Invisible Jacc

Version 1.1

http://www.invisiblesoft.com/jacc

Invisible Jacc is an LR parser generator with automatic error repair. It is written entirely in Java, and it produces Java scanners and parsers. Invisible Jacc has a fully object-oriented design. You can use Invisible Jacc to create compilers and other programs that need to analyze complex input files.

An Invisible Jacc scanner analyzes a source file, and recognizes tokens defined by regular expressions. The scanner supports start conditions (which allows different regular expressions to be active at different times during the scan), and right context (which allows a regular expression to be recognized only when it appears in a certain context). In addition, the scanner provides hooks for handling difficult cases where regular expressions are insufficient. The scanner can accept either ASCII or Unicode input files.

An Invisible Jacc parser analyzes a stream of tokens, and structures them according to a set of productions. Invisible Jacc supports LALR(1), full LR(1), and optimized LR(1) grammars. Invisible Jacc also supports ambiguous grammars, with very detailed control of ambiguity resolution. Ambiguous grammars make it easy to handle operators with differing precedence and associativity, and language constructs such as the “dangling else.”

The parser provides automatic error repair, using a modified LM error repair algorithm. When an error is found in the source, the parser automatically determines what symbols to insert and/or delete in order to continue the parse. A “cost” can be assigned to each symbol to tune the repair algorithm. Invisible Jacc generates the error repair tables automatically from the grammar specification.

The parser and scanner also include hooks for installing custom-written prescanners and preprocessors, to handle special language requirements.

If you want to use Invisible Jacc for educational or commercial purposes, we encourage you to contact Invisible Software to discuss your requirements. We also welcome comments and criticisms of the Invisible Jacc design and feature set.

Legal Stuff

Invisible Jacc License Agreement

Trademarks

Contents

Legal Stuff 2
Invisible Jacc License Agreement 2
Trademarks 2
Contents 3
1 Introduction 6
1.1 Packages 6
1.2 Grammar Specifications 7
1.3 Compilers 7
1.4 Development Environment 8
1.5 Run-Time Environment 8
1.6 Run-Time Components 8
1.7 Object Oriented Design 8
1.8 Guide to the Manual 8
1.9 Feedback 9
2 A First Example 10
2.1 Writing the Grammar Specification 10
2.1.1 The %options Section 12
2.1.2 The %terminals Section 12
2.1.3 The %productions Section 12
2.1.4 The %categories Section 13
2.1.5 The %tokens Section 14
2.2 Generating the Tables 15
2.2.1 The Invisible Jacc Graphical User Interface 15
2.2.2 The Invisible Jacc Command Line 17
2.3 Writing the Compiler 18
2.3.1 The Main Compiler Class 25
2.3.2 The Token Factory Classes 26
2.3.3 The Nonterminal Factory Classes 27
2.4 Running the Compiler 28
2.5 How It Works 30
2.5.1 How the Scanner Works 30
2.5.2 How the Parser Works 31
3 Invisible Jacc Grammar Specification 33
3.1 Lexical Structure 34
3.1.1 White Space 34
3.1.2 Line End 34
3.1.3 Comment 34
3.1.4 Keyword 34
3.1.5 Operator 35
3.1.6 Number 35
3.1.7 Identifier 35
3.2 File Structure 37
3.3 The %options Section 38
3.3.1 The %options Header 38
3.3.2 The %lr1, %plr1, and %lalr1 Options 39
3.3.3 The %repair Option 39
3.3.4 The %goal Option 40
3.3.5 The %charsetsize Option 40
3.3.6 The %java Option 40
3.4 The %terminals Section 42
3.4.1 The %terminals Header 42
3.4.2 Terminal Definition 42
3.4.3 More About Insertion and Deletion Costs 43

3.5 The %productions Section ... 44
3.5.1 The %productions Header ... 44
3.5.2 Production Definition ... 44
3.5.3 Parser Operation ... 46
3.5.4 Example: Using Link Names ... 46
3.5.5 Example: Using Parameters ... 47
3.5.6 Example: Left Recursion ... 48
3.5.7 Example: Right Recursion ... 49
3.5.8 Example: Optional Elements ... 49
3.5.9 Example: Interior Actions ... 50
3.5.10 Example: The Dangling Else ... 50
3.5.11 Example: Associativity ... 52
3.5.12 Example: Operator Precedence and Associativity ... 53
3.5.13 Example: Using a Nonterminal in a Precedence Clause ... 55
3.6 The %categories Section ... 57
3.6.1 The %categories Header ... 58
3.6.2 Category Definition ... 58
3.6.3 Category Expressions ... 58
3.6.4 Examples of Category Definitions ... 60
3.7 The %conditions Section ... 61
3.7.1 The %conditions Header ... 61
3.7.2 Condition Definition ... 61
3.7.3 Example: C-Style Comments ... 62
3.7.4 Example: Java Comments and White Space ... 64
3.7.5 Example: Rescanning Input ... 68
3.8 The %tokens Section ... 72
3.8.1 The %tokens Header ... 73
3.8.2 Token Definition ... 73
3.8.3 Regular Expressions ... 74
3.8.4 Examples of Regular Expressions ... 75
3.8.5 Scanner Operation ... 76
3.8.6 Example: Using Link Names ... 78
3.8.7 Example: Case-Sensitive Keywords ... 79
3.8.8 Example: Case-Insensitive Keywords ... 79
3.8.9 Example: Numeric Formats ... 80
3.8.10 Example: Identifiers ... 80
3.8.11 Example: Error Tokens ... 81
3.8.12 Example: Numeric Suffixes ... 82
3.8.13 Example: Java Strings ... 83
3.8.14 Example: Include Files ... 87
3.8.15 Example: The Fortran DO Statement ... 89
3.8.16 Example: The Range Operator ... 90
3.8.17 Example: Detecting End-of-File ... 91
3.8.18 Example: Rejecting a Token ... 92
3.8.19 Example: Setting the Token Length ... 93
3.8.20 Example: Assembling Different Types of Tokens ... 95
4 Running the Parser Generator ... 97
4.1 The Invisible Jacc Graphical User Interface ... 97
4.2 The Invisible Jacc Command Line ... 100
5 Additional Examples ... 102
5.1 Verbose Output ... 102
5.2 Debug Output ... 102
5.3 Scientific Calculator ... 102
5.4 Parsing Conflicts ... 103
5.5 Scanning Java Source Code ... 104
5.6 Advanced Example: A Java Class Summarizer ... 105

5.6.1 Special Requirements for Class Summaries ... 106
5.6.2 Signatures ... 106
5.6.3 Design Overview ... 107
5.6.4 Running the Class Summarizer ... 108
5.7 Invisible Jacc Input Syntax ... 109
6 Package invisible.jacc.gen ... 110
6.1 Class GenFrontEnd ... 111
6.2 Class GenGUI ... 117
6.3 Class GenGUIPanel ... 118
6.4 Class GenMain ... 119
6.5 Interface GenObserver ... 120
7 Package invisible.jacc.parse ... 124
7.1 Class CompilerModel ... 126
7.2 Class ErrorOutput ... 136
7.3 Class ErrorOutputMulticaster ... 140
7.4 Class ErrorOutputStream ... 142
7.5 Class ErrorOutputWriter ... 144
7.6 Class InternalCompilerException ... 146
7.7 Class InterruptedCompilerException ... 147
7.8 Class NonterminalFactory ... 148
7.9 Class Parser ... 150
7.10 Interface ParserClient ... 154
7.11 Class ParserTable ... 156
7.12 Interface Preprocessor ... 160
7.13 Class PreprocessorInclude ... 162
7.14 Interface Prescanner ... 164
7.15 Interface PrescannerByte ... 165
7.16 Class PrescannerByteStream ... 167
7.17 Interface PrescannerChar ... 168
7.18 Class PrescannerCharReader ... 170
7.19 Class PrescannerJavaSource ... 171
7.20 Interface PrescannerJavaSourceClient ... 173
7.21 Class ProductInfo ... 174
7.22 Class Scanner ... 175
7.23 Interface ScannerClient ... 186
7.24 Class ScannerTable ... 187
7.25 Class SyntaxException ... 191
7.26 Class Token ... 192
7.27 Class TokenFactory ... 195
7.28 Interface TokenStream ... 200
8 References ... 201

1 Introduction

Invisible Jacc is a Java class library for scanning and parsing. *Scanning* is the process of taking a source file and breaking it into tokens. *Parsing* is the process of taking a stream of tokens and structuring it according to a set of productions.

The Invisible Jacc scanner recognizes tokens that are defined by regular expressions. Token recognition is performed using deterministic finite automata that are generated by Invisible Jacc. At any given point in the input, the scanner finds the longest string that matches any of the token definitions. If necessary, the scanner can also perform lookahead on the input.

Each token definition can be associated with some Java code. The Java code is wrapped inside an object, which is called a *token factory*. Whenever the scanner recognizes a token, it executes the Java code in the corresponding token factory. The token factory can assign a value to the token, and perform any other actions that are required for the token.

The Invisible Jacc parser structures input according to a set of productions. Productions are recognized using an LR parser that is generated by Invisible Jacc. The parser accepts LALR(1) and LR(1) grammars, and it also accepts ambiguous grammars with rules that define precedence and associativity. In addition, the parser provides automatic error repair using an LM error repair table that is generated by Invisible Jacc.

Each production can be associated with some Java code. The Java code is wrapped inside an object, which is called a *nonterminal factory*. Whenever the parser recognizes a production, it executes the Java code in the corresponding nonterminal factory. The nonterminal factory can compute a value for the production's left hand side, and perform any other actions that are required for the production.

1.1 Packages

There are two main Java packages in Invisible Jacc.

- Package `invisible.jacc.gen` contains the classes that are used to generate scanner tables and parser tables. This package contains the code that reads a grammar specification file, and constructs the deterministic finite automata and the LR parser.
- Package `invisible.jacc.parse` contains the classes that are used to construct a compiler. This package contains the Invisible Jacc scanner and parser. It also contains prescanner classes, preprocessor classes, and a variety of related classes. In addition, there is a "model" compiler class.

In addition to the two main packages, there are seven additional packages:

- Package `invisible.jacc.util` contains a variety of utility classes that are used by the two main packages.
- Package `invisible.jacc.utilg` contains a variety of graphics utility classes that are used by the graphical user interface.
- Package `invisible.jacc.ex1` contains an example compiler for a scientific calculator language.
- Package `invisible.jacc.ex2` contains an example compiler for a very simple calculator language that only supports addition and subtraction. This example is recommended as a first example for learning Invisible Jacc.
- Package `invisible.jacc.ex3` contains several examples of grammar specifications that have parsing conflicts.
- Package `invisible.jacc.ex4` contains an example compiler that reads Java source code and outputs a list of all identifiers used in the code.
- Package `invisible.jacc.ex5` contains an example compiler that reads Java source code and outputs a summary of all the classes, interfaces, methods, fields, and constructors in the code. This is a large

example intended to illustate a real-world application with a real-world programming language. The example includes an LALR(1) parser for the complete Java language, including Java 1.1 extensions.

1.2 Grammar Specifications

To use Invisible Jacc, you write a *grammar specification file*. There are two main parts to the grammar specification file: a set of token definitions, and a set of production definitions.

Each token is defined in terms of a regular expression. For example, a token that represents an identifier in a programming language could be written like this:

```
identifier = letter (letter | digit)*;
```

This regular expression expresses the idea that an identifier consists of a letter, followed by a series of zero or more letters and digits.

Each production is a rule that specifies how tokens can be combined into larger structures. For example, a production that represents an assignment statement in a programming language could be written like this:

```
AssignmentStatement -> identifier '=' Expression ';';
```

This production expresses the idea that an assignment statement consists of an identifier, followed by an equals sign, followed by an expression, followed by a semicolon.

Once you have written the grammar specification file, you use Invisible Jacc to generate scanner and parser tables.

1.3 Compilers

The goal of Invisible Jacc is to provide the building blocks so you can construct a compiler quickly. We now describe the type of compiler that Invisible Jacc lets you construct.

Invisible Jacc is designed with the idea that a compiler works in four stages:

- First, a *prescanner* reads the source file into a buffer, and performs any preliminary transformations on the characters of the source file. You can think of the prescanner as a "filter" which can modify the stream of characters that passes through it. For example, a prescanner that reads Java code would recognize Unicode escape sequences and convert them into the corresponding character values. A prescanner that reads Fortran code would remove all space characters outside of literals. In many cases, the prescanner performs no transformations at all. The prescanner's output is a stream of characters.
- Second, a *scanner* reads a stream of characters from the prescanner, and groups them into *tokens*. Every time the scanner recognizes a token, it calls a user-supplied routine called a *token factory*. The token factory can perform whatever operations are necessary for that particular token. The scanner's output is a stream of tokens.
- Third, a *preprocessor* reads a stream of tokens from the scanner, and performs any required transformations on the token stream. You can think of the preprocessor as a "filter" which can modify the stream of tokens that passes through it. For example, the preprocessor can recognize "include" directives, which cause another source file to be included within the current source file. The preprocessor can also perform macro expansion. The preprocessor's output is a stream of tokens.
- Finally, a *parser* reads a stream of tokens from the preprocessor, and structures them according to the language grammar. Every time the parser recognizes a production, it calls a user-supplied routine called a *nonterminal factory*. The nonterminal factory can perform whatever actions are associated with that particular production. The parser does not have any output of it own; it is executed for the purpose of calling the nonterminal factories.

To construct a compiler, you must combine a prescanner class, a scanner class, a preprocessor class, and a parser class. The four classes are chained together, with each class sending its output to the next.

In Invisible Jacc, the scanner and parser classes are standardized. You never need to write your own scanner or parser. Instead, you use the grammar specification file, along with token factories and nonterminal factories, to customize the behavior of the scanner and parser. The Invisible Jacc scanner is in class `Scanner`, and the Invisible Jacc parser is in class `Parser`; you should never need to modify or replace these classes.

