexical action? . . . 28
3.16 How do I tokenize nested comments? . . . 29
3.17 What is a common token action? . . . 30
3.18 How do I throw a ParseException instead of a TokenMgrError? 30
3.19 Why are line and column numbers not recorded? . . . 30

4 The Parser and Lookahead 32
4.1 Where should I draw the line between lexical analysis and parsing? . . . 32
4.2 What is recursive descent parsing? . . . 32
4.3 What is left-recursion and why can't I use it? . . . 33
4.4 How do I match an empty sequence of tokens? . . . 34
4.5 What is "lookahead"? . . . 34
4.6 I get a message saying "Warning: Choice Conflict ... "; what should I do? . . . 35
4.7 I added a LOOKAHEAD specification and the warning went away; does that mean I fixed the problem? . . . 39
4.8 Are semantic actions executed during syntactic lookahead? . . 39
4.9 Are nested syntactic lookahead specifications evaluated during syntactic lookahead? . . . 39
4.10 Are parameters passed during syntactic lookahead? . . . 42
4.11 Are semantic actions executed during syntactic lookahead? . . 42
4.12 Is semantic lookahead evaluated during syntactic lookahead? . 42

4.13 Can local variables (including parameters) be used in semantic lookahead? 42
4.14 How does JavaCC differ from standard LL(1) parsing? 43
4.15 How do I communicate from the parser to the token manager? 43
4.16 How do I communicate from the token manager to the parser? 44
4.17 What does it mean to put a regular expression within a BNF production? 45
4.18 When should regular expressions be put directly into a BNF production? 46
4.19 How do I parse a sequence without allowing duplications? . . . 48
4.20 How do I deal with keywords that aren't reserved? 49
4.21 There's an error in the input, so why doesn't my parser throw a ParseException? 52

5 Semantic Actions **53**
5.1 I've written/found a parser, but it doesn't do anything? 53
5.2 How do I capture and traverse a sequence of tokens? 53

6 JJTree and JTB **56**
6.1 What are JJTree and JTB? 56
6.2 Where can I find JJTree? 56
6.3 Where can I find JTB? 56

7 Applications of JavaCC **57**
7.1 Where can I find a parser for x? 57
7.2 How do I parse arithmetic expressions? 57
7.3 I'm writing a programming language interpreter; how do I deal with loops? 57

8 Comparing JavaCC with other tools **59**
8.1 Since $LL(1) \subset LALR(1)$, wouldn't a tool based on LALR parsing be better? 59
8.2 How does JavaCC compare with Lex and Flex? 60
8.3 How does JavaCC compare with other Yacc and Bison? 60

Acknowledgments: Your maintainer would like to thank the following for help with the FAQ: Ken Beesley, Paul Cager, Tom Davies, Brian Goetz, Tony LaPaso, Eric Nickell, Phil Robare, David Rosenstrauch, and Michael Welle.

The latest copy of this FAQ can be found at The JavaCC FAQ[1].
In citing or linking to this FAQ, please use the following URI:

http://www.engr.mun.ca/˜theo/JavaCC-FAQ

[1] http://www.engr.mun.ca/∼theo/JavaCC-FAQ/

Chapter 1

General Information on JavaCC and Parsing

"DRAGONS DREAD
GO BACK TO BED!!"
Sheree Fitch, *Sleeping Dragons All Around.*

1.1 Latest changes to the FAQ

- Added question about (~[])* .
- Use the new mail list now. Please.

1.2 What is JavaCC?

JavaCC stands for "the Java Compiler Compiler"; it is a parser generator and lexical analyzer generator. JavaCC will read a description of a language and generate code, written in Java, that will read and analyze that language. JavaCC is particularly useful when you have to write code to deal with an input language has a complex structure; in that case, hand-crafting an input module without the help of a parser generator can be a difficult job.

This technology originated to make programming language implementation easier —hence the term "compiler compiler"— but make no mistake that JavaCC is of use only to programming language implementors.

1.3 Could you explain that in more detail?

Figures 1.1 and 1.2 show the relationship between a JavaCC generated lexical analyzer (called a "token manager" in JavaCC parlance) and a JavaCC generated parser. The figures show C as the input language. But JavaCC can handle any language —and not only programming languages— if you can describe the rules of the language to JavaCC.

The token manager reads in a sequence of characters and produces a sequence of objects called "tokens". The rules used to break the sequence of characters into a sequence of tokens obviously depend on the language; they are supplied by the programmer as a collection of "regular expressions".

The parser consumes the sequence of tokens, analyses its structure, and produces ... . Well what the parser produces is up to you; JavaCC is completely flexible in this regard[1]. The figure shows an "abstract syntax tree", but you might want to produce, say, a number (if you are writing a calculator), a file of assembly language (if you were writing a one-pass compiler), a modified sequence of characters (if you were writing a text processing application), and so on. The programmer supplies a collection of "Extended BNF production rules"; JavaCC uses these productions to generate the parser as a Java class. These production rules can be annotated with snippets of Java code, which is how the programmer tells the parser what to produce.

1.4 What does JavaCC not do?

JavaCC does not automate the building of trees (or any other specific parser output), although there are at least two tree building tools JJTree and JTB (see Chapter 6.) based on JavaCC, and building trees "by hand"with a JavaCC based parser is easy.

JavaCC does not build symbol-tables, although if you want a symbol table for a language, then a JavaCC based parser may provide a good framework.

JavaCC does not generate output languages. However once you have a tree, it is easy to generate string output from it.

[1]Another way of looking at it is that JavaCC is of little help in this regard. However, if you want to produce trees there are two tools, based on JavaCC, that are less flexible and more helpful, these are JJTree and JTB. See Chapter 6 .

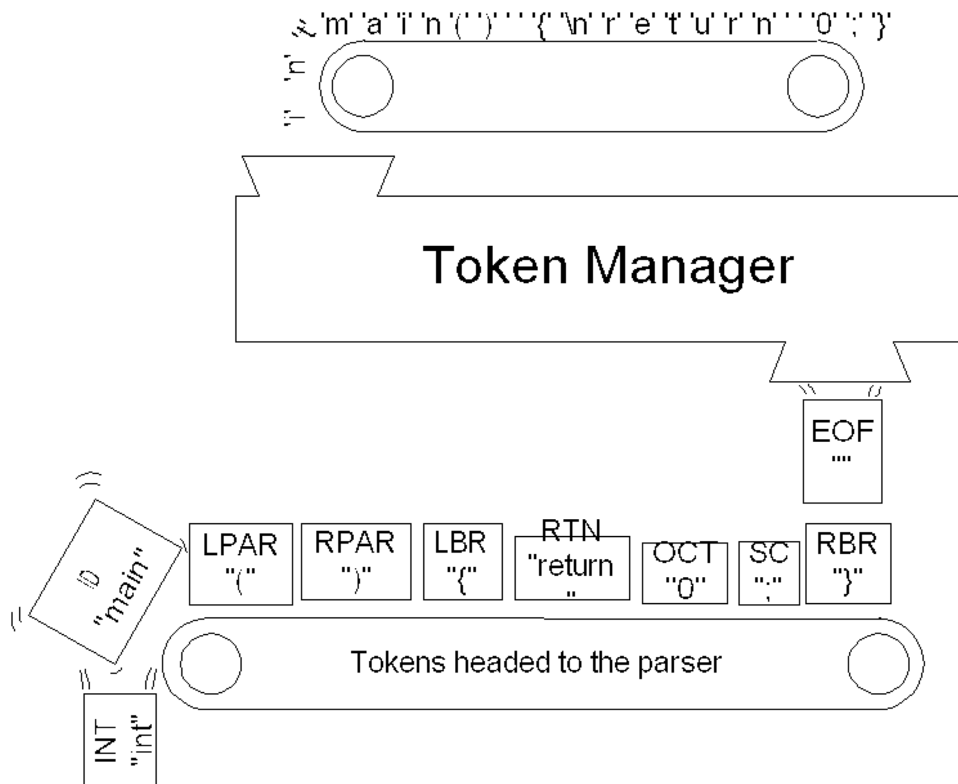

Figure 1.1: The token manager converts a sequence of characters to a sequence of Token objects.

1.5 What can JavaCC be used for?

JavaCC has been used to create parsers for: RTF, Visual Basic, Python, Rational Rose mdl files, XML, XML DTDs, HTML, C, C++, Java, JavaScript, Oberon, SQL, VHDL, VRML, ASN1, email headers, and lots of proprietary languages. It also get used for configuration file readers, calculators, and on and on.

1.6 Where can I get JavaCC?

JavaCC is available from experimentalstuff.com[2] or java.net[3] as a free download.

1.7 What legal restrictions are there on JavaCC?

There are essentially no restrictions on the use of JavaCC. In particular you may use the Java files that JavaCC produces in any way, including incorporating them into a product that you sell.