In contrast, prescanner and preprocessor classes are not standardized. Invisible Jacc comes with two different prescanner classes, and one preprocessor class. The classes included with Invisible Jacc are sufficient for many cases. But, you may find it necessary to replace the prescanner or preprocessor with one that is custom-written for your language. To replace the prescanner, you must write a class that implements the `Prescanner` interface. Likewise, to replace the preprocessor, you must write a class that implements the `Preprocessor` interface.

1.4 Development Environment

To use Invisible Jacc you need a Java development environment, including a Java compiler.

Invisible Jacc version 1.1 is designed to work with any Java 1.1 compliant compiler. We have successfully tested it with Microsoft's Visual J++ version 6.0. But Invisible Jacc is written in standard Java 1.1, so you should be able to use any Java 1.1 environment.

We have included full source for Invisible Jacc, so you can study it, modify it, and recompile it if necessary.

1.5 Run-Time Environment

Invisible Jacc is designed so that the programs you create should run on any Java VM.

Even if you use a Java 1.1 compiler, the programs you create won't require a Java 1.1 VM, unless you use Java 1.1 specific features in your code.

1.6 Run-Time Components

If you create a program with Invisible Jacc and you want to distribute your program to others, then you will have to distribute packages `invisible.jacc.parse` and `invisible.jacc.util` as part of your program.

We generally allow these two packages to be distributed free of charge for non-commercial purposes. Please send email to `invisoft@invisiblesoft.com` if you have a need to distribute these packages.

1.7 Object Oriented Design

One of our main goals for Invisible Jacc was to have a good object-oriented design. As a result, Invisible Jacc does not produce a monolithic, "ready-to-run" compiler. Instead, it provides a library of classes and interfaces, each of which encapsulates one part the compilation process. You can combine these classes, along with your own code, to produce the final working compiler.

1.8 Guide to the Manual

Chapter 2 gets you started with Invisible Jacc by presenting a fully worked out example. It shows you how to write a specification file for a simple grammar, and then use Invisible Jacc to create the scanner and parser tables. Next, it shows you how to write the compiler class, and the associated token factories and nonterminal factories. Finally, it shows you how to run the compiler on a sample input file.

Chapter 3 is a complete description of the Invisible Jacc grammar specification file format. It also contains a lot of examples that show you how to accomplish common tasks.

Chapter 4 presents the Invisible Jacc graphical user interface, and the Invisible Jacc command-line syntax.

Chapter 5 discusses the other examples that are included with Invisible Jacc.

Chapter 6 is complete documentation for the package `invisible.jacc.gen`. This package contains the Invisible Jacc parser generator.

Chapter 7 is complete documentation for the package `invisible.jacc.parse`. This package contains the Invisible Jacc scanner and parser, plus other classes used to construct a compiler.

Chapter 8 is a list of references.

1.9 Feedback

We welcome and encourage feedback on the design of Invisible Jacc. You can find us on the World Wide Web at the following locations:

Invisible Jacc home page:	`www.invisiblesoft.com/jacc`
Invisible Software home page:	`www.invisiblesoft.com`
Invisible Software email:	`invisoft@invisiblesoft.com`

You can use the Invisible Jacc home page to send us your comments and questions, and to obtain the latest information about Invisible Jacc.

Please remember that this copy of Invisible Jacc is for your personal, non-commercial use. Do not distribute copies of Invisible Jacc to anyone else, and don’t post copies on the Internet. If you want to use Invisible Jacc for educational or commercial purposes, please contact us and we’ll discuss your requirements.

Also, please remember that you got Invisible Jacc for free. We’ll try to respond to questions and problems, but our ability to do so is limited. References [1] and [2] are excellent sources of information about scanning, parsing, and compiler design.

2 A First Example

A good way to begin learning about Invisible Jacc is to study a simple example. We will present a complete compiler for a simple calculator language. The compiler reads arithmetic expressions from a source file, calculates the value of each expression, and prints the results. In order to keep the example simple, our calculator language only includes addition, subtraction, and parentheses, and the values are integers. Also, we allow comments that are introduced by "`//`" and extend to the end of the line. Here is an example of a source file written in our calculator language:

```
7+11;            // result should be 18
7-11;            // result should be -4
4-5-6;           // result should be -7
(4-5)-6;
4-(5-6);
100-(45+12)+123;
```

We will begin by showing how to write a specification for the calculator language. Then, we will show how to use Invisible Jacc to generate scanner and parser tables. Finally, we will show how to write the compiler code, and combine the pieces to make a fully functional compiler.

You can find all the files for this example in directory `invisible\jacc\ex2`.

2.1 Writing the Grammar Specification

The specification file for our grammar is shown below. You can find this file in `Ex2Grammar.jacc`. Notice that Invisible Jacc grammar specification files by convention use the file extension "`.jacc`". In this printed copy, we have removed most of the comments to make the file shorter.

```
// ----- Generator options -----

%options:
%java invisible.jacc.ex2.Ex2Grammar;   // Java package and class

// ----- Terminal symbols for the grammar -----

%terminals:
';';
'(';
')';
'+';
'-';
number;

// ----- Productions for the grammar -----

%productions:
Goal -> StatementList;
StatementList -> /* empty */ ;
StatementList -> StatementList Statement;
Statement -> Expression ';';
Expression {primary} -> Primary;
Expression {add} -> Expression '+' Primary;
Expression {subtract} -> Expression '-' Primary;
Primary {number} -> number;
Primary {paren} -> '(' Expression ')';
```

```
// ----- Character categories -----

%categories:
';' = ';';
'(' = '(';
')' = ')';
'+' = '+';
'-' = '-';
decDigit = '0'..'9';       // decimal digits 0 thru 9
space = 9 | 12 | 32;       // tab, form feed, and space
'/' = '/';                 // slash character for comments
cr = 13;                   // carriage return
lf = 10;                   // line feed
notEol = %any - 10 - 13;   // any character that isn't a line end

// ----- Tokens -----

%tokens:
';' = ';';
'(' = '(';
')' = ')';
'+' = '+';
'-' = '-';
number = decDigit+;
whiteSpace = space* ('/' '/' notEol*)?;
lineEnd = cr | lf | cr lf;
```

Let us examine the contents of this specification file. First, notice that you can write comments in the file, just like you do in a Java program. There are two types of comments: a *line comment* that is introduced by “`//`” and extends to the end of the line, and a *C-style comment* that is introduced by “`/*`” and terminated by “`*/`”.

The file is divided into five sections, each of which is introduced by a keyword and a colon. The five sections are the %options section, %terminals section, the %productions section, the %categories section, and the %tokens section.

In addition to these five sections, a specification file can have one other type of section: a %conditions section. We will describe this other section later on.

The sections in a specification file can be written in any order. Also, it is possible to have more than one section of a given type.

Notice that keywords all begin with the % character. This makes it easy to spot the keywords in a specification file.

Each section contains a series of *statements*. As with Java code, a statement is always terminated by a semicolon. Also like Java code, statements are free-form. You can write each statement on a separate line, or you can write several statements on one line, or you can make one statement span several lines.

We will now discuss each of the five sections in our example file. The first section, %options, specifies options for the parser generator. The next two sections, %terminals and %productions, are associated with the parser. These two sections define the context-free grammar that the parser recognizes. The last two sections, %categories and %tokens, are associated with the scanner. These sections define a set of regular expressions that the scanner recognizes as input tokens.

2.1.1 The %options Section

The %options section gives options for the parser generator. In this example, there is only one option: the `%java` option.

The `%java` option gives the Java name for the grammar specification. It is written as a fully-qualified Java package and class name. In our example, the Java name is `invisible.jacc.ex2.Ex2Grammar`. The `%java` statement is terminated by a semicolon.

The Java class name should match the grammar specification file name. In this case, the Java class name is `Ex2Grammar`, and the grammar specification file name is `Ex2Grammar.jacc`.

2.1.2 The %terminals Section

The %terminals section lists the terminal symbols of the grammar. In this example, there are six terminal symbols:

```
;  (  )  +  -  number
```

Notice that each terminal symbol is listed in a separate statement, and each statement is terminated by a semicolon. In addition to the symbol name, each statement can optionally include *costs* that are used to tune the error repair algorithm. None of the statements in our example include costs.

In a specification file, each symbol name is written enclosed in single quotes. However, you can omit the quotes if the symbol name happens to consist entirely of letters and digits, with a letter as the first character. (In Invisible Jacc, the underscore `'_'` and dollar `'$'` characters count as letters.) So the symbol `'number'` can be written with or without quotes. For example, the following two statements are equivalent:

```
number;
'number';
```

Invisible Jacc considers the quoted and unquoted forms of a name to be identical. You can use them interchangeably.

2.1.3 The %productions Section

The %productions section lists the productions of the grammar. Each production is written as a statement, terminated by a semicolon. Each production is required to have the following three components:

- A symbol name, which is called the *left hand side*;
- An *arrow*, formed by a minus sign and a greater-than sign; and
- A series of zero or more symbol names, which is called the *right hand side*.

In addition to these three mandatory components, a production can optionally have a *link name*, a *parameter*, and *precedence clauses*. In our example, the names enclosed in curly braces, like “`{primary}`,” are link names. For now, we will ignore these optional components.

A symbol which appears on the left hand side of a production is called a *nonterminal symbol*. Every symbol in the grammar must be either a terminal symbol that is listed in the %terminals section, or a nonterminal symbol that appears on the left hand side of a production. In our example, we have five nonterminal symbols:

```
Goal  StatementList  Statement  Expression  Primary
```

As was the case with the terminal symbol `'number'`, each of these nonterminal symbols could be written either with or without enclosing single quotes.

In our example there are nine productions. Let us examine what they say:

- The *Goal* is to compile a StatementList.

- A *StatementList* is a list of zero or more Statements.
- A *Statement* is an Expression followed by a semicolon.
- An *Expression* can be either a Primary, or an Expression plus a Primary, or an Expression minus a Primary.
- A *Primary* can be either a number, or an Expression enclosed in parentheses.

This describes a simple language where numbers can be combined using addition, subtraction, and parentheses. Each expression is written in a separate statement, terminated with a semicolon. A complete program consists of a series of such statements. In technical terms, this is an *LALR(1) grammar*.

2.1.4 The %categories Section

The %categories section lists the *character categories* used in scanning the source file. A character category is simply a set of characters. As we will see shortly, character categories are the basic building block from which regular expressions are constructed.

Each statement in the categories section contains three components:

- An identifier, which is called the *category name*;
- An equal sign; and
- A *category expression* which describes a set of characters.

Like symbols, category names are written enclosed in single quotes. Also like symbols, you can omit the quotes if the category name happens to consist of letters and digits, with a letter as the first character.

In our example there are eleven character categories:

```
 ;  (  )  +  -  decDigit  space  /  cr  lf  notEol
```

Let’s examine each one to see what it means.

- The character category `';'` consists of the single character `';'`. Although the statement `';' = ';'` might look like it isn’t doing anything, in fact it is. The `';'` on the left of the equal sign is the *name* of the category. The `';'` on the right of the equal sign is an *expression* that defines a set of characters, in this case just the semicolon character. Using the name `';'` makes the regular expressions easier to read, but you can use any name you like. For example, if you wanted to name this category `'semicolon'` instead of `';'` you could have written the following statement:

  ```
  semicolon = ';';
  ```

- Likewise, the character categories `'('`, `')'`, `'+'`, `'-'`, and `'/'` each consist of a single character.
- The character category `'decDigit'` consists of the digits `'0'` through `'9'`. Writing two dots indicates a *range* of character values. In fact, there are several different ways you could define `decDigit`, all of which are equivalent:

  ```
  decDigit = '0'..'9';
  decDigit = 0x30..0x39;
  decDigit = 48..57;
  decDigit = '0123456789';
  decDigit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
  ```

- The character category `'space'` consists of the three characters tab, form feed, and space. (Recall that 9 is the character code for tab, 12 is the character code for form-feed, and 32 is the character code for space.) The vertical bar indicates the *union* of two sets. The character codes could also be written in hexadecimal, like this:

  ```
  space = 0x09 | 0x0C | 0x20;
  ```

- The character category `'cr'` consists of the carriage return character (character code 13).
- Likewise, the character category `'lf'` consists of the line feed character (character code 10).
- The character category `'notEol'` includes any character except carriage return and line feed. The keyword `%any` denotes the set of all characters. (Remember that all keywords begin with the % character.) The minus sign indicates the *difference* of two sets.

2.1.5 The %tokens Section

The %tokens section lists the tokens recognized by the scanner. Each token is written as a statement, terminated by a semicolon. Each token is required to have the following three components:

- An identifier, which is called the *token name*;
- An equal sign; and
- A *regular expression* which describes a set of strings.

In addition to these three mandatory components, a token can optionally have a *link name*, a *parameter*, and a *right context*. For now, we will ignore these optional components.

Like symbols and category names, token names are written enclosed in single quotes. Also like symbols and category names, you can omit the quotes if the token name happens to consist of letters and digits, with a letter as the first character.

In our example there are eight tokens:

```
 ;  (  )  +  -  number  whiteSpace  lineEnd
```

Let us examine each one to see what it means.

- The token `';'` represents a semicolon. Although the statement `';' = ';'` might look like it isn't doing anything, in fact it is. The `';'` on the left of the equal sign is the name of the *token*. The `';'` on the right of the equal sign is the name of a *character category*. When a character category appears in a regular expression, it represents any one-character string where the character is a member of the category.
- Likewise, the tokens `'('`, `')'`, `'+'`, and `'-'` each includes a single one-character string.
- The token `'number'` represents a string of one or more decimal digits. Let us examine how this token is constructed:
  - The character category `'decDigit'` represents any decimal digit from `0` through `9`.
  - The operator "`+`" denotes *positive closure*, which means one or more repetitions of the previous element.
- The token `'whiteSpace'` represents white space characters and comments. In this example, a white space character is space, tab, and form feed. A comment is introduced by "`//`" and extends to the end of the line. Let us examine how this token is constructed:
  - The character category `'space'` represents a white space character.
  - The operator "`*`" denotes *Kleene closure*, which means zero or more repetitions of the previous element. So, the expression `space*` denotes zero or more white space characters.
  - The character category `'/'` represents a slash character.
  - The character category `'notEol'` represents any character other than carriage return and line feed. So, the expression `notEol*` denotes all characters up to the end of the line.

    - The expression `'/' '/' notEol*` denotes a slash character, followed by another slash character, followed by all characters up to the end of the line. In other words, this expression denotes a comment. Writing expressions one after the other denotes *catenation*, which means forming strings by selecting one string from each set and joining them end-to-end.
    - The operator “`?`” denotes *optional closure*, which means zero or one repetitions of the previous element. So, the expression `('/' '/' notEol*)?` denotes zero or one comments. Notice that you can use parentheses to group terms in a regular expression.
    - Finally, the expression `space* ('/' '/' notEol*)?` denotes zero or more white space characters, followed by zero or one comments. This is another example of *catenation*.
- The token `'lineEnd'` represents the end of a line. It can be a carriage return, a line feed, or carriage return followed by line feed. Let us examine how this token is constructed:
    - The character category `'cr'` represents a carriage return character.
    - The character category `'lf'` represents a line feed return character.
    - The expression `cr lf` denotes a carriage return followed by a line feed. Writing expressions one after the other denotes *catenation*, which means forming strings by selecting one string from each set and joining them end-to-end.
    - Finally, the expression `cr | lf | cr lf` denotes either a carriage return, or a line feed, or a carriage return followed by a line feed. The “`|`” operator denotes *alternation*, which means selecting a string from either of two sets.

Notice that six of our tokens have the same name as terminal symbols: `';'`, `'('`, `')'`, `'+'`, `'-'`, and `'number'`. When a token has the same name as a terminal symbol, Invisible Jacc automatically configures the scanner so that whenever the scanner recognizes the token, the scanner generates the corresponding terminal symbol.

Two of our tokens do not have the same name as any terminal symbol: `'whiteSpace'` and `'lineEnd'`. Invisible Jacc lets you define what happens when these tokens are recognized. In this example, we discard `'whiteSpace'` tokens. When a `'lineEnd'` token is recognized, we increment the current line number and then discard the token.

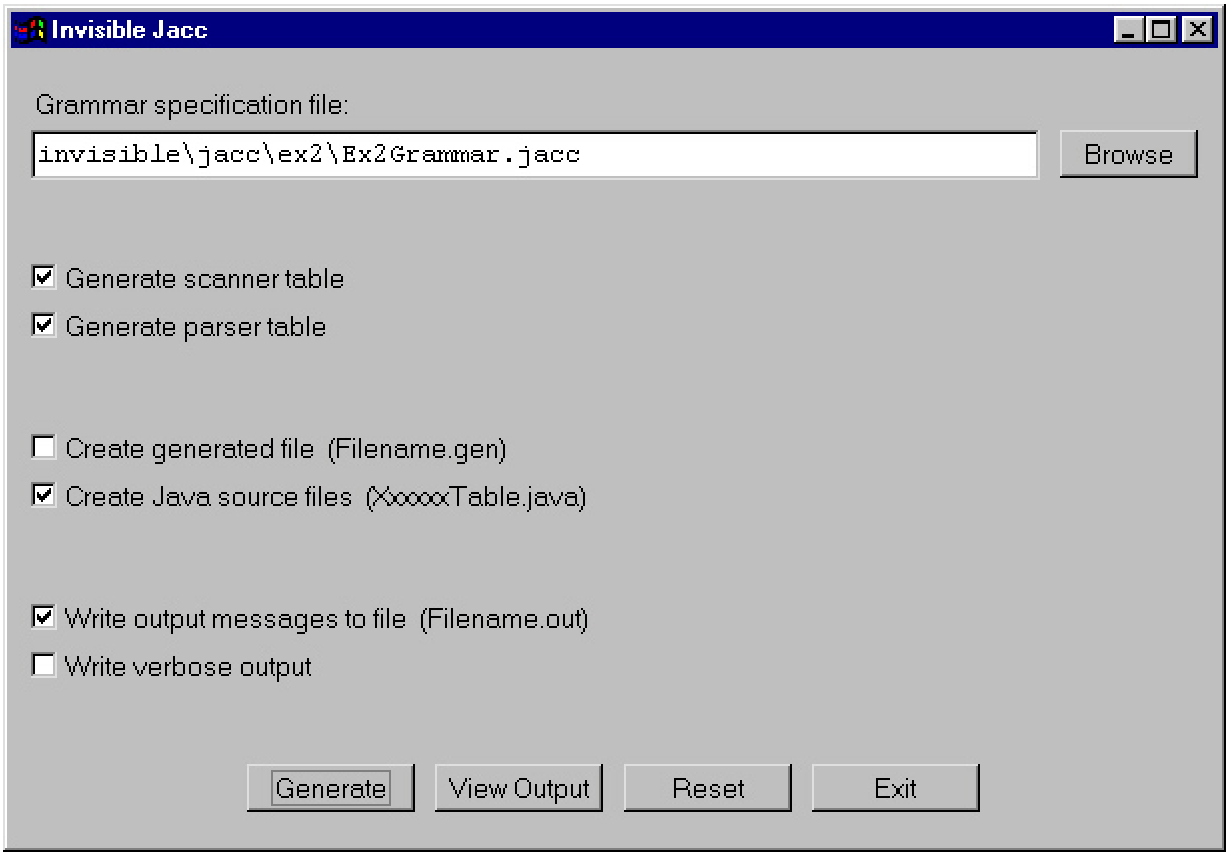
2.2 Generating the Tables

After writing the grammar specification, the next step is to use Invisible Jacc to generate the scanner table and the parser table. The tables contain finite-state machines that allow the scanner and parser to do their job very efficiently.

There are two ways to generate the tables. You can use the Invisible Jacc graphical user interface, or you can use the Invisible Jacc command line.

2.2.1 The Invisible Jacc Graphical User Interface

The class `invisible.jacc.gen.GenGUI` provides the graphical user interface. To generate the tables for our example, execute class `invisible.jacc.gen.GenGUI`. The following window appears:


Invisible Jacc

Grammar specification file:

invisible\jacc\ex2\Ex2Grammar.jacc

[✓] Generate scanner table

[✓] Generate parser table

[ ] Create generated file (Filename.gen)

[✓] Create Java source files (XxxxxTable.java)

[✓] Write output messages to file (Filename.out)

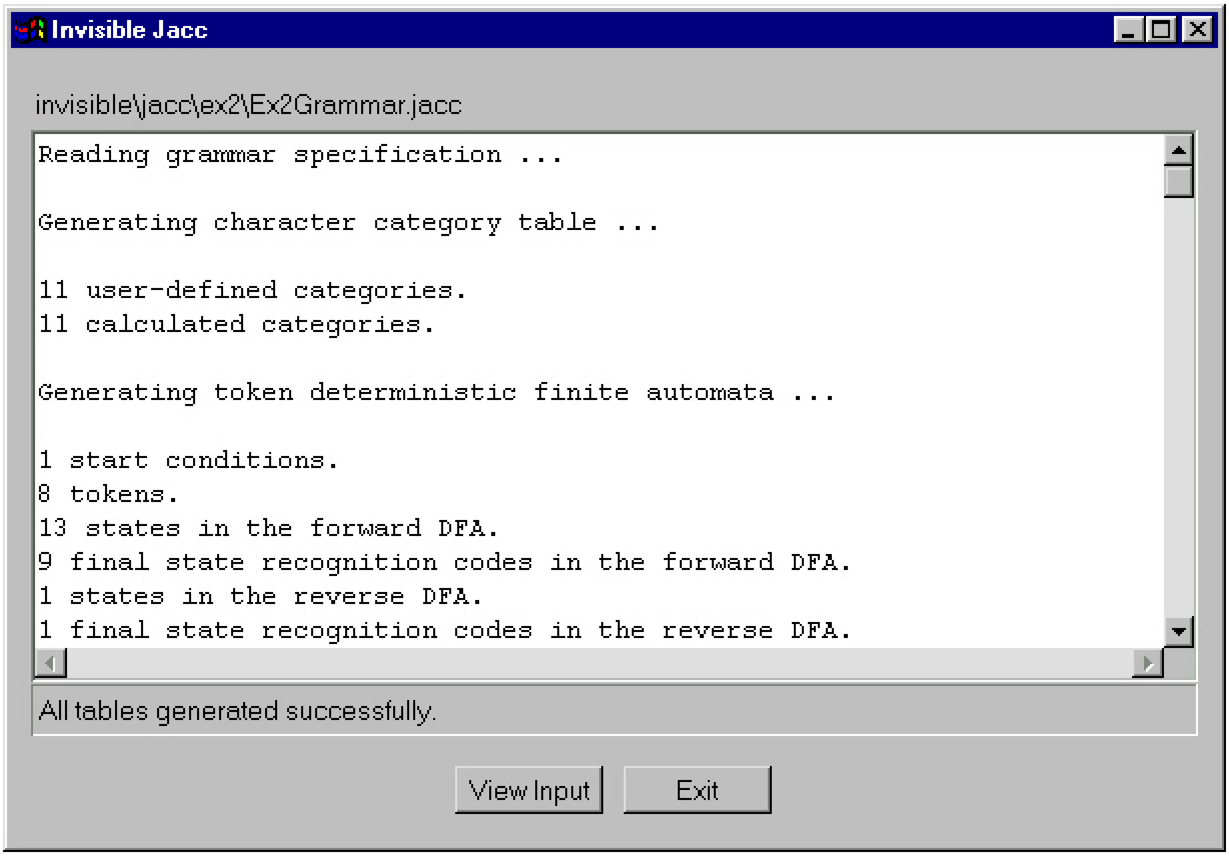
[ ] Write verbose output

1. Enter the grammar specification filename: `invisible\jacc\ex2\Ex2Grammar.jacc`. Or, click on **Browse** to bring up a file selection dialog, and then select the file.
2. Check the boxes labeled **Generate scanner table**, **Generate parser table**, **Create Java source files**, and **Write output messages to file**.
3. Then, click on **Generate**.

Invisible Jacc reads input from file `Ex2Grammar.jacc` and writes the following three output files:

- `Ex2Grammar.out` contains a summary of the grammar and the finite state machines. If any errors are encountered, this file also contains the error messages. If you click on **Write verbose output**, this file will contain a detailed description of the grammar and the finite state machines.
- `Ex2GrammarScannerTable.java` contains the scanner table, in the form of a Java source file.
- `Ex2GrammarParserTable.java` contains the parser table, in the form of a Java source file.

The summary of the grammar and the finite state machines is also displayed on-screen as shown below:



invisible\jacc\ex2\Ex2Grammar.jacc

```
Reading grammar specification ...

Generating character category table ...

11 user-defined categories.
11 calculated categories.

Generating token deterministic finite automata ...

1 start conditions.
8 tokens.
13 states in the forward DFA.
9 final state recognition codes in the forward DFA.
1 states in the reverse DFA.
1 final state recognition codes in the reverse DFA.
```

All tables generated successfully.

You can use the **View Input** and **View Output** buttons to switch back and forth between the input and output screens. The **Reset** button clears the input screen so you can enter new values. Finally, click **Exit** to exit from Invisible Jacc.

2.2.2 The Invisible Jacc Command Line

The class `invisible.jacc.gen.GenMain` provides a simple command-line interface to the generator. To generate the tables for our example, execute class `invisible.jacc.gen.GenMain` with the following command line:

```
-o -j invisible\jacc\ex2\Ex2Grammar.jacc
```

This command reads input from file `Ex2Grammar.jacc` and writes the following three output files:

- `Ex2Grammar.out` contains a summary of the grammar and the finite state machines. If any errors are encountered, this file also contains the error messages. If you add the option `–v` to the command line, this file will contain a detailed description of the grammar and the finite state machines.
- `Ex2GrammarScannerTable.java` contains the scanner table, in the form of a Java source file.
- `Ex2GrammarParserTable.java` contains the parser table, in the form of a Java source file.

Here is the file `Ex2Grammar.out` produced in this example:

```
Reading grammar specification ...
```

```
Generating character category table ...

11 user-defined categories.
11 calculated categories.

Generating token deterministic finite automata ...

1 start conditions.
8 tokens.
13 states in the forward DFA.
9 final state recognition codes in the forward DFA.
1 states in the reverse DFA.
1 final state recognition codes in the reverse DFA.

Generating LALR(1) configuration finite state machine ...

13 symbols.
10 productions.
9 LALR(1) machine states.

Writing scanner table Java source ...
Writing parser table Java source ...

All tables generated successfully.
```

If you want to see the other two files, you can find them in directory `invisible\jacc\ex2`.

2.3 Writing the Compiler

After generating the scanner and parser tables, the next step is to write the compiler.

The compiler is written as a Java class. Invisible Jacc supplies the scanning and parsing code, so the compiler class only needs to supply code that is specific to the language. In particular, the compiler needs to supply token factories and nonterminal factories. A *token factory* is called by the scanner whenever it recognizes a token. A *nonterminal factory* is called by the parser whenever it reduces a production.

(If you are familiar with the Unix programs Lex and Yacc, the token factories are equivalent to the actions specified in a Lex file, and the nonterminal factories are equivalent to the actions specified in a Yacc file.)