JavaCC is freely redistributable under the its open source licence. See the licence for details.

1.8 Is the source code for JavaCC publicly available?

Yes. As of June 2003, JavaCC is open source. The source code can be found at java.net[4].

1.9 Is there any documentation?

Yes. It is available online at experimentalstuff.com[5].

The on-line documentation is currently a bit out of date. *You should also read the release notes that come with the JavaCC download.*

The documentation is rather terse and is much easier to read if you already know a bit about parsing theory. Nevertheless, the documentation is an indispensable resource that is in no way superceded by this FAQ.

It used to be possible to download the documentation in a big ZIP file. At the moment your maintainer does not know where the documentation can be downloaded from, but you can currently view it on-line.

[2] http://www.experimentalstuff.com/Technologies/JavaCC/
[3] http://javacc.dev.java.net/
[4] http://javacc.dev.java.net/
[5] http://www.experimentalstuff.com/Technologies/JavaCC/docindex.html

1.10 Are there books, articles, or tutorials on JavaCC?

At the moment there are no books on JavaCC.

Tutorials and articles:

- There are mini-tutorials in the documentation. (See Question 1.9, "Is there any documentation?".)
- A draft tutorial by Theodore Norvell[6]
- A lexer tutorial by Sreeni Viswanadha.[7]
- A tutorial that involves JJTree.[8]
- A related but different tutorial that involves JJTree.[9]
- A short tutorial on JJTree by Jocelyn Paine.[10]
- A couple of articles have been published in JavaWorld. [11]

1.11 Are there books or tutorials on parsing theory?

Yes many. Most text-books on compiler technology contain more than enough background on parsing theory. Here are some suggestions

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1985.
- Charles N. Fischer and Richard J. Leblanc, Jr., *Crafting a Compiler With C*, Addison-Wesley, 1991.

---

[6] http://www.engr.mun.ca/∼theo/JavaCC-Tutorial/.

[7] http://www.cs.albany.edu/∼sreeni/JavaCC/lexertips.html.

[8] http://www-106.ibm.com/developerworks/xml/library/x-javacc1/.

[9] http://www.fatdog.com/Extreme.html.

[10] http://www.j-paine.org/jjtree.html.

[11] At http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html and http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html.

1.12 Is there a newsgroup or mailing list?

comp.compilers.tools.javacc is a usenet newsgroup for discussing the JavaCC and related technologies, like JJTree and JTB.

There is also a mail-list at java.net[12]. To sign up for this list you must first register as a java.net user, second join the JavaCC project, and then request to be added to the "users" mail-list.

The mailing list and comp.compilers.tools.javacc are not currently gatewayed.

1.13 Should I send my question to the newsgroup or mailing list?

Yes, but only if your question relates in some way to JavaCC, JJTree, or JTB.

The newsgroup and mailing list are not suitable fora for discussing the Java programming language or javac, which is Sun's Java compiler, or any other topic that does not relate directly to the Java Compiler Compiler.

Questions on parsing theory or parser generation in general might be better addressed to comp.compilers.

Questions directly answered in the FAQ need not be asked again in the newgroup or mailing list.

1.14 Who wrote JavaCC and who maintains it?

JavaCC was created by Sreeni Viswanadha and Sriram Sankar when they worked for Sun. They are contiuing to improve it.

Since JavaCC is now open source it is being maintained by its developer community. Luckly this includes the original authors.

[12]http://javacc.dev.java.net/

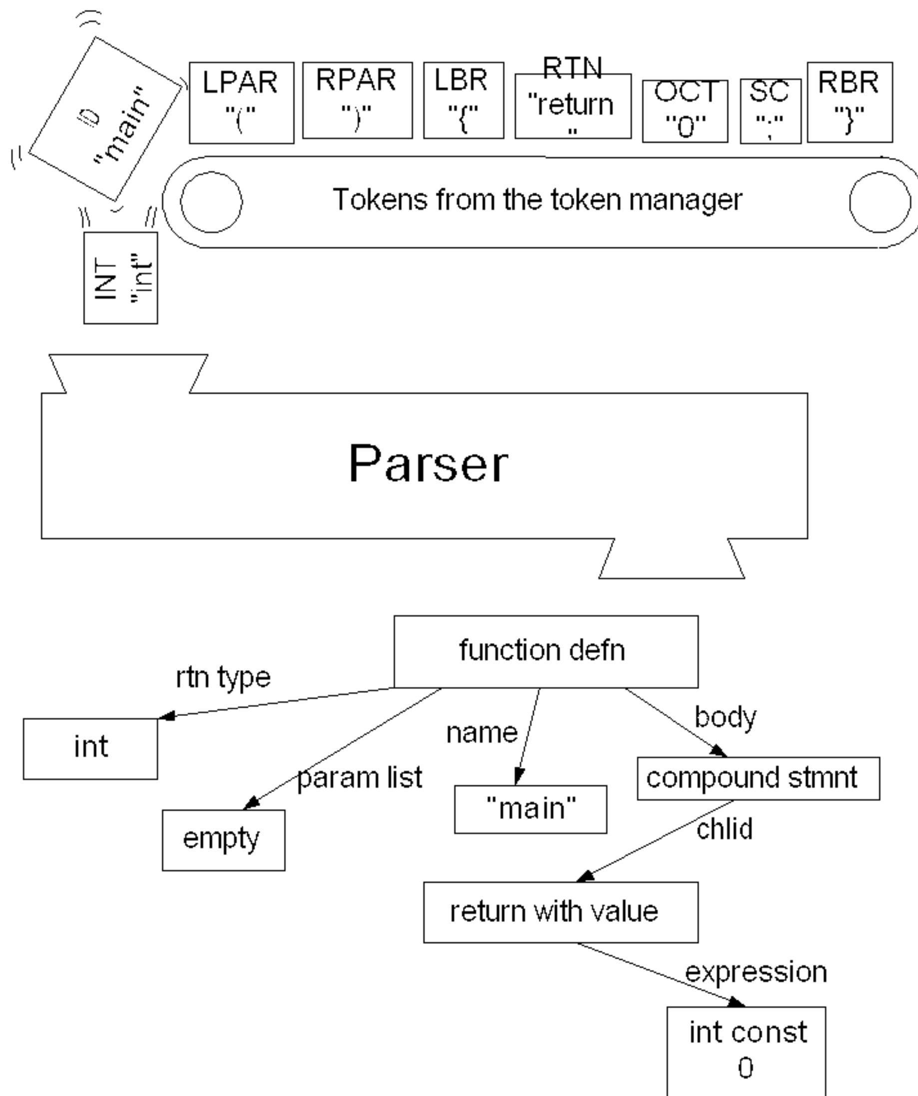

Figure 1.2: The parser analyzes the sequence of tokens.

Chapter 2

Common Issues

2.1 What files does JavaCC produce?

JavaCC is a program generator. It reads a .jj file and, if that .jj file is error free, produces a number of Java source files. With the default options, it generates the following files:

- Boiler-plate files
  - `SimpleCharStream.java` — represent the stream of input characters.
  - `Token.java` — represents a single input token
  - `TokenMgrError.java` — an error thrown from the token manager.
  - `ParseException.java` — an exception indicating that the input did not conform to the parser's grammar.
- Custom files (*XXX* is whatever name you choose).
  - *XXX*`.java` — the parser class
  - *XXX*`TokenManager.java` — the token manager class.
  - *XXX*`Constants.java` — an interface associating token classes with symbolic names.

If you use the option JAVA_UNICODE_ESCAPE then `SimpleCharStream.java` will not be produced, but rather `JavaCharStream.java`. (Prior to version 2.1, one of four possible files was generated: `ASCII_CharStream.java` `ASCII_UCodeESC_CharStream.java`, `UCode_CharStream.java`, or `UCode_UCodeESC_CharStream.java`.)

If you use the option USER_CHAR_STREAM, then `CharStream.java` (an interface) will be produced instead of the class `SimpleCharStream.java`. Similarly the option USER_TOKEN_MANAGER will cause the generation of an interface `TokenManager.java`, rather than a concrete token manager.

The boiler-plate files will only be produced if they don't already exist. There are two important consequences: First, you should delete them prior to running JavaCC, if you make any changes that might require changes to these files. (See Question 2.3, "I changed option x; why am I having trouble?".) Second, if you really want to, you can modify these files and be sure that JavaCC won't overwrite them. (See Question 2.2, "Can I modify the generated files?".)

2.2 Can I modify the generated files?

Modifying any generated files should be generally avoided, since someday you will likely want to regenerate them and then you'll have to re-modify them.

That said, modifying the `Token.java`, `ParserException.java` and `TokenManagerError.java` files is a fairly safe thing to do as the contents of these files do not depend on the options, or the contents of the specification file, other than the package declaration. Modifying the `SimpleCharStream.java` (or `JavaCharStream.java`) file should not be done until you are certain of your options, especially the STATIC and JAVA_UNICODE_ESCAPE options.

The custom files (`XXX.java`, `XXXTokenManager.java`, and `XXXConstants.java`) are produced every time you run JavaCC. Modifying any of the custom files is generally a very bad idea, as you'll have to modify them again after any change to the specification. Some people have written scripts (in, say, Perl) to do the modifications for them. I would regard this as a very last resort.

2.3 I changed option x; why am I having trouble?

Try deleting **all** files generated by JavaCC (see Question 2.1, " What files does JavaCC produce?") and then rerunning JavaCC. This issue usually comes up when the STATIC option is changed; JavaCC needs to generate new files, but it will not generate boiler-plate files unless they aren't there already.

2.4 How do I put the generated classes in a package?

Put a package declaration right after the PARSER_BEGIN(*XXX*) declaration in the .jj file.

2.5 How do I use JavaCC with ANT?

First, of course, download and install ANT from the Apache project's website (go to http://jakarta.apache.org/builds/jakarta-ant/release/ and choose the largest release number). This comes with a prebuilt JavaCC step documented under the "Optional Tasks" category.

The ANT task only invokes JavaCC if the grammar file is newer than the generated Java files. JavaCC assumes that the Java class name of the generated parser is the same as the name of the grammar file, ignoring the .jj. If this is not the case, the javacc task will still work, but it will always generate the output files.

Next, create a `build.xml` file which calls the step named javacc. Note that capitalization is important and that the ANT documentation for this step is titled JavaCC although the step name is javacc (the example in the documentation is right). I assume for this example that you have installed JavaCC in /usr/local on a Unix or GNU Linux box. A simple step will look like:

```
<javacc
    target="${sampleDir}/SimpleExamples/Simple1.jj"
```

```
    outputdirectory="${sampleDir}/SimpleExamples/java"
    javacchome="/usr/local/javacc2.1"
/>
```

ANT makes it easy to put the generated files in a separate directory. The javacchome attribute defines where you installed JavaCC.

This will need to be followed by a javac step to compile the generated files.

```
<javac
    srcdir="${sampleDir}/SimpleLevels/java"
    destdir="${sampleDir}/SimpleLevels/classes" />
```

Before running ANT you must add the JavaCC .zip file to your class path. The JavaCC step does not take a <classpath> modifier, so adding it to the global classpath is the only way to get this information into the step.

```
CLASSPATH=$CLASSPATH:/usr/local/javacc2.1/bin/lib/JavaCC.zip
export CLASSPATH
```

Now all you have to do is issue the command "ant".

A complete `build.xml` file is available at http://www.engr.mun.ca/~theo/JavaCC-FAQ/build.xml.

Chapter 3

The Token Manager

3.1 What is a token manager?

In conventional compiling terms, a token manager is a lexical analyzer. If that is Greek to you, here is an explanation. The token manager analyzes the input stream of characters breaking it up into chunks called tokens and assigning each token a "token kind". For example suppose the input is a C file

```
int main() {
     /*a short program */
     return 0 ; }
```

Then the token manager might break this into chunks as follows:

"int", " ", "main", "(", ")", " ", "{", "\n", "   ", "/*a short program */", ...

White space and comments are typically discarded, so the chunks are then

"int", "main", "(", ")", "{", "return", "0", ";", "}"

Each chunk of text is classified as one of a finite set of "token kinds".[1] For example the chunks above could be classified as, respectively,

[1] JavaCC's terminology here is a bit unusual. The conventional name for what JavaCC calls a "token kind" is "terminal" and the set of all token kinds is the "alphabet" of the EBNF grammar.

KWINT, ID, LPAR, RPAR, LBRACE, KWRETURN, OCTALCONST, SEMICOLON, RBRACE

Each chunk of text is represented by an object of class Token, each with the following attributes:

- .kind the token kind encoded as an int,
- .image the chunk of input text as a string,

and a few others.

This sequence of Token objects is produced based on regular expressions appearing in the .jj file.

The sequence is usually sent on to a parser object for further processing.

3.2 How do I read from a string instead of a file?

Here is one way

```
java.io.StringReader sr = new java.io.StringReader( str );
java.io.Reader r = new java.io.BufferedReader( sr );
XXX parser = new XXX( r );
```

3.3 What if more than one regular expression matches a prefix of the remaining input?

First a definition: If a sequence x can be constructed by catenating two other sequences y and z, i.e., $x = yz$, then y is called a "prefix" of x. If in addition y contains at least one character, it is called a "nonempty prefix".

There are three golden rules for picking which regular expression to use to identify the next token:

1. The regular expression must describe a nonempty prefix of the remaining input stream.

2. If more than one regular expression describes a nonempty prefix, then the regular expression that describes the longest possible prefix of the input stream is used. (This is called the "maximal munch rule".)

3. If more than one regular expression describes the longest possible nonempty prefix, then the regular expression that comes first in the .jj file is used.

For example, suppose you are parsing Java, C, or C++. The following three regular expression productions might appear in the .jj file

---

TOKEN : { <PLUS : "+">}
TOKEN : { <ASSIGN : "=">}
TOKEN : { <PLASSIGN : "+=" >}

Suppose the remaining input stream starts with

"`+=1; ...`" .

Rule 1 rules out the second production. Rule 2 says that the third production is preferred over the first. The order of the productions has no effect on this example.

Sticking with Java, C, or C++, suppose you have regular expression productions

---

TOKEN : { <KWINT : "int">}
TOKEN : { <IDENT : ["a"-"z","A"-"Z", "_"] (["a"-"z","A"-"Z","0"-"9","_"])* >}

Suppose the remaining input steams starts with

"`integer i; ...`" ,

then the second production would be preferred by the maximal munch rule (rule 2). But if the remaining input stream starts with

"`int i; ...`" ,

then the maximal munch rule is no help, since both rules match a prefix of length 3. In this case the KWINT production is preferred (by rule 3) because it comes first in the .jj file.

3.4 What if no regular expression matches a prefix of the remaining input?

If the remaining input is empty, an EOF token is generated. Otherwise, a TokenMgrError is thrown.

3.5 How do I make a character sequence match more than one token kind?

A common misapprehension of beginners is that the token manager will make its decisions based on what the parser expects. They write a couple of token definitions, for example

```
TOKEN : { <A : "x" | "y" > }
TOKEN : { <B : "y" | "z" > }
```

and expect the token manager to interpret "y" as an A if the parser "expects" an A and as a B if the parser "expects" a B. This is an interesting idea, but it isn't how JavaCC works[2]. As discussed in Question 3.3, " What if more than one regular expression matches a prefix of the remaining input?", the first match wins.

So what do you do. Let's consider the a more general situation where a and b are regular expressions. And we have token definitions

```
TOKEN : { <A : a > }
TOKEN : { <B : b > }
```

Suppose that a describes a set A and b describes a set B. Then (ignoring other regular expressions) A matches A but B matches $B - A$.

You want to the parser to be able to request a member of set B. If A is a subset of B there is a simple solution; create a nonterminal

```
Token b() : {Token t ; }{ (t=<A> | t=<B>) {return t;} }
```

[2] It's also an idea that leaves some questions open. What should the token manager do if the parser would accept either an A or a B? How do we write a parser for a language with reserved words?

Now use b() instead of <B> when you want a token in the set B.

If A is not a subset of B, there is more work to do. Create regular expressions a', b', and c' matching sets A', B', C' such that

$$A = C' \cup A'$$
$$B = C' \cup (B' - A')$$

Now you can write the following productions

```
TOKEN : { <C : c' > }
TOKEN : { <A : a' >
TOKEN : { <B : b' > }
Token a() : {Token t ; } { (t=<C> | t=<A>) {return t;} }
Token b() : {Token t ; } { (t=<C> | t=<B>) {return t;} }
```

Use a() when you need a member of set A and b() when you need a member of set B. Applied to the motivating example, we have

```
TOKEN : { <C : "y" > }
TOKEN : { <A : "x" > }
TOKEN : { <B : "z" > }
Token a() : {Token t ; } { (t=<C> | t=<A>) {return t;} }
Token b() : {Token t ; } { (t=<C> | t=<B>) {return t;} }
```

Of course this idea can be generalized to any number of overlapping sets.

There are two other approaches that might also be tried: One involves lexical states and the other involves semantic actions. All three approaches are discussed in Question 4.20, "How do I deal with keywords that aren't reserved?", which considers a special case of the problem discussed in this question.

3.6 How do I match any character?

Use ~[] .

3.7 How do I match exactly n repetitions of a regular expression?

If X is the regular expression and n is an integer constant, write

$$(X)\{n\}$$

You can also give a lower and upper bound on the number of repetitions:

$$(X)\{l, u\}$$

This syntax applies only to the tokenizer, it can't be used for parsing.