Here is the compiler for our example. You can find this code in `Ex2Compiler.java`. In this printed copy, we have removed some of the comments to make the file shorter.

```
package invisible.jacc.ex2;

import java.io.IOException;

import invisible.jacc.parse.CompilerModel;
import invisible.jacc.parse.NonterminalFactory;
import invisible.jacc.parse.Parser;
import invisible.jacc.parse.Scanner;
import invisible.jacc.parse.SyntaxException;
import invisible.jacc.parse.Token;
import invisible.jacc.parse.TokenFactory;
```

```
public class Ex2Compiler extends CompilerModel
{

   // This flag enables the use of debug token and nonterminal factories.

   static final boolean _debug = false;


   // Constructor must create the scanner and parser tables.

   public Ex2Compiler ()
   {
      super();

      // Get our scanner table

      _scannerTable = new Ex2GrammarScannerTable ();

      // Link the token factories to the scanner table

      _scannerTable.linkFactory ("number", "", new Ex2Number ());
      _scannerTable.linkFactory ("lineEnd", "", new Ex2LineEnd ());

      // Get our parser table

      _parserTable = new Ex2GrammarParserTable ();

      // Link the nonterminal factories to the parser table

      _parserTable.linkFactory ("Statement", "", new Ex2Statement ());
      _parserTable.linkFactory ("Expression", "add",
         new Ex2ExpressionAdd ());
      _parserTable.linkFactory ("Expression", "subtract",
         new Ex2ExpressionSubtract ());
      _parserTable.linkFactory ("Primary", "paren",
         new Ex2PrimaryParen ());

      // If debug mode, activate debugging features.  This checks the
      // scanner and parser tables for internal consistency, and checks
      // all the strings passed to the linkFactory functions to ensure
      // they match names in the grammar specification.  It also installs
      // tracing code, which outputs a message on every call to a token
      // factory or nonterminal factory.

      if (_debug)
      {
         if (setDebugMode (true, true))
         {
            throw new InternalError (
                  "Ex2Compiler: Consistency check failed.");
         }
      }

      // Done

      return;
   }
```

```
    // This is a simple front end that lets you invoke the compiler from
    // the command line.  The command line can contain the name of one or
    // more source files.  The compiler is invoked on each source file.

    public static void main (String[] args) throws Exception
    {

       // Create the compiler object

       Ex2Compiler compiler = new Ex2Compiler ();

       // For each filename listed on the command line ...

       for (int i = 0; i < args.length; ++i)
       {

          // Print the filename

          System.out.println ();
          System.out.println ("Compiling " + args[i] + " ...");

          // Compile the file

          compiler.compile (args[i]);
       }

       return;
    }


 // ----- Token Factories -----


 /*->

   The following classes define the token factories for this compiler.

   When the scanner recognizes a token, it calls the makeToken() entry
   point in the corresponding token factory.  The token factory can do one
   of three things:  (a) assemble the token by filling in the fields of the
   Token object;  (b) discard the token;  or (c) reject the token.
   (Rejecting is only used in special cases;  none of the token factories
   in this example ever reject a token.)

   The scanner supplies a Token object initialized as follows:

    token.number = token parameter (from the grammar specification)

    token.value = null

    token.file = current filename

    token.line = current line number

    token.column = current column number
```

```
  In most cases, the token parameter is the numerical value of the
  terminal symbol that corresponds to the token.

  You will notice that not every token listed in the grammar specification
  has a token factory.  Tokens without a token factory are handled by the
  default token factory.  The default token factory does the following:
  (a) if the parameter is zero, it discards the token;  and (b) if the
  parameter is nonzero, it assembles a token whose number equals the
  parameter, and whose value is null (that is, it leaves the Token object
  unchanged).

->*/


// Token factory class for number.
//
// The value of a number is an Integer object, or null if there was a
// conversion error.

final class Ex2Number extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Get the value as a string

      String numString = scanner.tokenToString ();

      // Convert it to an Integer object

      try
      {
         token.value = new Integer (Integer.parseInt (numString, 10));
      }

      // If conversion error, print error message and return null

      catch (NumberFormatException e)
      {
         reportError (token, null,
            "Invalid number '" + numString + "'." );
         token.value = null;
      }

      // Assembled token

      return assemble;
   }
}


// Token factory class for lineEnd.
//
// A lineEnd is discarded after counting the line.
```

```
final class Ex2LineEnd extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Bump the line number

      scanner.countLine ();

      // Discard token

      return discard;
   }
}


// ----- Nonterminal Factories -----


/*->

  The following classes define the nonterminal factories for this
  compiler.

  When the parser reduces a production, it calls the makeNonterminal()
  entry point for the production's nonterminal factory.  The nonterminal
  factory must return the value of the nonterminal symbol on the
  production's left hand side.  The parser saves the returned value on the
  value stack.

  The parser supplies a parameter which is the production parameter from
  the grammar specification.  In this example, the parameter is always
  zero.

  When a nonterminal factory reads a terminal symbol T from the
  production's right hand side, the nonterminal factory must be prepared
  to handle the possibility that the value of T could be null.  This is
  true even if the token factory for T never returns a null value.  The
  reason is that during error repair, the parser may insert terminal
  symbols that are not present in the source file, and these error
  insertions always have null value.

  On the other hand, when a nonterminal factory reads a nonterminal symbol
  S from the production's right hand side, the value of S is always the
  result of calling the nonterminal factory for S.  If the nonterminal
  factory for S never returns null, then the value of S is never null.

  You will notice that not every production listed in the grammar
  specification has a nonterminal factory.  Productions without a
  nonterminal factory are handled by the default nonterminal factory.  The
  default nonterminal factory does the following:  (a) if the production
  has a non-empty right hand side, it returns the value of the first item
  on the right hand side;  (b) if the production has an empty right hand
  side, it returns null.

->*/
```

```
// Nonterminal factory class for Statement.
//
// A Statement has null value.
//
// When this nonterminal factory is called, it prints the value of the
// expression on the standard output.

final class Ex2Statement extends NonterminalFactory
{
   public Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException
   {

      // Get items from value stack

      Integer value = (Integer) parser.rhsValue (0);

      // If value is an error insertion, do nothing

      if (value == null)
      {
         return null;
      }

      // Print the value

      System.out.println (value.intValue());

      // Return null value

      return null;
   }
}


// Nonterminal factory class for Expression{add}.
//
// The value of an Expression is an Integer object.  It is null if there
// is an error.

final class Ex2ExpressionAdd extends NonterminalFactory
{
   public Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException
   {

      // Get items from value stack

      Integer leftOperand = (Integer) parser.rhsValue (0);
      Integer rightOperand = (Integer) parser.rhsValue (2);

      // If either operand is an error insertion, return null to indicate
      // error
      if ((leftOperand == null) || (rightOperand == null))
      {
         return null;
      }

      // Return the sum of the two operands

      return new Integer (
         leftOperand.intValue() + rightOperand.intValue() );
   }
}


// Nonterminal factory class for Expression{subtract}.
//
// The value of an Expression is an Integer object.  It is null if there
// is an error.

final class Ex2ExpressionSubtract extends NonterminalFactory
{
   public Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException
   {

      // Get items from value stack

      Integer leftOperand = (Integer) parser.rhsValue (0);
      Integer rightOperand = (Integer) parser.rhsValue (2);

      // If either operand is an error insertion, return null to indicate
      // error

      if ((leftOperand == null) || (rightOperand == null))
      {
         return null;
      }

      // Return the difference of the two operands

      return new Integer (
         leftOperand.intValue() - rightOperand.intValue() );
   }
}


// Nonterminal factory class for Primary{paren}.
//
// The value of a Primary is an Integer object.  It is null if there is
// an error.

final class Ex2PrimaryParen extends NonterminalFactory
{
   public Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException
   {

      // Return the operand in parentheses
```

```
        return parser.rhsValue (1);
    }
  }


  }  // End class Ex2Compiler
```

In this example there are seven classes. `Ex2Compiler` is the main compiler class. There are two token factory classes: `Ex2Number` and `Ex2LineEnd`. There are four nonterminal factory classes: `Ex2Statement`, `Ex2ExpressionAdd`, `Ex2ExpressionSubtract`, and `Ex2PrimaryParen`. We will discuss each class in turn.

Before beginning, notice that each of the six factory classes is implemented as an inner class, nested inside the main compiler class. This allows each factory class to access variables and methods in the main compiler class.

As a matter of coding style, we have elected not to indent the inner classes. This is because inner classes often contain the bulk of a compiler's code, and there seems to be no gain in clarity by having an extra level of indentation.

Also, note that the use of inner classes does not prevent the compiler from running on a Java 1.0 VM. You need to have a Java 1.1 development system to write inner classes, but the resulting Java class files work on any Java VM.

2.3.1 The Main Compiler Class

The main compiler class is `Ex2Compiler`. It begins by defining a `boolean` variable `_debug,` which can be made `true` to activate debugging code.

Next, there is the constructor `Ex2Compiler`. The constructor performs the following steps:

1. Invoke the `super` method to initialize the superclass, in this case `CompilerModel`.
2. Create a scanner table by creating a new instance of class `Ex2GrammarScannerTable`. (Recall that `Ex2GrammarScannerTable` was generated by Invisible Jacc.)
3. Link the token factories to the scanner table by calling the `linkFactory` method. This attaches each token factory to one or more tokens recognized by the scanner. The first argument to `linkFactory` is a string containing the token name. The second argument to `linkFactory` is a string containing the link name. Notice that both strings are case-sensitive. The third argument to `linkFactory` is an object of type `TokenFactory`. In our example:
   - The token factory `Ex2Number` is attached to the token named “`number`”. There is no link name, so the link name is written as an empty string.
   - The token factory `Ex2LineEnd` is attached to the token named “`lineEnd`”. There is no link name, so the link name is written as an empty string.
4. Create a parser table by creating a new instance of class `Ex2GrammarParserTable`. (Recall that `Ex2GrammarParserTable` was generated by Invisible Jacc.)
5. Link the nonterminal factories to the parser table by calling the `linkFactory` method. This attaches each nonterminal factory to one or more productions that are recognized by the parser. The first argument to `linkFactory` is a string containing the production’s left hand side. The second argument to `linkFactory` is a string containing the link name. Notice that both strings are case-sensitive. The third argument to `linkFactory` is an object of type `NonterminalFactory`. In our example:
   - The nonterminal factory `Ex2Statement` is attached to the production with left hand side “`Statement`”. This production has no link name, so the link name is written as an empty string.

- The nonterminal factory `Ex2ExpressionAdd` is attached to the production with left hand side "`Expression`" and link name "`add`". Recall that in the grammar specification, the link name is written between curly braces.
- The nonterminal factory `Ex2ExpressionSubtract` is attached to the production with left hand side "`Expression`" and link name "`subtract`".
- The nonterminal factory `Ex2PrimaryParen` is attached to the production with left hand side "`Primary`" and link name "`paren`".

6. If debugging mode is enabled, use the `setDebugMode` method to check the scanner and parser tables, and throw an exception if there is an error. The `setDebugMode` method also enables tracing. When tracing is enabled, the scanner prints a message every time it recognizes a token, and the parser prints a message every time it reduces a production. Tracing can be helpful in debugging a grammar specification.

Finally, there is the method `main`, which is a simple front-end that lets you invoke the compiler from the command line. It accepts a list of file names on the command line. The `main` method does the following:

1. Create a new instance of class `Ex2Compiler`.
2. For each file on the command line, print the name of the file and then invoke the `compile` method to compile the file. (Note that the `compile` method is inherited from class `CompilerModel`.)

2.3.2 The Token Factory Classes

We now turn our attention to the token factory classes. There are two token factory classes in our example, `Ex2Number` and `Ex2LineEnd`. Each token factory class is attached to one of the tokens defined in the grammar specification file.

A token factory class must be a subclass of `TokenFactory`. Each token factory class must override the method `makeToken` to include code which processes the token. Whenever the scanner recognizes a token, it calls the `makeToken` method of the corresponding token factory, and passes the following two parameters:

- The `Scanner` object. The token factory can use this parameter to obtain the input text that matched the regular expression, and to call other functions provided by the scanner.
- A `Token` object. This object has five fields, which the scanner initializes as follows:

  `token.number` = an `int` which contains the token parameter.

  `token.value` = an `Object` which contains `null`.

  `token.file` = a `String` which contains the current filename.

  `token.line` = an `int` which contains the current line number.

  `token.column` = an `int` which contains the current column number.

If the token name is the same as the name of a terminal symbol, then the token parameter is the numerical value of the terminal symbol (which is a positive integer). If the token name is not the same as the name of a terminal symbol, then the token parameter is taken from the grammar specification file, and defaults to zero if not specified. In our example, the six tokens `';'`, `'('`, `')'`, `'+'`, `'-'`, and `'number'` each have a nonzero parameter which equals the numerical value of the corresponding terminal symbol. The two tokens `'whiteSpace'` and `'lineEnd'` have a zero parameter.

The token factory can do three things: assemble a token, discard a token, or reject a token. If the token factory assembles a token, the token is passed to the parser as a terminal symbol. If the token factory discards a token, the scanner continues scanning the input to find the next token. Rejecting a token is used for special cases, and is not discussed here. The `makeToken` method indicates which action to take by returning one of the three values `assemble`, `discard`, or `reject`, all of which are defined in the class `TokenFactory`.

If the token factory assembles a token, it can also supply a *value* for the token. The value can be any object. The token factory assigns a value to the token by storing the value into the `token.value` field. This value can be retrieved by a nonterminal factory.

If some error occurs in calculating a token's value, the token factory should assemble a token with `token.value` set to `null` (after printing an error message). It should not discard the token, because that would interfere with the parser's ability to correctly parse the input. As we will see later, the parser's error repair algorithm can insert terminal symbols into the input, and these *error-insertion symbols* always have `null` value. So, setting `token.value` to `null` is used uniformly throughout Invisible Jacc as a signal that an error occurred.