Note that this syntax is implemented essentially as a macro, so $(X)\{3\}$ is implemented the same as $(X)(X)(X)$ would be. Thus you should use it with discretion, aware that it can lead to a big generated tokenizer, if used to excess.

3.8 What are **TOKEN**, **SKIP**, and **SPECIAL_TOKEN**?

Regular expression productions are classified as one of four kinds:

- **TOKEN** means that when the production is applied, a Token object should be created and passed to the parser.
- **SKIP** means that when the production is applied, no Token object should be constructed.
- **SPECIAL_TOKEN** means that when the production is applied a Token object should be created but it should not be passed to the parser. Each of these "special tokens" can be accessed from the next Token produced (whether special or not), via its specialToken field.
- **MORE** is discussed in Question 3.13, "What is **MORE**?".

3.9 What are lexical states all about?

Lexical states allow you to bring different sets of regular expression productions in-to and out-of effect.

Suppose you wanted to write a JavaDoc processor. Most of Java is tokenized according to regular ordinary Java rules. But between a "`/**`" and the next "`*/`" a different set of rules applies in which keywords like "`@param`" must be recognized and where newlines are significant. To solve this problem, we could use two lexical states. One for regular Java tokenizing and one for tokenizing within JavaDoc comments. We might use the following productions:

```
// When a /** is seen in the DEFAULT state, switch to the IN_JAVADOC_COMMENT state
TOKEN : {
      <STARTDOC : "/**" > : IN_JAVADOC_COMMENT }

// When @param is seen in the IN_JAVADOC_COMMENT state, it is a token.
// Stay in the same state.
<IN_JAVADOC_COMMENT> TOKEN : {
      <PARAM : "@param" >}
...

// When a */ is seen in the IN_JAVADOC_COMMENT state, switch
// back to the DEFAULT state
<IN_JAVADOC_COMMENT> TOKEN : {
      <ENDDOC: "*/">: DEFAULT }
```

Productions that are prefixed by <IN_JAVADOC_COMMENT> apply when the lexical analyzer is in the IN_JAVADOC_COMMENT state. Productions that have no such prefix apply in the DEFAULT state. It is possible to list any number of states (comma separated) before a production. The special prefix <*> indicates that the production can apply in all states.

Lexical states are also useful for avoiding complex regular expressions. Suppose you want to skip C style comments. You could write a regular expression production:

SKIP : { <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/" >}

But how confident are you that this is right?[3] The following version uses a lexical state called IN_COMMENT to make things much clearer:

// When a / is seen in the DEFAULT state, skip it and switch to the IN_COMMENT state*
SKIP : {
    "/*" : IN_COMMENT }

// When any other character is seen in the IN_COMMENT state, skip it.
<IN_COMMENT>**SKIP** : {
    < ~[] >}

*// When a */ is seen in the IN_COMMENT state, skip it and switch back to the DEFAULT state*
<IN_COMMENT>**SKIP** : {
    "*/" : DEFAULT }

The previous example also illustrates a subtle behavioural difference between using lexical states and performing the same task with a single, apparently equivalent, regular expression. Consider tokenizing the C "statement":

```
i = j/*p ;
```

Assuming that there are no occurrences of `*/` later in the file, this is an error (since a comment starts, but doesn't end) and should be diagnosed. If we use

[3]This example is quoted from

examples/JJTreeExamples/eg4.jjt

Your maintainer inspected it carefully before copying it into another .jjt file. As testing revealed, it is not, however, correct, which shows that even the experts can be befuddled by complex regular expressions, sometimes. Can you spot the error? A correct regular expression is

---

<"/*" (~["*"] |("*")+ ~["/"])* ("*")+ "/">

... I think.

a single, complex, regular expression to find comments, then the lexical error will be missed and, in this example at least, a syntactically correct sequence of seven tokens will be found. If we use the lexical states approach then the behaviour is different, though again incorrect; the comment will be skipped; an EOF token will be produced after the token for "j"[4]; no error will be reported. We can correct the lexical states approach, however, with the use of **MORE**; see Question 3.13, "What is **MORE**?".

3.10 Can the parser force a switch to a new lexical state?

Yes, but it is very easy to create bugs by doing so. You can call the token manager's method SwitchTo from within a semantic action in the parser like this

```
{ token_source.SwitchTo(name_of_state) ; }
```

However, owing to look-ahead, the token manager may be well ahead of the parser. Consider Figure 1.2; at any point in the parse, there are a number of tokens on the conveyer belt, waiting to be used by the parser; technically the conveyer belt is a queue of tokens held within the parser object. Any change of state will take effect for the first token not yet in the queue. Syntactic look-ahead usually means there is at least one token in the queue, but there may be many more.

If you are going to force a state change from the parser be sure that, at that point in the parsing, the token manager is a known and fixed number of tokens ahead of the parser, and that you know what that number is.

If you ever feel tempted to call SwitchTo from the parser, stop and try to think of an alternative method that is harder to get wrong.

3.11 Is there a way to make SwitchTo safer?

Brian Goetz submitted the following code to make sure that, when a SwitchTo is done, any queued tokens are removed from the queue. There are three parts

[4] The rule that an EOF token is produced at the end of the file applies regardless of the lexical state.

to the solution:

- In the parser add a subroutine SetState to change the state. This subroutine can be found at http://www.engr.mun.ca/~theo/JavaCC-FAQ/SetState.txt. Use this subroutine to change states within semantic actions of the parser.

- In the token manager add a subroutine:

```
TOKEN_MGR_DECLS : {
    // Required by SetState
    void backup(int n) { input_stream.backup(n); }
}
```

- Use the USER_CHAR_STREAM option and use BackupCharStream as the CharStream class. BackupCharStream can be found at http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/webmacro/webmacro/src/org/webmacro/parser/BackupCharStream.java

3.12 How do I match an arbitrarily long sequence of characters?

You might be tempted to use (~[])+. This will match all characters up to the end of the file (see Question 3.3, “ What if more than one regular expression matches a prefix of the remaining input?”), which may not be what you want. Usually what you really want is to match all characters up to either the end of the file or until some stopping point. Consider for example a scripting language in which scripts are embedded in an otherwise uninterpreted text file set off by “<<” and “>>” tokens. Between the start of the file or a “>>” and the next “<<” or the end of file we need to match an arbitrarily long sequence that does not contain two “<” characters in a row. We could use a regular expression

(~[“<”] | “<” ~[“<”])+

Of course you don’t want to match this regular expression within a script and so we would use lexical states to separate tokenizing within scripts from

tokenizing outside of scripts (see Question 3.9, “What are lexical states all about?”).

A simpler method uses ~[] and moves the repetition up to the grammar level:

```
<DEFAULT> TOKEN : { <STARTSCRIPT : "<<" > : SCRIPT }
<DEFAULT> TOKEN : { < TEXT : ~[] > }

<SCRIPT> TOKEN : { <ENDSCRIPT : ">>" > : DEFAULT }
<SCRIPT> ... other TOKEN and SKIP productions for the SCRIPT state
```

Then the grammar is, in part,

```
void start() : {}

{
    (<TEXT>)*
    ( <STARTSCRIPT> script() <ENDSCRIPT>)*
    (<TEXT>)*
    <EOF>
}
```

3.13 What is MORE?

Regular expression productions are classified as one of four kinds:

- **TOKEN**, **SKIP**, and **SPECIAL_TOKEN** are discussed in Question 3.8, “What are **TOKEN**, **SKIP**, and **SPECIAL_TOKEN**?”.
- **MORE**.

MORE means that no token should be produced yet. Rather the characters matched will form part of the next token to be recognized. **MORE** means that there will be more to the token. After a sequence of one or more **MORE** productions have been applied, we must reach a production that is marked **TOKEN**, **SKIP**, **SPECIAL_TOKEN**. The token produced (or not produced

in the case of **SKIP**) will contain the saved up characters from the preceding **MORE** productions. Note that if the end of the input is encountered when the token manager is looking for more of a token, then a TokenMgrError is thrown. The assumption made by JavaCC is that the EOF token should correspond exactly to the end of the input, not to some characters leading up to the end of the input.

Let's revisit and fix the comment example from Question 3.9, "What are lexical states all about?". The problem was that unterminated comments were simply skipped rather than producing an error. We can correct this problem using **MORE** productions to combine the entire comment into a single token.

```
// When a /* is seen in the DEFAULT state, skip it and switch to the IN_COMMENT state
MORE : {
    "/*": IN_COMMENT }

// When any other character is seen in the IN_COMMENT state, skip it.
<IN_COMMENT>MORE : {
    < ~[] >}

// When a */ is seen in the IN_COMMENT state, skip it and switch back to the DEFAULT state
<IN_COMMENT>SKIP : {
    "*/": DEFAULT }
```

Suppose that a file ends with "/*a". Then no token can be recognized, because the end of file is found when the token manager only has a partly recognized token. Instead a TokenMgrError will be thrown.