In our example, the token factory `Ex2Number` is called by the scanner whenever the token `'number'` is recognized. The token factory performs the following steps:

1. Convert the token text to a `String` object by calling the method `scanner.tokenToString`.
2. Use the method `Integer.parseInt` to convert the `String` object into an `Integer` object that contains the numerical value of the number.
3. Assemble a token whose value is the resulting `Integer` object.
4. If `Integer.parseInt` throws a `NumberFormatException`, then print an error message and assemble a token whose value is `null`.

The token factory `Ex2LineEnd` is called by the scanner whenever the token `'lineEnd'` is recognized. The token factory performs the following steps:

1. Increment the current line number by calling the method `scanner.countLine`.
2. Discard the token.

Notice that our grammar specification includes eight tokens, but we have supplied only two token factories. For the other six tokens, Invisible Jacc automatically supplies a *default token factory*. In this example, the default token factory does the following:

- For tokens `';'`, `'('`, `')'`, `'+'`, and `'-'`, the default token factory assembles a token with `null` value. In this case, the `null` value does not indicate an error; rather, it simply indicates that these tokens don't have any values associated with them.
- For token `'whiteSpace'`, the default token factory discards the token. It does this because the token parameter is zero. (If a token doesn't have the same name as a terminal symbol, and there is no parameter specified in the grammar specification file, then the parameter defaults to zero.)

2.3.3 The Nonterminal Factory Classes

We now turn our attention to the nonterminal factory classes. There are four nonterminal factory classes in our example, `Ex2Statement`, `Ex2ExpressionAdd`, `Ex2ExpressionSubtract`, and `Ex2PrimaryParen`. Each nonterminal factory class is attached to one of the productions defined in the grammar specification file.

A nonterminal factory class must be a subclass of `NonterminalFactory`. Each nonterminal factory class must override the method `makeNonterminal` to include code which calculates the value of the production's left hand side. Whenever the parser reduces a production, it calls the `makeNonterminal` method of the corresponding nonterminal factory, and passes the following two parameters:

- The `Parser` object. The nonterminal factory can use this parameter to obtain the values of symbols on the right hand side of the production, and to call other functions provided by the parser.
- An `int` parameter. This is the production parameter from the grammar specification file. In our example, the grammar specification file does not include production parameters, so they all default to zero.

The `makeNonterminal` method must return the value of the nonterminal symbol on the production's left hand side. The value can be any object, and it can be `null`.

In our example, the nonterminal factory `Ex2Statement` is called by the parser whenever it reduces the production `Statement -> Expression ';'`. The nonterminal factory performs the following steps:

1. Call the method `parser.rhsValue(0)` to obtain the value of the first symbol on the right hand side. In this case, the value is a `Integer` object that contains the value of an Expression.
2. If the value is `null` (which indicates that an error occurred in evaluating the expression), do nothing.
3. Otherwise, print the value of the Expression on the standard output.
4. Return `null`, since a Statement has no value.

The nonterminal factory `Ex2ExpressionAdd` is called by the parser whenever it reduces the production `Expression {add} -> Expression '+' Primary`. The nonterminal factory performs the following steps:

1. Call the method `parser.rhsValue(0)` to obtain the value of the first symbol on the right hand side. In this case, the value is a `Integer` object that contains the value of an Expression.
2. Call the method `parser.rhsValue(2)` to obtain the value of the third symbol on the right hand side. In this case, the value is a `Integer` object that contains the value of a Primary.
3. If either value is `null` (which indicates that an error occurred), do nothing and return `null`.
4. Otherwise, add the two values, and return a new `Integer` object that contains the result.

The nonterminal factory `Ex2ExpressionSubtract` is called by the parser whenever it reduces the production `Expression {subtract} -> Expression '-' Primary`. It is the same as `Ex2ExpressionAdd`, except that it subtracts the two values instead of adding them.

The nonterminal factory `Ex2PrimaryParen` is called by the parser whenever it reduces the production `Primary {paren} -> '(' Expression ')'`. The nonterminal factory performs the following steps:

1. Call the method `parser.rhsValue(1)` to obtain the value of the second symbol on the right hand side. In this case, it is the value of an Expression, and it can be either an `Integer` object or `null`.
2. Return the resulting value.

Notice that our grammar specification includes nine productions, but we have supplied only four nonterminal factories. For the other five productions, Invisible Jacc automatically supplies a *default nonterminal factory*. The default nonterminal factory returns the value of the first symbol on the production's right hand side. (If the right hand side is empty, the default nonterminal factory returns `null`.) In this example, there are only two productions where this matters:

- For the production `Expression {primary} -> Primary`, the default nonterminal factory returns the value of the Primary, which can be either an `Integer` object or `null`.
- For the production `Primary {number} -> number`, the default nonterminal factory returns the value of the number, which can be either an `Integer` object or `null`.

For the remaining three productions, the return value is always `null`.

2.4 Running the Compiler

After writing the compiler, you can run it. We have included a sample input file called `Ex2Input.txt`. Here it is, with some of the comments removed:

```
// Try some expressions to show that parentheses work, and that addition
// and subtraction group from left to right.

7+11;               // should be 18
```

```
7-11;              // should be -4

4-5-6;                // should be (4-5)-6 = -7
(4-5)-6;           // should be -7
4-(5-6);           // should be 5

100-(45+12)+123;      // should be 166


// Try an invalid number.  9876543210 is too big to be an int, so the
// compiler should issue an error message.  (Note that early versions of
// Microsoft's Java VM don't produce an error;  they just reduce the
// number modulo 2^32, yielding the value 1286608618.)

9876543210;           // should be error


// Demonstrate how error repair works.  We'll intentionally leave off the
// semicolon at the end of a statement.  The compiler should automatically
// insert the semicolon and process the statement.

10+12-6               // missing semicolon, should be 16
18-3+44;           // should be 59


// This time we'll leave off a right parenthesis.  The compiler should
// automatically insert the parenthesis.

23-(9-5;           // missing right parenthesis, should be 19
345+11-46;            // should be 310

// This time we'll put in an extra right parenthesis.  The compiler should
// simply remove it.

82-12-7);             // extra right parenthesis, should be 63
364+89-1023;          // should be -570
```

The resulting output is:

```
Compiling invisible\jacc\ex2\Ex2Input.txt ...
18
-4
-7
-7
5
166
invisible\jacc\ex2\Ex2Input.txt(39,1): error: Invalid number '9876543210'.
invisible\jacc\ex2\Ex2Input.txt(47,3): error: Expected ';'.
16
59
invisible\jacc\ex2\Ex2Input.txt(53,9): error: Expected ')'.
19
310
invisible\jacc\ex2\Ex2Input.txt(59,9): error: Unexpected ')'.
```

```
63
-570
```

There are three things to notice. First, it works! This compiler actually does parse expressions and correctly evaluate them.

Second, the token factory class `Ex2Number` is able to detect an invalid number in the input and produce an appropriate error message. (Some early implementations of the Java VM don't produce the error message, but instead reduce the number modulo 2 to the 32 power. So it is possible you won't get the error message if you are using an older Java VM.)

Third, this compiler has error repair! In our input we deliberately put in three syntax errors. In each case, the compiler reported the error, made a reasonable repair, and continued parsing. Notice how this required no effort on our part. The only thing we needed to do was to make sure that every nonterminal factory can handle a `null` value for any terminal symbol on the right hand side.

Each error is reported along with the input filename, line number, and column number. This shows how the compiler is able to keep track of where each token came from. The token factory class `Ex2LineEnd` is responsible for counting the lines as the input is scanned. (Line numbers don’t match up with this printed copy of the input file because we deleted some comment lines. They match up correctly with the actual input file.)

The ability to provide error repair, automatically, without any changes to the grammar or special coding, is one of the strengths of Invisible Jacc.

2.5 How It Works

It is instructive to take a few minutes and see how the scanner and parser do their job. Let’s keep things simple by considering a one-line program:

```
23+7-12;
```

To make this even more explicit, let’s write <LF> for a line-feed character, and <EOF> for end-of-file. Then, our one-line program becomes:

```
23+7-12;<LF><EOF>
```

We will first explain how the scanner breaks up the input into tokens, and then how the parser structures the tokens into productions.

2.5.1 How the Scanner Works

The function of the scanner is to break the input up into tokens. At any given point in the input, the scanner finds the longest string that matches any of the token definitions. For convenience, let’s repeat our token definitions:

```
';' = ';';
'(' = '(';
')' = ')';
'+' = '+';
'-' = '-';
number = decDigit+;
whiteSpace = space* ('/' '/' notEol*)?;
lineEnd = cr | lf | cr lf;
```

The following table shows how the scanner applies these token definitions to our sample input.

Remaining Input	**Matching Token**	**Token Factory Action**
`23+7-12;<LF><EOF>`	`number(23)`	`assemble`
`+7-12;<LF><EOF>`	`'+'`	`assemble`
`7-12;<LF><EOF>`	`number(7)`	`assemble`
`-12;<LF><EOF>`	`'-'`	`assemble`
`12;<LF><EOF>`	`number(12)`	`assemble`
`;<LF><EOF>`	`';'`	`assemble`
`<LF><EOF>`	`lineEnd`	`discard`
`<EOF>`	`%%EOF`	`-`

You can see how the scanner splits up the input into eight tokens. The ‘lineEnd’ token is discarded by the token factory, so the remaining seven tokens are the output from the scanner.

For convenience, we have written ‘number(23)’ to denote a ‘number’ token whose value is an `Integer` object containing 23. Also, we have written ‘%%EOF’ to denote the end-of-file token.

2.5.2 How the Parser Works

The function of the parser is to organize the tokens into productions. For convenience, let’s repeat our production definitions:

```
0:  Goal -> StatementList;
1:  StatementList ->;
2:  StatementList -> StatementList Statement;
3:  Statement -> Expression ';';
4:  Expression -> Primary;
5:  Expression -> Expression '+' Primary;
6:  Expression -> Expression '-' Primary;
7:  Primary -> number;
8:  Primary -> '(' Expression ')';
```

For convenience, we have numbered the productions from 0 through 8. Note that these numbers do not appear in the grammar specification file. We have also omitted the link names.

We showed how the scanner breaks up the input into the following stream of seven tokens:

```
number(23)  '+'  number(7)  '-'  number(12)  ';'  %%EOF
```

Now let’s see how the parser structures them. The parser maintains a *stack*. Each entry on the stack holds the value of one symbol, which can be either a terminal symbol or a nonterminal symbol. The value can be any Java object, or it can be `null`.

At any given point in the input, the parser can do one of two things:

- The parser can *shift* the next input symbol. In this case, the value of the symbol is pushed onto the parser’s stack.
- The parser can *reduce* a production. In this case, the top elements of the stack are the values of the production’s right hand side. The parser pops these values off the stack, and then pushes the value of the production’s left hand side.

The following table shows how the parser’s stack works in this example.

Parser's Stack	**Action**
	`reduce 1`
`StatementList`	`shift number(23)`
`StatementList  number(23)`	`reduce 7`
`StatementList  Primary(23)`	`reduce 4`
`StatementList  Expression(23)`	`shift '+'`
`StatementList  Expression(23)  '+'`	`shift number(7)`
`StatementList  Expression(23)  '+'  number(7)`	`reduce 7`
`StatementList  Expression(23)  '+'  Primary(7)`	`reduce 5`
`StatementList  Expression(30)`	`shift '-'`
`StatementList  Expression(30)  '-'`	`shift number(12)`
`StatementList  Expression(30)  '-'  number(12)`	`reduce 7`
`StatementList  Expression(30)  '-'  Primary(12)`	`reduce 6`
`StatementList  Expression(18)`	`shift ';'`
`StatementList  Expression(18)  ';'`	`reduce 3`
`StatementList  Statement`	`reduce 2`
`StatementList`	`reduce 0`
`Goal`	`shift %%EOF`

Every time the parser shifts a symbol, the symbol is pushed onto the stack. Terminal symbols are shifted in the same order they appear in the input, reading from left to right.

Every time the parser reduces a production, the top symbols on the stack are the production’s right hand side. These symbols are popped off the stack. Then, the symbol on the production’s left hand side is pushed onto the stack.

Notice how the end result of the parse is to reduce the entire input to the goal symbol.

3 Invisible Jacc Grammar Specification

The Invisible Jacc grammar specification is used to describe a language. There are two main parts: A set of regular expressions defines *tokens* that are recognized by the scanner. Then, a set of productions defines a *grammar* that is recognized by the parser.

3.1 Lexical Structure

An Invisible Jacc grammar specification is written in an ASCII text file. By convention, the filename ends in the extension "`.jacc`". The grammar specification is first broken up into tokens. Some of these tokens then go on to become the terminal symbols used in the grammar specification language.

The following sections describe the tokens.

3.1.1 White Space

White space consists of the characters space, tab, and form feed. Also, an ASCII sub character (0x1A) which appears as the last character of the file is considered to be white space. White space is discarded, and has no effect except insofar as it serves to separate other tokens.

3.1.2 Line End

Line end consists of carriage return, line feed, or the pair carriage return followed by line feed. Line ends are counted and then discarded.

3.1.3 Comment

There are two forms of comment:

- The characters `//` followed by all characters up to the next line end.
- An arbitrary string of characters introduced by `/*` and terminated by `*/`.

The first type of comment is called a line comment, and the second type of comment is called a C-style comment. Comments are discarded.