3.14 Why do the example Java and C++ parsers report an error when the last line of a file is a single line comment?

The file is likely missing a newline character (or the equivalent) at the end of the last line.

These parsers use lexical states and **MORE** type regular expression productions to process single line comments thusly:

```
MORE : {
    "//": IN_SINGLE_LINE_COMMENT }

<IN_SINGLE_LINE_COMMENT>SPECIAL_TOKEN : {
    <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n">: DEFAULT }

<IN_SINGLE_LINE_COMMENT>MORE : {
    < ~[] >}
```

Clearly if an EOF is encountered while the token manager is still looking for more of the current token, there should be a TokenMgrError thrown.

Both the Java and the C++ standards agree with the example .jj files, but some compilers are more liberal and do not insist on that final newline. If you want the more liberal interpretation, try

```
SPECIAL_TOKEN : {
    <SINGLE_LINE_COMMENT: "//"(~["\n","\r"])* ("\n"|"\r"|"\r\n")? >}
```

3.15 What is a lexical action?

Sometimes you want some piece of Java code to be executed immediately after a token is matched. Lexical actions are placed immediately after the regular expression in a regular expression production. For example:

```
TOKEN : {
    <TAB : "\t"> { tabcount+=1; }
}
```

The Java statement { tabcount+=1; } will be executed after the production is applied.

3.16 How do I tokenize nested comments?

The answer lies in the fact that you can use SwitchTo in a lexical action (See Question 3.10. "Can the parser force a switch to a new lexical state?"and Question 3.15. " What is a lexical action?"). This technique might be useful for a number of things, but the one that comes up is nested comments. For example consider a language where comment start with "(*" and end with "*)" but can be nested so that

```
(* Comments start with (* and end with *) and can nest.  *)
```

is a valid comment. When a "*)" is found within a comment, it may or may not require us to switch out of the comment processing state.

Start by declaring a counter

```
TOKEN_MGR_DECLS : {
    int commentNestingDepth ;
}
```

When a "(*" is encounted in the DEFAULT state, set the counter to 1 and enter the COMMENT state:

```
SKIP : {  "(*" { commentNestingDepth = 1 ; } : COMMENT }
```

When a "(*" is encounted in the COMMENT state, increment the counter:

```
<COMMENT> SKIP : {  "(*" { commentNestingDepth += 1 ; } }
```

When a "*)" is encountered in the COMMENT state, either switch back to the DEFAULT state or stay in the comment state:

```
<COMMENT> SKIP : {  "*)" {
    commentNestingDepth -= 1;
    SwitchTo( commentNestingDepth==0 ? DEFAULT : COMMENT ) ; } }
```

Finally a rule is needed to mop up all the other characters in the comment.

```
<COMMENT> SKIP : { ~[] }
```

3.17 What is a common token action?

A common token action is simply a subroutine that is called after each token is matched. Note that this does not apply to "skipped tokens" nor to "special tokens".

3.18 How do I throw a ParseException instead of a TokenMgrError?

If you don't want any TokenMgrErrors being thrown, try putting a regular expression production at the very end of your .jj file that will match any character:

```
<*>TOKEN :
{
<UNEXPECTED_CHAR : ~[] >
}
```

However, this may not do the trick. In particular, if you use MORE, it may be hard to avoid TokenMgrErrors altogether. It is best to make a policy of catching TokenMgrErrors, as well as ParseExceptions, whenever you call an entry point to the parser. The only time I don't do this is when the token manager specification is so simple that I can be sure that no TokenMgrErrors can be thrown.

3.19 Why are line and column numbers not recorded?

In version 2.1 a new feature was introduced. You now have the option that the line and column numbers will not be recorded in the Token objects. The option is called KEEP_LINE_COLUMN. The default is true, so not knowing about this option *shouldn't* hurt you.

However, there appears to be a bug in the GUI interface to JavaCC (javaccw.exe), which sets this option to false (even if you explicitly set it to true in the .jj file).

The solution is to delete **all** generated files (see Question 2.3, " I changed option x; why am I having trouble?") and henceforth to not use the GUI interface to JavaCC.

Chapter 4

The Parser and Lookahead

4.1 Where should I draw the line between lexical analysis and parsing?

This question is dependant on the application. A lot of simple applications only require a token manager. However, many people try to do too much with the lexical analyzer, for example they try to write an expression parser using only the lexical analyzer.

4.2 What is recursive descent parsing?

JavaCC's generated parser classes work by the method of "recursive descent". This means that each BNF production in the .jj file is translated into a subroutine with roughly the following mandate:

> *If there is a prefix of the input sequence of tokens that matches this nonterminal's definition,*
> *then remove such a prefix from the input sequence*
> *else throw a ParseException*

I say only roughly, as the actual prefix matched is not arbitrary, but is determined by the rules of JavaCC.

4.3 What is left-recursion and why can't I use it?

Left-recursion is when a nonterminal contains a recursive reference to itself that is not preceded by something that will consume tokens.

The parser class produced by JavaCC works by recursive descent. Left-recursion is banned to prevent the generated subroutines from calling themselves recursively ad-infinitum. Consider the following obviously left recursive production

```
void A() : {} {
    A() B()
|
    C()
}
```

This will translate to a Java subroutine of the form

```
void A() : {} {
    if( some condition ) {
        A() ;
        B() ; }
    else {
        C() ; }
}
```

Now if the condition is ever true, we have an infinite recursion.

Luckly JavaCC will produce an error message, if you have left-recursive productions.

The left-recursive production above can be transformed, using looping, to

```
void A() : {} {
    C() ( B() )*
}
```

or, using right-recursion, to

```
void A() : {} {
    C() A1()
}
void A1() : {} {
    [ B() A1() ]
}
```

where A1 is a new production. General methods for left-recursion removal can be found in any text book on compiling.

4.4 How do I match an empty sequence of tokens?

Use {}. Usually you can use optional clauses to avoid the need. E.g. the production

```
void A() : {} { B() | {} }
```

is the same as the production

```
void A() : {} { [ B() ] }
```

Sometimes I'll write the former rather than the latter because I know that in the future there will be some semantic action associated with the empty alternative.

4.5 What is "lookahead"?

To use JavaCC effectively you have to understand how it looks ahead in the token stream to decide what to do. Your maintainer strongly recommends reading the lookahead mini-tutorial in the JavaCC documentation.

(See Question1.9) . The following questions of the FAQ address some common problems and misconceptions about lookahead. Ken Beesley has kindly contributed some supplementary documentation[1].

4.6 I get a message saying "Warning: Choice Conflict ... "; what should I do?

Some of JavaCC's most common error messages go something like this

> *Warning: Choice conflict ...*
>
> *Consider using a lookahead of 2 for ...*

Read the message carefully. Understand why there is a choice conflict (choice conflicts will be explained shortly) and take appropriate action. The appropriate action, in my experience, is rarely to use a lookahead of 2.

So what is a choice conflict. Well suppose you have a BNF production

```
void a() : {} {
    <ID> b()
|
    <ID> c()
}
```

When the parser applies this production, it must choose between expanding it to <ID> b() and expanding it to <ID> c(). The default method of making such choices is to look at the next token. But if the next token is of kind ID then either choice is appropriate. So you have a "choice conflict". For alternation (i.e. |) the default choice is the first choice; that is, if you ignore the warning, the first choice will be preferred, and in this example, the second choice is unreachable.

To resolve this choice conflict you can add a "LOOKAHEAD specification"to the first alternative. For example, if nonterminal b and nonterminal c can be distinguished on the basis of the token after the ID token, then the parser need only lookahead 2 tokens. You tell JavaCC this by writing:

[1] http://www.engr.mun.ca/~theo/JavaCC-FAQ/kens-javacc-lookahead-summary.txt

```
void a() : {} {
    LOOKAHEAD( 2 )
    <ID> b()
|
    <ID> c()
}
```

Ok, but suppose that b and c can start out the same and are only distinguishable by how they end. No predetermined limit on the length of the lookahead will do. In this case, you can use "syntactic lookahead". This means you have the parser look ahead to see if a particular syntactic pattern is matched before committing to a choice. Syntactic lookahead in this case would look like this:

```
void a() : {} {
    // Take the first alternative if an <ID>followed by a b() appears next
    LOOKAHEAD( <ID> b() )
    <ID> b()
|
    <ID> c()
}
```

The sequence <ID> b() may be parsed twice: once for lookahead and then again as part of regular parsing.

Another way to resolve conflicts is to rewrite the grammar. The above nonterminal can be rewritten as

```
void a() : {} {
    <ID>
    (
        b()
    |
        c()
    )
```

```
}
```

which may resolve the conflict.

Choice conflicts also come up in loops. Consider

```
void paramList() : {} {
    param()
    (
        <COMMA>
        param()
    )*
    (
        (<COMMA>)? <ELLIPSIS>
    )?
}
```

There is a choice of whether to stay in the * loop or to exit it and process the optional ELLIPSIS. But the default method of making the choice based on the next token does not work; a COMMA token could be the first thing seen in the loop body, or it could be the first thing after the loop body. For loops the default choice is to stay in the loop.