3.1.4 Keyword

A keyword consists of the character `%` followed by a string of letters and digits. The keywords listed below are valid. All other keywords are invalid and will be flagged as errors.

```
%any
%categories
%charsetsize
%conditions
%digit
%goal
%java
%lalr1
%letter
%lowercase
%lr1
%none
%options
%plr1
%productions
%reduce
%repair
%shift
%terminals
%titlecase
%tokens
```

```
%unicode
%uppercase
```

Note: In the keywords `%lr1`, `%plr1`, and `%lalr1`, the last character is the digit one. All other characters appearing in legal keywords are letters.

In the definition of keyword, the term “letter” means any character that is a letter according to the Java method `Character.isLetter`, plus the underscore `'_'` and dollar `'$'` characters. Likewise, the term “digit” means any character that is a digit according to the Java method `Character.isDigit`.

3.1.5 Operator

The following character strings are recognized as operators.

```
-
~
&
@
*
+
?
/
:
;
=
(
)
{
}
|
#
.
->
..
```

3.1.6 Number

There are two forms of number:

- A series of one or more decimal digits, which are interpreted as a decimal number. The decimal digits are:

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- The characters `0x` or `0X`, followed by one or more hexadecimal digits, which are interpreted as a hexadecimal number. The hexadecimal digits are:

  ```
  0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
  ```

3.1.7 Identifier

There are two forms of identifier:

- A letter, followed by a series of zero or more letters and digits.
- A string of characters, introduced by a single quote and terminated by a single quote. The string may contain any characters except space, tab, form feed, carriage return, line feed, and single quote; except that the first and/or last characters of the string may be a single quote.

If an identifier happens to consist of a letter followed by a string of letters and digits, then it can be written either with or without quotes. The quoted and unquoted forms refer to the same identifier, and may be used interchangeably. For example, the identifiers `AB9C` and `'AB9C'` are identical and may be used interchangeably.

The rule that allows a single quote as the first and/or last character of an identifier lets you create identifiers that occur commonly in programming languages. For example:

`'''`	is an identifier consisting of one single quote character.
`''''`	is an identifier consisting of two single quote characters.
`'\''`	is an identifier consisting of a backslash followed by one single quote character.
`'a'b'`	is invalid (because the single quote is neither the first nor the last character of the string).

Notice that there are no escape sequences for characters within an identifier. This is because it is often necessary to define identifiers that are escape sequences in the language being defined, and it would be too confusing to use Invisible Jacc escape sequences to describe the language's escape sequences. With these rules, anything appearing between single quotes is exactly as it appears in the language text.

In the definition of identifier, the term "letter" means any character that is a letter according to the Java method `Character.isLetter`, plus the underscore `'_'` and dollar `'$'` characters. Likewise, the term "digit" means any character that is a digit according to the Java method `Character.isDigit`.

3.2 File Structure

We will use a set of productions to define the Invisible Jacc file structure. *Goal* is the goal symbol of the grammar. The following productions describe the overall file structure:

```
Goal -> SectionList

SectionList -> Section

SectionList -> SectionList Section
```

An Invisible Jacc specification file contains a list of sections. Sections may appear in the file in any order. There are six different types of sections, which are described below. It is legal to have more than one section of a given type.

3.3 The %options Section

The %options section specifies various options that control the parser generator. The following productions describe the %options section:

```
Section -> OptionHeader OptionDefList

OptionHeader -> '%options' ':'

OptionDefList ->

OptionDefList -> OptionDefList OptionDefinition

OptionDefinition -> '%lr1' ';'

OptionDefinition -> '%plr1' ';'

OptionDefinition -> '%lalr1' ';'

OptionDefinition ->
      '%repair' MaxInsertions MaxDeletions ValidationLength ';'

OptionDefinition -> '%charsetsize' CharSetSize ';'

OptionDefinition -> '%goal' Symbol ';'

OptionDefinition -> '%java' JavaName ';'

MaxInsertions -> number

MaxDeletions -> number

ValidationLength -> number

CharSetSize -> number

Symbol -> identifier

JavaName -> JavaIdentifier

JavaName -> JavaName '.' JavaIdentifier

JavaIdentifier -> identifier
```

The %options section consists of a header, followed by a series of option definitions. Each option definition is a statement that is terminated by a semicolon.

3.3.1 The %options Header

The %options section header consists of the keyword `%options` followed by a colon.

3.3.2 The %lr1, %plr1, and %lalr1 Options

There are three options which specify the type of grammar that the parser generator should generate. To generate an LALR(1) grammar, include the following option:

```
%lalr1;
```

To generate a full LR(1) grammar, include the following option:

```
%lr1;
```

To generate an optimized LR(1) grammar, include the following option:

```
%plr1;
```

Note: In the keywords `%lalr1`, `%lr1`, and `%plr1`, the last character is the digit one. The other characters are letters.

The following conditions apply to the three grammar selection options:

- Only one grammar selection option can appear in the file.
- If no grammar selection option is specified, the grammar defaults to LALR(1).

It is outside the scope of this document to describe the technical characteristics of LALR(1) and LR(1) grammars. Refer to references [1] and [2] for this information. However, you should be aware that the LALR(1) grammars are a proper subset of the LR(1) grammars. That is, every LALR(1) grammar is also an LR(1) grammar, but not every LR(1) grammar is LALR(1).

LALR(1) grammars are the most popular type, and also generate the smallest parser tables.

Full LR(1) grammars tend to produce very large parser tables. An LR(1) grammar for a realistic programming language can have tens of thousands of parser states. Unless your grammar is very small, generating a full LR(1) grammar may well exceed the memory capacity of your computer.

Optimized LR(1) grammars are a compromise between LALR(1) and full LR(1). Optimized LR(1) accepts the same grammars as full LR(1). However, the parsing tables produced by optimized LR(1) are not much larger than the tables produced by LALR(1). Optimized LR(1) is an excellent choice for grammars that are “almost LALR(1).”

Technical note: Invisible Jacc uses Pager’s algorithm to construct optimized LR(1) grammars. Pager’s algorithm is described in reference [2].

3.3.3 The %repair Option

When the parser encounters an error in the input, it attempts to repair the error so that it can continue parsing. To repair an error, the parser can delete some symbols from the input and/or insert some symbols into the input. Then, the parser *validates* the repair by making sure it can parse the next few input symbols.

The `%repair` option is used to configure the parser’s error repair algorithm. The `%repair` keyword is followed by three numbers. The first number specifies the maximum number of input symbols that may be inserted during a repair. The second number specifies the maximum number of input symbols that may be deleted during a repair. The third number specifies the number of additional input symbols that are checked to validate the repair.

The following conditions apply to the `%repair` option:

- Only one `%repair` option can appear in the file.
- Each of the three numbers in the `%repair` option can range from 0 to 1000.
- If no `%repair` option is specified, the default values are 100 insertions, 200 deletions, and a validation length of 5.

For example, the following statement sets the maximum number of insertions to 50, the maximum number of deletions to 100, and the validation length to 3:

```
%repair 50 100 3;
```

If you want to completely disable error repair, set all three numbers to zero, as shown below. If you do this, the parser will abandon parsing when it encounters an error. This might be appropriate if your parser is intended to process machine-generated files which should never contain an error.

```
%repair 0 0 0;
```

Technical note: Invisible Jacc uses a modified version of the LeBlanc-Mongiovi algorithm to perform error repair. The LM algorithm is described in reference [2].

3.3.4 The %goal Option

The `%goal` option is used to specify the *goal symbol* of the grammar. The `%goal` keyword is followed by the name of the goal symbol.

The following conditions apply to the `%goal` option:

- Only one `%goal` option can appear in the file.
- If no `%goal` option is specified, the goal symbol is the left hand side of the first production in the file.

The following example sets the goal symbol to `'MyGoal'`:

```
%goal MyGoal;
```

3.3.5 The %charsetsize Option

The `%charsetsize` option is used to specify the size of the input character set recognized by the scanner. The `%charsetsize` keyword is followed by a number which gives the size of the character set.

The following conditions apply to the `%charsetsize` option:

- Only one `%charsetsize` option can appear in the file.
- The number in the `%charsetsize` option can range from 2 to 65536.
- If no `%charsetsize` option is specified, the character set size defaults to 256.

If your scanner is intended to process ASCII input, you will almost always want the character set size to be 256. If your scanner is intended to process Unicode input, you will almost always want the character set size to be 65536.

The following example sets the character set size to 65536, which is the appropriate value for Unicode input. If you make the character set size 65536, your scanner will be able to process either ASCII or Unicode, since it is not an error for the character set size to be “too big.”

```
%charsetsize 0x10000;
```

The scanner only accepts input characters which are less than the character set size. For example, if you make the character set size 128, the scanner will accept input characters 0 through 127. (The current implementation of the scanner throws an exception if it receives an input character that is greater than or equal to the character set size. However, your compiler should not rely on this behavior.)

3.3.6 The %java Option

The `%java` option is used to specify the *Java name* of the grammar. The `%java` keyword is followed by a fully-qualified Java package and class name.

You must include the `%java` option if you intend to save the scanner and parser tables in the form of Java source files. Invisible Jacc uses the grammar's Java name to determine the package and class names for the saved scanner and parser tables. If you don't intend to save the tables as Java source files, then you can omit the `%java` option.

The following conditions apply to the `%java` option:

- Only one `%java` option can appear in the file.
- If no `%java` option is specified, the Java name is undefined.

The following example sets the Java name to `'MyApp.Pack2.MyGrammar'`:

```
 %java MyApp.Pack2.MyGrammar;
```

The Java class name should agree with the name of the grammar specification file. In the above example, the grammar specification file should be `MyGrammar.jacc`. Invisible Jacc warns you if the Java class name does not agree with the grammar specification file name.

In the above example, the scanner table would be saved as Java class `MyGrammarScannerTable`. The parser table would be saved as Java class `MyGrammarParserTable`. Both classes would be in the Java package `MyApp.Pack2`.

3.4 The %terminals Section

The %terminals section lists the terminal symbols of the grammar. The following productions describe the %terminals section:

```
Section -> TerminalHeader TerminalDefList

TerminalHeader -> '%terminals' ':'

TerminalDefList ->

TerminalDefList -> TerminalDefList TerminalDefinition

TerminalDefinition -> Symbol ';'

TerminalDefinition -> Symbol InsertionCost DeletionCost ';'

Symbol -> identifier

InsertionCost -> number

DeletionCost -> number
```

The %terminals section consists of a header, followed by a series of terminal definitions. Each terminal definition is a statement that is terminated by a semicolon.

3.4.1 The %terminals Header

The %terminals section header consists of the keyword `%terminals` followed by a colon.

3.4.2 Terminal Definition

A terminal definition can be written in either of two forms. The first form consists of just a terminal symbol. The second form consists of a terminal symbol followed by two numbers, which give the symbol's *insertion cost* and *deletion cost*.

The following conditions apply to terminal definitions:

- Each terminal symbol in the grammar must be listed exactly once in a terminal definition.
- The insertion cost and deletion cost can each range from 1 to 1000. If the costs are omitted, they each default to 1.
- When generating error repair tables, if two terminal symbols have the same insertion cost, then priority is given to the symbol whose first appearance in the file is earliest. (If the %terminals section precedes the %productions section, this will be the symbol that is listed first in the %terminals section.)
- Except as stated in the preceding paragraph, the order in which terminal symbols appear is immaterial.

For example, the following defines a terminal symbol called `'+'`. The insertion and deletion costs default to 1.

```
%terminals:
'+';
```

The following defines a terminal symbol called `'number'`. The insertion cost is 10, and the deletion cost is 20.

```
%terminals:
number 10 20;
```

3.4.3 More About Insertion and Deletion Costs

The insertion and deletion costs are used to tune the parser's error repair algorithm. When the parser encounters an error in the input, it attempts to repair the error so it can continue parsing. A repair consists of deleting some terminal symbols and/or inserting some terminal symbols.

Each repair is assigned a *cost*. The cost of a repair equals the total insertion cost of the symbols that are inserted, plus the total deletion cost of the symbols that are deleted. If several different repairs are possible, the parser attempts to select the repair that has the least cost.

Therefore, a symbol's insertion cost controls the likelihood that the symbol will be inserted during an error repair. A symbol with low insertion cost is more likely to be inserted than a symbol with high insertion cost. Likewise, a symbol's deletion cost controls the likelihood that the symbol will be deleted during an error repair. A symbol with low deletion cost is more likely to be deleted than a symbol with high deletion cost.

The following rules of thumb can be used to assign costs:

- Symbols that are merely punctuation should have low insertion and deletion costs, on the order of 1 or 2. This includes symbols like commas, semicolons, and parentheses.
- Symbols that carry values should have high insertion and deletion costs, on the order of 10 or 20. This includes symbols like identifiers, literal numbers, and literal strings.
- Other symbols should have intermediate costs. This includes symbols like keywords and block delimiters.
- Deletion costs should be higher than insertion costs, because it is usually better to work with the user's input than to discard it. Typically, a symbol's deletion cost would be twice its insertion cost.

Remember, these are just rules of thumb. There is no point in agonizing over insertion and deletion costs, because slight changes in costs are unlikely to have a noticeable effect. Experience shows that the error repair algorithm gives reasonable results with almost any reasonable set of costs.

3.5 The %productions Section

The %productions section lists the productions of the grammar. The following productions describe the %productions section:

```
Section -> ProductionHeader ProductionDefList

ProductionHeader -> '%productions' ':'

ProductionDefList ->

ProductionDefList -> ProductionDefList ProductionDefinition

ProductionDefinition ->
      Symbol LinkName Parameter '->' SymbolList ProductionPrec ';'

Symbol -> identifier

LinkName ->

LinkName -> '{' identifier '}'

Parameter ->

Parameter -> '#' number

SymbolList ->

SymbolList -> SymbolList Symbol

ProductionPrec ->

ProductionPrec -> ProductionPrec '%shift' SymbolSet

ProductionPrec -> ProductionPrec '%reduce' SymbolSet

SymbolSet ->

SymbolSet -> SymbolSet Symbol
```

The %productions section consists of a header, followed by a series of production definitions. Each production definition is a statement that is terminated by a semicolon.