To solve this example we could use a lookahead of 2 at the appropriate choice point (assuming a param can not be empty and that one can't start with an ELLIPSIS.

```
void paramList() : {} {
    param()
    (
        LOOKAHEAD(2)
        <COMMA>
        param()
    )*
    (
        (<COMMA>)? <ELLIPSIS>
    )?
```

```
}
```

We could also rewrite the grammar, replacing the loop with a recursion.

Sometimes the right thing to do is to simply ignore the warning. Consider this classical example, again from programming languages

```
void statement() : {}
{
    <IF> exp() <THEN> statement()
    ( <ELSE> statement() )?
|
    ...other possible statements...
}
```

Because an ELSE token could legitimately follow a statement, there is a conflict. The fact that an ELSE appears next is not enough to indicate that the optional "<ELSE> statement()"should be parsed. Thus there is a conflict. The default for parsers is to take an option rather than to leave it; and that turns out to be the right interpretation in this case (at least for C, C++, Java, Pascal, etc.). If you want, you can write:

```
void statement() : {}
{
    <IF> exp() <THEN> statement()
    ( LOOKAHEAD( <ELSE>) <ELSE> statement() )?
|
    ...other possible statements...
}
```

to suppress the warning.

4.7 I added a LOOKAHEAD specification and the warning went away; does that mean I fixed the problem?

No. JavaCC will not report choice conflict warnings if you use a LOOKAHEAD specification. The absence of a warning doesn't mean that you've solved the problem correctly, it just means that you added a LOOKAHEAD specification.

Consider the following example:

```
void eg() : {}
{
    LOOKAHEAD(2)
    <A> <B> <C>
|
    <A> <B> <D>
}
```

Clearly the lookahead is insufficient (lookahead 3 would do the trick), but JavaCC produces no warning. When you add a LOOKAHEAD specification, JavaCC assumes you know what you are doing and suppresses any warnings.

4.8 Are semantic actions executed during syntactic lookahead?

No.

4.9 Are nested syntactic lookahead specifications evaluated during syntactic lookahead?

No!

This can is a bit surprising. Consider a grammar

```
void start( ) : { } {
    LOOKAHEAD( a() )
    a()
    <EOF>
|
    w()
    <EOF>
}
void a( ) : { } {
    (
        LOOKAHEAD( w() y() )
        w()
    |
        w() x()
    )
    y()
}
void w() : {} { "w" }
void x() : {} { "x" }
void y() : {} { "y" }
```

and an input sequence of “wxy”. You might expect that this string will be parsed without error, but it isn’t. The lookahead on a() fails so the parser takes the second (wrong) alternative in start. So why does the lookahead on a() fail? The lookahead specification within a() is intended to steer the parser to the second alternative when the remaining input starts does not start with “wy”. However during syntactic lookahead, this inner syntactic lookahead is ignored. The parser considers first whether the remaining input, “wxy”, is matched by the alternation (w() | w() x()). First it tries the first alternative w(); this succeeds and so the alternation (w() | w() x()) succeeds. Next the parser does a lookahead for y() on a remaining input of “xy”; this of course fails, so the whole lookahead on a() fails. Lookahead does not backtrack and try the second alternative of the alternation. Once one alternative of an alternation has succeeded, the whole alternation is considered to have

succeeded; other alternatives are not considered. Nor does lookahead pay attention to nested synactic LOOKAHEAD specifications.

This problem usually comes about when the LOOKAHEAD specification looks past the end of the choice it applies to. So a solution to the above example is to interchange the order of choices like this:

```
void a( ) : { } {
    (
        LOOKAHEAD( w() x() )
        w() x()
    |
        w()
    )
    y()
}
```

Another solution sometimes is to distribute so that the earlier choice is longer. In the above example, we can write

```
void a( ) : { } {
         LOOKAHEAD( w() y() )
        w() y()
    |
        w() x() y()
}
```

Generally it is a bad idea to write syntactic look-ahead specifications that look beyond the end of the choice they apply to. If you have a production

$$a \rightarrow A \mid B$$

and you transliterate it into JavaCC as

```
void a() : {} { LOOKAHEAD(C) A | B }
```

then it is a good idea that $L(C)$ (the language of strings matched by C) is a set of prefixes of $L(A)$. That is

$$\forall u \in L(C) \cdot \exists v \in \Sigma^* \cdot uv \in L(A)$$

In some cases to accomplish this you can put the "longer" choice first (that is, the choice that doesn't include prefixes of the other); in other cases you can use distributivity to lengthen the choices.

4.10 Are parameters passed during syntactic lookahead?

No.

4.11 Are semantic actions executed during syntactic lookahead?

No.

4.12 Is semantic lookahead evaluated during syntactic lookahead?

Yes. It is also evaluated during evaluation of LOOKAHEAD(n), for $n > 1$.

4.13 Can local variables (including parameters) be used in semantic lookahead?

Yes to a point.

The problem is that semantic lookahead specifications are evaluated during syntactic lookahead (and during lookahead of more than 1 token). But the subroutine generated to do the syntactic lookahead for a nonterminal will not declare the parameters or the other local variables of the nonterminal. This means that the code to do the semantic lookahead will fail to compile (in this subroutine) if it mentions parameters or other local variables.

So if you use local variables in a semantic lookahead specification within the BNF production for a nonterminal n, make sure that n is not used in syntactic lookahead, or in a lookahead of more than 1 token.

This is a case of three rights not making a right! It is right that semantic lookahead is evaluated in during syntactic lookahead, it is right (or at least useful) that local variables can be mentioned in semantic lookahead, and it is right that local variables do not exist during syntactic lookahead. Yet putting these three features together tricks JavaCC into producing uncompilable code. Perhaps a future version of JavaCC will put these interacting features on a firmer footing.

4.14 How does JavaCC differ from standard LL(1) parsing?

Well first off JavaCC is more flexible. It lets you use multiple token lookahead, syntactic lookahead, and semantic lookahead. If you don't use these features, you'll find that JavaCC is only subtly different from LL(1) parsing; it does not calculate "follow sets"in the standard way — in fact it can't as JavaCC has no idea what your starting nonterminal will be.

4.15 How do I communicate from the parser to the token manager?

It is usually a bad idea to try to have the parser try to influence the way the token manager does its job. The reason is that the token manager may produce tokens long before the parser consumes them. This is a result of lookahead.

Often the work-around is to use lexical states to have the token manager change its behaviour on its own.

In other cases, the work-around is to have the token manager not change its bevhaviour and have the parser compensate. For example in parsing C, you need to know if an identifier is a type or not. If you were using lex or yacc, you would probably write your parser in terms of token kinds ID and TYPEDEF_NAME. The parser will add typedef names to the symbol table after parsing each typedef definition. The lexical analyzer will look up

identifiers in the symbol table to decide which token kind to use. This works because with lex and yacc, the lexical analyzer is always one token ahead of the parser. In JavaCC, it is better to just use one token kind, ID, and use a nonterminal in place of TYPEDEF_NAME:

```
void typedef_name() : {} {
    LOOKAHEAD( { getToken(1).kind == ID && symtab.isTypedefName(
   getToken(1).image ) } )
    <ID>}
```

But you have to be careful using semantic look-ahead like this. It could still cause trouble. Consider doing a syntactic lookahead on nonterminal ‘statement’. If the next statement is something like

```
{ typedef int T ; T i ; i = 0 ; return i ; }
```

The lookahead will fail since the semantic action putting T in the symbol table will not be done during the lookahead! Luckly in C, there should be no need to do a syntactic lookahead on statements.

[TBD. Think through this example very carefully.]

4.16 How do I communicate from the token manager to the parser?

As with communication between from the parser to the token manager, this can be tricky because the token manager is often well ahead of the parser.

For example, if you calculate the value associated with a particular token kind in the token manager and store that value in a simple variable, that variable may well be overwritten by the time the parser consumes the relevant token. Instead you can use a queue. The token manager puts information into the queue and the parser takes it out.

Another solution is to use a table. For example in dealing with `#line` directives in C or C++, you can have the token manager fill a table indicating on which physical lines the `#line` directives occur and what the value given by the `#line` is. Then the parser can use this table to calculate the “source line number”from the physical line numbers stored in the Tokens.

4.17 What does it mean to put a regular expression within a BNF production?

It is possible to embed a regular expression within a BNF production. For example

---

```
//A regular expression production
TOKEN : { <ABC : "abc" >}

//A BNF production
void nonterm() : {} {
    "abc"
    "def"
    <(["0"-"9"])+>
    "abc"
    "def"
    <(["0"-"9"])+>
}
```

There are six regular expressions within the BNF production. The first is simply a Java string and is the same string that appears in the earlier regular expression production. The second is simply a Java string, but does not (we will assume) appear in a regular expression production. The third is a "complex regular"expression. The next three simply duplicate the first three.

The code above is essentially equivalent to the following:

```
//A regular expression production
TOKEN : { <ABC : "abc" >}
TOKEN : { <ANON0 : "def" >}
TOKEN : { <ANON1 : <(["0"-"9"])+>}
TOKEN : { <ANON2 : <(["0"-"9"])+>}

//A BNF production
void nonterm() : {}
{
    <ABC>
```

```
    <ANON0>
    <ANON1>
  <ABC>
    <ANON0>
    <ANON2>
}
```

In general when a regular expression is a Java string and identical to regular expression occurring in a regular expression production[2], then the Java string is interchangeable with the token kind from the regular expression production.

When a regular expression is a Java string, but there is no corresponding regular expression production, then JavaCC essentially makes up a corresponding regular expression production. This is shown by the "def" which becomes an anonymous regular expression production. Note that all occurrences of the same string end up represented by a single regular expression production.

Finally consider the two occurrences of the complex regular expression `<(["0"-"9"])+>`. Each one is turned into a different regular expression production. This spells trouble, as the ANON2 regular expression production will never succeed. (See Question 3.3. " What if more than one regular expression matches a prefix of the remaining input?")

See also Question 4.18, " When should regular expressions be put directly into a BNF production?" .

4.18 When should regular expressions be put directly into a BNF production?

First read Question 4.17, " What does it mean to put a regular expression within a BNF production?" .

For regular expressions that are simply strings, you might as well put them directly into the BNF productions, and not bother with defining them in a regular expression production.[3] For more complex regular expressions,

[2] And provided that regular expression applies in the DEFAULT lexical state.

[3] Ok there are still a few reasons to use a regular expression production. One is if you are using lexical states other than DEFAULT. Another is if you want to ignore the

it is best to give them a name, using a regular expression production. There are two reasons for this. The first is error reporting. If you give a complex regular expression a name, that name will be used in the message attached to any ParseExceptions generated. If you don't give it a name, JavaCC will make up a name like "<token of kind 42>". The second is perspicuity. Consider the following example:

```
void letter_number_letters() : {
    Token letter, number, letters; }
{
    letter=<["a"-"z"]>
    number=<["0"-"9"]>
    letters=<(["a"-"z"])+>
    { return some function of letter, number and letters ; }
}
```

The intention is to be able to parse strings like "`a9abc`". Written this way it is a bit hard to see what is wrong. Rewrite it as

```
TOKEN : < LETTER : ["a"-"z"] >}
TOKEN : < NUMBER : ["0"-"9"] >}
TOKEN : < LETTERS : (["a"-"z"])+ >}

void letter_number_letters() : {
    Token letter, number, letters; }
{
    letter=<LETTER>
    number=<NUMBER>
    letters=<LETTERS>
    { return some function of letter, number and letters ; }
}
```

and it might be easier to see the error. On a string like "`z7d`"the token manager will find a LETTER, a NUMBER and then another LETTER; the

case of a word. Also, some people just like to have an alphabetical list of their keywords somewhere.

BNF production can not succeed. (See Question 3.3, " What if more than one regular expression matches a prefix of the remaining input?")

4.19 How do I parse a sequence without allowing duplications?

This turns out to be a bit tricky. Of course you can list all the alteratives. Say you want A, B, C, each optionally, in any order, with no duplications; well there are only 16 possibilities:

```
void abc() : {} {
    [<A> [ <B> [ <C> ] ] ]
|
    <A> <C>[ <B> ]
|
    <B> [ <A> [ <C> ] ]
|
    <B> <C>[ <A> ]
|
    <C> [ <A> [ <B> ] ]
|
    <C> <B> [ <A> ]
}
```

This approach is already ugly and won't scale.

A better approach is to use semantic actions to record what has been seen

```
void abc() : {} {
    (
        <A>
        { if( seen an A already ) throw ParseException("Duplicate A");
        else record an A }
    |
```

```
        <B>
        { if( seen an B already ) throw ParseException("Duplicate B");
        else record an B }
    |
        <C>
        { if( seen an C already ) throw ParseException("Duplicate C");
        else record an C }
    )*
}
```

The problem with this approach is that it will not work well with syntactic lookahead. Ninety-nine percent of the time you won't care about this problem, but consider the following highly contrived example:

```
void toughChoice() : {}
{
    LOOKAHEAD( abc() )
    abc()
|
    <A> <A> <B> <B>
}
```

When the input is two A's followed by two B's, the second choice should be taken. If you use the first (ugly) version of abc, above, then that's what happens. If you use the second (nice) version of abc, then the first choice is taken, since syntactically abc is (<A> | <B> | <C>)*.

4.20 How do I deal with keywords that aren't reserved?

In Java, C++, and many other languages, keywords, like "int", "if", "throw" and so on are *reserved*, meaning that you can't use them for any purpose other than that defined by the language; in particular you can use them for variable names, function names, class names, etc. In some applications, keywords are not reserved. For example, in the PL/I language, the following is a valid statement

```
if if = then then then = else ; else else = if ;
```

Sometimes you want “if”, “then”, and “else” to act like keywords and sometimes like identifiers.

This is a special case of a problem a more general problem discussed in Question 3.5, “How do I make a character sequence match more than one token kind?”.

For a more modern example, in parsing URL’s, we might want to treat the word “http”as a keyword, but we don’t want to prevent it being used as a host name or a path segment. Suppose we write the following productions[4]:

```
TOKEN : { <HTTP : "http">}
TOKEN : { <LABEL : <ALPHANUM>|<ALPHANUM>( <ALPHANUM>|"-")* <ALPHANUM> > }
void httpURL() : {} { <HTTP> ":" "//" host() port_path_and_query() }
void host() : {} { <LABEL>("."<LABEL>)* }
```

Both the regular expressions labelled HTTP and LABEL, match the string “`http`”. As covered in Question 3.3, “ What if more than one regular expression matches a prefix of the remaining input?”, the first rule will be chosen; thus the URL

`http://www.http.org/`

will not be accepted by the grammar. So what can you do? There are basically three strategies: put choices in the grammar, replace keywords with semantic lookahead, or use lexical states.

Putting choices in the grammar. Going back to the original grammar, we can see that, the problem is that where we say we expect a LABEL we actually intended to expect either a LABEL or a HTTP. We can rewrite the last production as

---

```
void host() : {} { label() ("."label())* }
void label() : {} { <LABEL> | <HTTP> }
```

---

[4]This example is based on the syntax for HTTP URLs in RFC: 2616 of the IETF by R. Fielding, *et. al.* However, I’ve made a number of simplifications and omissions for the sake of a simpler example.

Replacing keywords with semantic lookahead. Here we eliminate the offending keyword production. In the example we would eliminate the regular expression production labelled HTTP. Then we have to rewrite httpURL as

```
void httpURL() : {} {
    LOOKAHEAD( {getToken(1).kind == LABEL && getToken(1).image.equals("http")}
   )
    <LABEL> ":" "//" host() port_path_and_query() }
```

The added semantic lookahead ensures that the URL really begins with a LABEL which is actually the keyword "http". [TBD Check this example.]

Using lexical states. The idea is to use a different lexical state when the word is reserved and when it isn't. (See Question 3.9, "What are lexical states all about?") In the example, we can make "http" reserved in the default lexical state, but not reserved when a label is expected. In the example, this is easy because it is clear when a label is expected: after a "//" and after a ".".[5] Thus we can rewrite the regular expression productions as

```
TOKEN : { <HTTP : "http">}
TOKEN : { <DSLASH : "//"> : LABELEXPECTED }
TOKEN : { <DOT : "."> : LABELEXPECTED }
<LABELEXPECTED> TOKEN : { <LABEL : <ALPHANUM>|<ALPHANUM>(
<ALPHANUM>|"-")* <ALPHANUM> > :DEFAULT }
```

And the BNF productions are

```
void httpURL() : {} { <HTTP> ":" <DSLASH> host() port_path_and_query()
}
void host() : {} { <LABEL>(<DOT> <LABEL>)* }
```

[5] And we are assuming that double slashes and dots are always followed by labels, in a syntactically correct input stream.

4.21 There's an error in the input, so why doesn't my parser throw a ParseException?

Perhaps you forgot the <EOF> in the production for your start nonterminal.

Chapter 5

Semantic Actions

5.1 I've written/found a parser, but it doesn't do anything?

You need to add semantic actions. Semantic actions are bits of Java code that get executed as the parser parses.

5.2 How do I capture and traverse a sequence of tokens?

Each Token object has a pointer to the next Token object. Well that's not quite right. There are two kinds of Token objects. There are regular token objects, created by regular expression productions prefixed by the keyword TOKEN. And, there are special token objects, created by regular expression productions prefixed by the keyword SPECIAL_TOKEN. Each regular Token object has a pointer to the next regular Token object. We'll deal with the special tokens later.