3.5.1 The %productions Header

The %productions section header consists of the keyword `%productions` followed by a colon.

3.5.2 Production Definition

A production definition contains the following six elements:

1. A symbol name. This is called the *left hand side* of the production.
2. An optional *link name*. If included, it is an identifier enclosed in curly braces. The link name is used to determine which nonterminal factory is linked to this production. It has no other significance. (In particular,

it is immaterial whether or not the link name is the same as the name of a symbol.) If the link name is omitted, it defaults to an empty string.

3. An optional *parameter*. If included, it is a number preceded by the # character. If omitted, the parameter defaults to zero. The parameter is passed to the nonterminal factory whenever this production is reduced. The parameter has no other significance.
4. An arrow, which is formed by a minus sign and a greater-than sign.
5. A series of zero or more symbol names. This is called the *right hand side* of the production.
6. Optionally, a series of *precedence clauses*. Each precedence clause consists of the keyword `%shift` or `%reduce`, followed by a series of zero or more symbol names. Precedence clauses are used to resolve shift-reduce conflicts in the grammar. They allow the use of ambiguous grammars.

The six elements must appear in exactly the order listed above.

The following conditions apply to production definitions:

- If there is no `%goal` option, then the left hand side of the first production is used as the goal symbol.
- When generating error repair tables, if two productions have the same cost then priority is given to the production that appears earliest.
- Except as stated in the preceding two paragraphs, the order in which productions appear is immaterial.
- Every production has a *link name*, which is an identifier. The link name, together with the production's left hand side, is used to select which nonterminal factory to use when that production is reduced. If two productions have the same left hand side and the same link name, then the same nonterminal factory is used for both productions.
- Every production has a *parameter*, which is an integer. The parameter is passed to the nonterminal factory whenever the production is reduced. The meaning of the parameter is client-defined. One possible use for the parameter is to distinguish between different productions that have the same nonterminal factory.
- A production may optionally have one or more `%shift` or `%reduce` clauses, appearing after the right hand side. Each `%shift` or `%reduce` clause contains a (possibly empty) list of symbols, which may include both terminal and nonterminal symbols. Listing a nonterminal symbol *X* is equivalent to listing every terminal symbol that could be the first symbol of an expansion of *X*.
- If a shift-reduce conflict is encountered, it is resolved as follows. Let *y* be the input symbol (the "shift") and let *p* be the production (the "reduce"). If *y* appears in a `%shift` clause of *p*, and does not appear in any `%reduce` clause of *p*, then the conflict is resolved in favor of shift. If *y* appears in a `%reduce` clause of *p*, and does not appear in any `%shift` clause of *p*, then the conflict is resolved in favor of reduce. Otherwise, the conflict is not resolved and Invisible Jacc reports an error.
- It is not an error for a terminal symbol *y* to appear in both a `%shift` clause and a `%reduce` clause in the same production, either explicitly or implicitly. ("Explicitly" means that the clause includes *y*. "Implicitly" means that the clause includes a nonterminal symbol *X*, and *y* can be the first symbol of an expansion of *X*.) However, in this case a shift-reduce conflict involving *y* is not resolved, and Invisible Jacc reports an error if such a conflict occurs. The purpose of this rule is to make it easier to use nonterminal symbols in `%shift` and `%reduce` clauses.
- If a reduce-reduce conflict is encountered, Invisible Jacc reports an error. Invisible Jacc does not provide any way to resolve a reduce-reduce conflict. In such a case, you must revise the grammar.
- If the default nonterminal factory is used for a production, the value of the left hand side is determined as follows: (a) If the right hand side is nonempty, the value is the value of the first symbol on the right hand side. (b) If the right hand side is empty, the value is `null`.
- Every nonterminal symbol must appear on the left hand side of at least one production.

- Every symbol (terminal or nonterminal) must be reachable from the goal symbol. In other words, given any terminal or nonterminal symbol *s*, there must be some expansion of the goal symbol that contains *s*. Invisible Jacc reports an error if the grammar does not satisfy this rule.
- Every nonterminal symbol must have an expansion that produces either a string of terminal symbols, or the empty string. Invisible Jacc reports an error if the grammar does not satisfy this rule.

3.5.3 Parser Operation

Let us now describe how the parser makes use of productions. The parser operates using the LR(1) parsing algorithm. A description of the algorithm can be found in references [1] and [2]. The function of the parser is to structure the input according to the grammar's productions. The input to the parser consists of a series of *tokens*. Each token represents one of the terminal symbols of the grammar.

The parser maintains an internal stack, which stores the values of terminal and nonterminal symbols. Each entry in the stack can store an object of type `Object`, or `null`.

During the parse, the parser performs a series of actions. There are only two possible actions: the parser can *shift* a terminal symbol, or it can *reduce* a production.

When the parser shifts a terminal symbol, it reads one token from the input. Recall that the token corresponds to a terminal symbol of the grammar. The parser obtains the symbol's value, and pushes that value onto the parser's stack. (If the input tokens are coming from a scanner, then the token's value is the value supplied by the token factory.)

When the parser reduces a production, the parser calls the nonterminal factory for that production. Suppose that the production is `Y -> X1 ... Xn`. Then, the top `n` elements on the parser's stack are the values of the symbols `X1` through `Xn`. The parser makes the values of `X1` through `Xn` available to the nonterminal factory.

After the nonterminal factory returns, the values of the right hand side are popped off the parser's stack, and the value returned by the nonterminal factory is pushed onto the parser's stack. Thus, the nonterminal factory returns the value of the production's left hand side. As a result, the size of the parser's stack is reduced by `n-1` elements. (If the production has an empty right hand side, so that `n` equals zero, then the parser's stack actually increases in size by one element.)

In a successful parse, the last production reduced is a production with the goal symbol on the left hand side. This means that the entire input has been reduced to the goal symbol.

3.5.4 Example: Using Link Names

Link names are used in attaching nonterminal factories to productions. They let you associate a different nonterminal factory object with each production. Whenever the parser reduces a production, the parser calls the corresponding nonterminal factory.

The following fragment illustrates three productions with the same left hand side and different link names.

```
%productions:
Expression {primary} -> Primary;
Expression {plus} -> Expression '+' Primary;
Expression {minus} -> Expression '-' Primary;
```

The following code fragment shows how to attach a different nonterminal factory class to each of the three productions. The function `linkFactory` (which is defined in class `ParserTable`), accepts three parameters: a string that specifies the production's left hand side, a string that specifies the link name, and an object of class `NonterminalFactory`.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
```

```
    {
       ...
       _parserTable.linkFactory ("Expression", "primary",
          new ExprPrimary ());
       _parserTable.linkFactory ("Expression", "plus",
          new ExprPlus ());
       _parserTable.linkFactory ("Expression", "minus",
          new ExprMinus ());
       ...
    }
    ...

 final class ExprPrimary extends NonterminalFactory
 {
    public Object makeNonterminal (Parser parser, int param)
       throws IOException, SyntaxException
    {
       // Handle Expression -> Primary
       ...
    }
    ...
 }

 final class ExprPlus extends NonterminalFactory
 {
    public Object makeNonterminal (Parser parser, int param)
       throws IOException, SyntaxException
    {
       // Handle Expression -> Expression + Primary
       ...
    }
    ...
 }

 final class ExprMinus extends NonterminalFactory
 {
    public Object makeNonterminal (Parser parser, int param)
       throws IOException, SyntaxException
    {
       // Handle Expression -> Expression - Primary
       ...
    }
    ...
 }
 ...
 }  // End class Compiler
```

3.5.5 Example: Using Parameters

You can assign a parameter to each production. The parameter is passed to the nonterminal factory as an argument to the `makeNonterminal` function. The most common use of the parameter is to distinguish between several different productions with the same nonterminal factory. However, you can use the parameter for any purpose you want.

The following fragment illustrates three productions with the same left hand side and different parameters.

```
%productions:
Expression #0 -> Primary;
Expression #1 -> Expression '+' Primary;
Expression #2 -> Expression '-' Primary;
```

The following code fragment shows how to use a `switch` statement in the nonterminal factory to distinguish between the three productions. Notice that the second argument to `linkFactory` is an empty string, because these productions don't have any link names.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
   {
      ...
      _parserTable.linkFactory ("Expression", "", new Expression ());
      ...
   }
   ...

final class Expression extends NonterminalFactory
{
   public Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException
   {
      switch (param)
      {
      case 0:
         // Handle Expression -> Primary
         ...
      case 1:
         // Handle Expression -> Expression + Primary
         ...
      case 2:
         // Handle Expression -> Expression - Primary
         ...
      default:
         throw new InternalCompilerException ();
      }
   }
   ...
}
...
}  // End class Compiler
```

3.5.6 Example: Left Recursion

It is often necessary to write productions to recognize a list or sequence of items. Invisible Jacc uses LR parsing, which favors the use of *left recursive* rules. Here is a left-recursive fragment to recognize a list of items:

```
%productions:
ItemList -> Item;
ItemList -> ItemList Item;
```

When the parser encounters a list of items, it reduces the first production for the first item, then reduces the second production once for each succeeding item.

If you want to allow an empty list, you can write the productions like this:

```
%productions:
ItemList -> ;
ItemList -> ItemList Item;
```

In this case, the parser reduces the first production at the start of the list. Then, it reduces the second production once for each item in the list.

If you want to recognize a list of items separated by commas, you can write this:

```
%productions:
ItemList -> Item;
ItemList -> ItemList ',' Item;
```

If you want to recognize a list of items separated by commas, and also allow an empty list, it's a little more complicated:

```
%productions:
ItemList -> ;
ItemList -> NonemptyItemList;
NonemptyItemList -> Item;
NonemptyItemList -> NonemptyItemList ',' Item;
```

3.5.7 Example: Right Recursion

Invisible Jacc also supports the use of *right recursive* rules. Here is a right-recursive fragment to recognize a list of items:

```
%productions:
ItemList -> Item;
ItemList -> Item ItemList;
```

When the parser encounters a list of items, it reads all the items and pushes them onto the parser's stack. Then, the parser reduces the first production for the last item. Finally, the parser reduces the second production once for each remaining item, beginning with the next-to-last item and ending with the first item.

Right recursion is less desirable then left recursion because of the need to push all the items onto the parser's stack. This can consume a lot of memory if the list is lengthy. (The Invisible Jacc parser can enlarge the stack dynamically, so the parser's stack won't overflow unless your computer runs out of memory.)

3.5.8 Example: Optional Elements

Programming languages often have constructions that contain optional elements. For example, suppose you're defining a language where each statement can optionally be labeled. One way to do this is to write two productions, one for unlabeled statements and one for labeled statements:

```
%productions:
Statement -> Expression ';';
Statement -> identifier ':' Expression ';';
```

A second approach is to introduce a new nonterminal symbol to represent the optional element. The new nonterminal symbol has two productions: an empty production, and a production that contains the optional element.

```
%productions:
Statement -> OptionalLabel Expression ';';
OptionalLabel ->;
OptionalLabel -> identifier ':';
```

In some grammars, using empty productions for optional elements can cause parsing conflicts. The above two examples illustrate why this sometimes occurs. In the second example, the parser must reduce `OptionalLabel` at the start of the statement. This means that the parser must decide if a label is present when it has only seen the first terminal symbol in the statement. By comparison, in the first example the parser does not have to decide if a label is present until it has seen the entire statement.

If Invisible Jacc reports parsing conflicts, and you have used empty productions for optional elements, try eliminating the empty productions and rewriting the grammar using two productions for each optional element.

3.5.9 Example: Interior Actions

The parser calls a nonterminal factory whenever it reduces a production. That means the nonterminal factory isn't executed until after the parser has processed the production's entire right hand side. Sometimes, you might like to have code that executes in the middle of a production, instead of at the end. For example, suppose you have a production that defines a simple assignment statement:

```
%productions:
AssignmentStatement -> identifier '=' Expression ';';
```

The nonterminal factory for `'AssignmentStatement'` is called after the entire statement has been processed. But you might like to have code that executes as soon as the `'identifier'` is seen. For instance, you might want to check that the `'identifier'` is the name of a variable, and make a note of its type. This would be useful if you need to process expressions in different ways, depending on the type of variable that they're being assigned to.

Code that is executed in the middle of a production is called an *interior action*. To define an interior action, create a new nonterminal symbol that refers to part of the production. Here is how to do it:

```
%productions:
AssignmentStatement -> AssignmentLHS '=' Expression ';';
AssignmentLHS -> identifier;
```

Now, the parser reduces the production for `'AssignmentLHS'` as soon as it sees the `'identifier'`. Therefore, the nonterminal factory for `'AssignmentLHS'` is called as soon as the `'identifier'` is seen. The nonterminal factory for `'AssignmentLHS'` can then perform whatever actions are needed.

3.5.10 Example: The Dangling Else

Invisible Jacc lets you use ambiguous grammars. An *ambiguous grammar* is a grammar that has parsing conflicts because a given input can be grouped into productions in more than one way. (It is also possible for parsing conflicts to arise simply because the grammar is not LR(1) or LALR(1), even if the grammar is unambiguous.)

Before launching into an example, a few remarks are in order. An ambiguous grammar can always be rewritten into an unambiguous grammar that recognizes the same language. An excellent example can be found the Java Language Specification, which includes an unambiguous LALR(1) grammar for the complete Java language. In there, you will find solutions for the dangling else problem, for operator precedence and associativity, and other problems.