Now since the tokens are nicely linked into a list, we can represent a sequence of tokens occurring in the document with a class by pointing to the first token in the sequence and the first token to follow the sequence.

```
class TokenList {
    private Token head ;
    private Token tail ;
```

```
    TokenList( Token head, Token tail ) {
        this.head = head ;
        this.tail = tail ; }
    ...
}
```

We can create such a list using semantic actions in the parser like this:

```
TokenList CompilationUnit() : {
    Token head ;
} {
    { head = getToken( 1 ) ; }
    [ PackageDeclaration() ] ( ImportDeclaration() )* ( TypeDeclaration() )*
    <EOF>
    { return new TokenList( head, getToken(0) ) ; }
}
```

To print regular tokens in the list, we can simply traverse the list

```
class TokenList {
    ...
    void print( PrintStream os ) {
        for( Token p = head ; p != tail ; p = p.next ) {
            os.print( p.image ) ; } }
    ...
}
```

This method of traversing the list of tokens is appropriate for many applications.

Here is some of what I got from printing the tokens of a Java file:

```
publicclassToken{publicintkind;publicintbeginLine,
```

Obviously this is not much good for either human or machine consumption. I could just print a space between each pair of adjacent tokens. A nicer solution

is to capture all the spaces and comments using special tokens. Each Token object (whether regular or special) has a field called specialToken, which points to the special token that appeared in the text immediately prior, if there was one, and is null otherwise. So prior to printing the image of each token, we print the image of the preceding special token, if any:

```
class TokenList {
    ...
    private void printSpecialTokens( PrintStream ps, Token st ) {
        if( st != null ) {
            printSpecialTokens( ps, st.specialToken ) ;
            ps.print( st.image ) ; } }

    void printWithSpecials( PrintStream ps ) {
        for( Token p = head ; p != tail ; p = p.next ) {
            printSpecialTokens( ps, p.specialToken ) ;
            ps.print( p.image ) ; } }
}
```

If you want to capture and print a whole file, don't forget about the special tokens that precede the EOF token.

Chapter 6

JJTree and JTB

TBD. Your maintainer knows little about either of these tools and would especially appreciate volunteers to contribute to this part of the FAQ.

6.1 What are JJTree and JTB?

These are preprocessors that produce .jj files. The .jj files produced will produce parsers that produce trees.

6.2 Where can I find JJTree?

JJTree comes with JavaCC. See Question 1.6, " Where can I get JavaCC?".
.

6.3 Where can I find JTB?

See JTB: The Java Tree Builder Homepage[1]..

[1] http://www.cs.purdue.edu/jtb/

Chapter 7

Applications of JavaCC

7.1 Where can I find a parser for x?

First look in Dongwon Lee's JavaCC Grammar Repository[1].

Then ask the newsgroup or the mailing list.

7.2 How do I parse arithmetic expressions?

See the examples that come with JavaCC.

See any text on compiling.

See Parsing Epressions by Recursive Descent[2] and a tutorial by Theodore Norvell[3].

7.3 I'm writing a programming language interpreter; how do I deal with loops?

A lot of people who want to write an interpreter for a programming language seem to start with a calculator for expressions, evaluating during parsing, as is quite reasonable. Then they add, say, assignments and if-then-else statements, and all goes well. Now they want to add loops. Having committed to the idea that they are evaluating while parsing they want to know how

[1]http://www.cobase.cs.ucla.edu/pub/javacc/

[2]http://www.engr.mun.ca/∼theo/Misc/index.html#parsingExps

[3]http://www.engr.mun.ca/∼theo/JavaCC-Tutorial/.

back up the token manager so that loop-bodies can be reparsed, and thus reevaluated, over and over and over again.

It's a sensible idea, but it's clear that JavaCC will not make this approach pleasant. Your maintainer suggests translating to an intermediate code during parsing, and then executing the intermediate code. A tree makes a convenient intermediate code. Consider using JJTree or JTB (see Chapter 6).

If you still want to back up the token manager. I suggest that you start by tokenizing the entire file, capturing the tokens in a list (see Question 5.2, " How do I capture and traverse a sequence of tokens?".) or, better, a Vector. Now write a custom token manager that delivers this captured sequence of tokens, and also allows backing up.

Chapter 8

Comparing JavaCC with other tools

[TBD. Your maintainer would welcome comparisons with ANTLR, CUP, JLex, JFlex, and any others that users can contribute. My limited understanding is that CUP is similar to Yacc and Bison, and that JLex and JFlex are similar to Lex and Flex.]

8.1 Since $LL(1) \subset LALR(1)$, wouldn't a tool based on LALR parsing be better?

It's true that there are strictly more languages that can be described by LALR(1) grammars than by LL(1) grammars. Furthermore almost every parsing problem that arises in programming languages has an LALR(1) solution and the same can not be said for LL(1).

But the situation in parser generators is a big more complicated. JavaCC is based on LL(1) parsing, but it allows you to use grammars that are not LL(1). As long as you can use JavaCC's look-ahead specification to guide the parsing where the LL(1) rules are not sufficient, JavaCC can handle any grammar that is not left-recursive. Similarly tools based on LALR(1) or LR(1) parsing generally allow input grammars outside those classes.

A case in point is the handling of if-statements in C, Java, and similar languages. Abstracted, the grammar is

$$S \to x \mid iS \mid iSeS$$

The theoretical result is that there is no LL(1) grammar that can handle the construct, but there is an LALR(1) grammar. Experienced parser generator users ignore this result. Both users of LALR(1) based parser generator (such as yacc) and users of LL(1) based parser generators (such as JavaCC) generally use the same ambiguous set of grammar rules, which is neither LALR(1) nor LL(1), and use other mechanisms to resolve the ambiguity.

8.2 How does JavaCC compare with Lex and Flex?

Lex is the lexical analyzer supplied for many years with most versions of Unix. Flex is a freely distributable relative associated with the GNU project. JavaCC and lex/flex are actually quite similar. Both work essentially the same way, turning a set of regular expressions into a big finite state automaton and use the same rules (for example the maximal munch rule). The big difference is the lex and flex produce C, whereas JavaCC produces Java.

One facility that lex and flex have, that JavaCC lacks, is the ability to look ahead in the input stream past the end of the matched token. For a classic example, to recognize the Fortran keyword "DO", you have to look forward in the input stream to find a comma. This is because

```
DO 10 I = 1,20
```

is a do-statement, whereas

```
DO 10 I = 1.20
```

is an assignment to a variable called `DO10I` (Fortran totally ignores blanks). Dealing with this sort of thing is easy in lex, but very hard in JavaCC.

However JavaCC does have some nice features that Lex and Flex lack: Common token actions, MORE rules, SPECIAL_TOKEN rules.

8.3 How does JavaCC compare with other Yacc and Bison?

Yacc is a parser generator developed at Bell labs. Bison is a freely distributable reimplementation associated with the GNU project. Yacc and Bison produce C whereas JavaCC produces Java.

The other big difference is that Yacc and Bison work bottom-up, whereas JavaCC works top-down. This means that Yacc and Bison make choices after consuming all the tokens associated with the choice, whereas JavaCC has to make its choices prior to consuming any of the tokens associated with the choice. However, JavaCC's lookahead capabilities allow it to peek well ahead in the token stream without consuming any tokens; the lookahead capabilities ameliorate most of the disadvantages of the top-down approach.

Yacc and Bison require BNF grammars, whereas JavaCC accepts EBNF grammars. In a BNF grammar, each nonterminal is described as choice of zero or more sequences of zero or more terminals and nonterminals. EBNF extends BNF with looping, optional parts, and allows choices anywhere, not just at the top level. For this reason Yacc/Bison grammars tend to have more nonterminals than JavaCC grammars and to be harder to read. For example the JavaCC production

```
void eg() : {} {a() (b() [","])* }
```

might be written as

```
eg : a eg1
;
eg1 : /* empty */
      |eg1 b optcomma
;
optcomma : /* empty */
      | ','
;
```

More importantly, it is often easier to write semantic actions for JavaCC grammars than for Yacc grammars, because there is less need to communicate values from one rule to another.

Yacc has no equivalent of JavaCC's parameterized nonterminals. While it is fairly easy to pass information up the parse-tree in Yacc (and JavaCC), it is hard to pass information down the tree in Yacc. For example, if in the above example, if we computed information in parsing the a that we wanted to pass to the b, this is easy in JavaCC, using parameters, but hard in Yacc.

As the example above shows, Yacc has no scruples about left-recursive productions.

My assessment is that if your language is totally unsuitable for top-down parsing, you'll be happier with a bottom-up parser like Yacc or Bison. However, if your language can be parsed top-down without too many appeals to lookahead, then JavaCC's combination of EBNF and parameters can make life much more enjoyable.