So, in some sense it is not necessary to use ambiguous grammars. But there are two reasons why you might want to use ambiguous grammars. First, an ambiguous grammar can be much simpler to write than an unambiguous grammar. This is especially true for the examples we will consider here, namely the dangling else and operator precedence. Second, an ambiguous grammar can produce a more efficient parser, because it often requires fewer reductions than an unambiguous grammar.

Let us now describe the *dangling else* problem. Suppose we want to define a programming language which supports *if-then* and *if-then-else* constructs. We might try to write productions like this:

```
// This fragment doesn't work
%productions:
Statement -> if '(' Expression ')' Statement;
Statement -> if '(' Expression ')' Statement else Statement;
```

Call the first production the *if-then* rule, and call the second production the *if-then-else* rule. These two rules are an ambiguous grammar. To see this, consider the following input:

```
if ( E1 ) if ( E2 ) S1 else S2
```

There are two ways to group the input. One possibility is to attach the 'else' to the second 'if', as shown below. This corresponds to applying first the *if-then-else* rule, and then the *if-then* rule. This is the rule adopted by most programming languages, including Java and C++.

```
if ( E1 )
{
   if ( E2 )
   {
      S1
   }
   else
   {
      S2
   }
}
```

The other possibility is to attach the 'else' to the first 'if', as shown below. This corresponds to applying first the *if-then* rule, and then the *if-then-else* rule.

```
if ( E1 )
{
   if ( E2 )
   {
      S1
   }
}
else
{
   S2
}
```

We say that this grammar is *ambiguous* because there are two different sequences of productions that can be applied to the same input. Let us see what this means to the parser. The parser reads program input from left to right. Consider what happens when the parser has seen the following input:

```
if ( E1 ) if ( E2 ) S1
```

The parser knows that the next symbol on the input is the terminal symbol 'else'. There are now two possibilities. The first possibility is to shift the 'else' onto the parse stack, then shift S2, yielding:

```
if ( E1 ) if ( E2 ) S1 else S2
```

Then, the parser applies the *if-then-else* rule to reduce the input to:

```
if ( E1 ) Statement
```

Finally, the parser applies the *if-then* rule. This corresponds to the first interpretation described above.

The other possibility is to immediately apply the *if-then* rule, reducing the input to:

```
if ( E1 ) Statement
```

Then, the parser shifts ‘else’ and S2 onto the parse stack, yielding:

```
if ( E1 ) Statement else S2
```

Finally, the parser applies the *if-then-else* rule. This corresponds to the second interpretation described above.

This illustrates a *shift-reduce conflict*. When the parser sees the terminal symbol ‘else’, it has a choice of either shifting or reducing. Each choice yields a different interpretation of the input.

Invisible Jacc lets you write a *precedence clause* to resolve the shift-reduce conflict. Suppose that you want to attach the “dangling else” to the nearest enclosing ‘if’, as in Java. Then you would write the productions like this:

```
// Resolves the "dangling else" the same way as Java or C++
%productions:
Statement -> if '(' Expression ')' Statement %shift else;
Statement -> if '(' Expression ')' Statement else Statement;
```

Notice that we have added a `%shift` clause to the *if-then* rule. Let’s see what it says. It says that if the next symbol on the input is `'else'`, and the parser has a choice of either shifting the `'else'` or reducing the *if-then* rule, then the parser should shift the `'else'`. As we showed above, this yields the standard “dangling else” rule used by Java and C++.

Alternatively, suppose we want to attach the “dangling else” to the outermost unmatched ‘if’. Then you would write the productions like this:

```
// Resolves the "dangling else" differently than Java or C++
%productions:
Statement -> if '(' Expression ')' Statement %reduce else;
Statement -> if '(' Expression ')' Statement else Statement;
```

Notice that we have added a `%reduce` clause to the *if-then* rule. Let’s see what it says. It says that if the next symbol on the input is `'else'`, and the parser has a choice of either shifting the `'else'` or reducing the *if-then* rule, then the parser should reduce the *if-then* rule. As we showed above, this yields the opposite “dangling else” rule from the one used by Java and C++.

3.5.11 Example: Associativity

Perhaps the most common application of ambiguous grammars is to parse expressions. Expressions are most simply described in terms of the *precedence* and *associativity* of the operators. For example, in Java, multiplication and division are higher precedence than addition and subtraction, and all four operations are left-associative (that is, group from left-to-right).

Let’s begin with a very simple example. If you want to write a production to recognize subtraction, you might try this:

```
// This fragment doesn’t work
%productions:
Expression -> Expression '-' Expression;
```

Let’s see why this is ambiguous. Consider the following input:

```
E1 – E2 – E3
```

There are two possible interpretations. One interpretation is the following, which is known as *left associativity*:

```
( E1 – E2 ) – E3
```

The other interpretation is the following, which is known as *right associativity*:

```
E1 – ( E2 – E3 )
```

Let’s look at what this means to the parser. Suppose the parser has seen the following input:

```
E1 – E2
```

The next symbol on the input is the terminal symbol `'-'`. The parser now has two choices. The first choice is to immediately reduce the input to:

```
Expression
```

Then, the parser can shift the terminal symbol `'-'`, and then shift `E3`, to get:

```
Expression – E3
```

Finally, the parser can reduce this to an `'Expression'`. This is a left-associative parse.

The second choice is to shift the terminal symbol `'-'`, and then shift `E3`, to get:

```
E1 – E2 – E3
```

Then, the parser can reduce the right-hand part of the input to get:

```
E1 - Expression
```

Finally, the parser can reduce this to an `'Expression'`. This is a right-associative parse.

So, this is another example of a *shift-reduce conflict*. When the parser sees the second `'-'` symbol, it has two choices. It can perform a reduce operation, which leads to a left-associative parse. Or it can perform a shift operation, which leads to a right-associative parse.

Invisible Jacc lets you write a *precedence clause* to resolve this conflict. If you want to make subtraction left-associative, you would write the production this way:

```
// Define subtraction to be left-associative
%productions:
Expression -> Expression '-' Expression %reduce '-';
```

Notice that we’ve added a `%reduce` clause to the production. Let’s see what it says. It says that if the next input symbol is `'-'`, and the parser has the choice of either shifting the `'-'` or reducing the production, the parser should reduce the production. As we have explained above, this action makes subtraction left-associative.

If you want to make subtraction right-associative, you would write the production this way:

```
// Define subtraction to be right-associative
%productions:
Expression -> Expression '-' Expression %shift '-';
```

Notice that we’ve added a `%shift` clause to the production. Let’s see what it says. It says that if the next input symbol is `'-'`, and the parser has the choice of either shifting the `'-'` or reducing the production, the parser should shift the `'-'`. As we have explained above, this action makes subtraction right-associative.

3.5.12 Example: Operator Precedence and Associativity

We now present a more complicated example that shows how to use precedence clauses to define the precedence and associativity of a set of operators. In this example, we want to define the following eight operators:

`+`	Addition, an infix binary operator.
`-`	Subtraction, an infix binary operator.
`*`	Multiplication, an infix binary operator.

`/`	Division, an infix binary operator.
`^`	Exponentiation, an infix binary operator.
`-`	Negation, a prefix unary operator.
`abs`	Absolute value, a prefix unary operator.
`!`	Factorial, a postfix unary operator.

A *binary operator* is an operator with two operands, while a *unary operator* is an operator with one operand. The term *infix operator* means an operator that is written between the operands, for example, `"2+3"`. The term *prefix operator* means an operator that is written before the operand, for example, `"-7"`. The term *postfix operator* means an operator that is written after the operand, for example, `"5!"`.

We group our operators into five *precedence levels* as shown below. The minus sign appearing at the lowest precedence level is subtraction, while the minus sign at the third level is negation. For each precedence level, we also specify if the operators are left-associative or right-associative. (Notice that postfix operators are always left-associative, and prefix operators are always right-associative. Infix operators can be either left- or right-associative.)

`^`	right-associative, highest precedence
`!`	left-associative
`abs  -`	right-associative
`*  /`	left-associative
`+  -`	left-associative, lowest precedence

We also want to allow operators to be grouped with parentheses. The following fragment shows how to define these operators.

```
%productions:
Expr -> identifier;
Expr -> '(' Expr ')';
Expr -> Expr '^' Expr %shift '^' %reduce '!' '*' '/' '+' '-';
Expr -> Expr '!';
Expr -> abs Expr      %shift '^' '!' %reduce '*' '/' '+' '-';
Expr -> '-' Expr      %shift '^' '!' %reduce '*' '/' '+' '-';
Expr -> Expr '*' Expr %shift '^' '!' %reduce '*' '/' '+' '-';
Expr -> Expr '/' Expr %shift '^' '!' %reduce '*' '/' '+' '-';
Expr -> Expr '+' Expr %shift '^' '!' '*' '/' %reduce '+' '-';
Expr -> Expr '-' Expr %shift '^' '!' '*' '/' %reduce '+' '-';
```

There is a pattern to the `%shift` and `%reduce` clauses:

- Each production that defines a prefix or infix operator has `%shift` and `%reduce` clauses.
- The `%shift` clause contains:
  - All infix and postfix operators with higher precedence than the operator being defined, and
  - All right-associative infix operators with the same precedence as the operator being defined.
- The `%reduce` clause contains:
  - All infix and postfix operators with lower precedence than the operator being defined, and
  - All left-associative infix operators with the same precedence as the operator being defined.

Following this pattern lets you easily construct productions to define a set of operators with specified precedence and associativity.

3.5.13 Example: Using a Nonterminal in a Precedence Clause

In all of the examples so far, precedence clauses have contained terminal symbols. Invisible Jacc also allows precedence clauses to contain nonterminal symbols. Listing a nonterminal symbol in a precedence clause is equivalent to listing every terminal symbol that could be the first symbol of an expansion of the nonterminal symbol.

This is useful in defining implicit operators. An *implicit operator* is a binary operator that is written by writing the two operands one after the other, with no operator symbol. For example, you would write `"x y"` to apply the operator to operands `x` and `y`.

In this example, we show how to write productions to recognize regular expressions, where catenation is an implicit operator. We define the following three operators:

`*`	Kleene closure, a postfix unary operator.
`(implicit)`	Catenation, an implicit binary operator.
`|`	Alternation, an infix binary operator.

We group our operators into three *precedence levels* as shown below. For each precedence level, we also specify if the operators are left-associative or right-associative. (Notice that postfix operators are always left-associative, and prefix operators are always right-associative. Infix operators can be either left- or right-associative.)

`*`	left-associative, highest precedence
`(implicit)`	left-associative
`|`	left-associative, lowest precedence

We also want to allow operators to be grouped with parentheses. The following fragment shows how to define these operators.

```
%productions:
Expr -> identifier;
Expr -> '(' Expr ')';
Expr -> Expr '*';
Expr -> Expr Expr     %shift '*' %reduce Expr '|';
Expr -> Expr '|' Expr %shift '*' Expr %reduce '|';
```

The pattern of the `%shift` and `%reduce` clauses is the same as in the previous example, with one exception: the nonterminal symbol (`'Expr'` in this example) is used as if it were an infix operator denoting the implicit operator. Here are updated rules:

- Each production that defines a prefix, infix, or implicit operator has `%shift` and `%reduce` clauses.
- The `%shift` clause contains:
    - All infix and postfix operators with higher precedence than the operator being defined, and
    - All right-associative infix operators with the same precedence as the operator being defined, and
    - The nonterminal symbol, if (a) the implicit operator has higher precedence than the operator being defined, or (b) the implicit operator is right-associative and has the same precedence as the operator being defined.
- The `%reduce` clause contains:
    - All infix and postfix operators with lower precedence than the operator being defined, and
    - All left-associative infix operators with the same precedence as the operator being defined, and
    - The nonterminal symbol, if (a) the implicit operator has lower precedence than the operator being defined, or (b) the implicit operator is left-associative and has the same precedence as the operator being defined.

Recall that listing a nonterminal symbol in a precedence clause is the same as listing every terminal symbol that could be the first symbol of an expansion of the nonterminal symbol. In this example, the terminal symbols `'('` and `'identifier'` could be the first symbol of an expansion of the nonterminal symbol `'Expr'`. So, the productions could be written as shown below.

```
%productions:
Expr -> identifier;
Expr -> '(' Expr ')';
Expr -> Expr '*';
Expr -> Expr Expr     %shift '*' %reduce '(' identifier '|';
Expr -> Expr '|' Expr %shift '*' '(' identifier %reduce '|';
```

It is more convenient to use the nonterminal symbol, because Invisible Jacc automatically figures out which terminal symbols could be the first symbol of an expansion of the nonterminal symbol.

3.6 The %categories Section

The %categories section lists the character categories used by the scanner. A *character category* is a set of characters. The character categories are used as the building blocks of regular expressions. The following productions describe the %categories section:

```
Section -> CategoryHeader CategoryDefList

CategoryHeader -> '%categories' ':'

CategoryDefList ->

CategoryDefList -> CategoryDefList CategoryDefinition

CategoryDefinition -> Category '=' CatExp ';'

Category -> identifier

CatExp -> number

CatExp -> identifier

CatExp -> number '..' number

CatExp -> identifier '..' identifier

CatExp -> '%any'

CatExp -> '%none'

CatExp -> '%unicode'

CatExp -> '%uppercase'

CatExp -> '%lowercase'

CatExp -> '%titlecase'

CatExp -> '%letter'

CatExp -> '%digit'

CatExp -> '(' CatExp ')'

CatExp -> CatExp '-' CatExp

CatExp -> CatExp '&' CatExp

CatExp -> CatExp '|' CatExp
```

The %categories section consists of a header, followed by a series of category definitions. Each category definition is a statement that is terminated by a semicolon.

3.6.1 The %categories Header

The %categories section header consists of the keyword `%categories` followed by a colon.

3.6.2 Category Definition

A category definition contains the following three elements:

1. An identifier. This is called the *category name*.
2. An equal sign.
3. A *category expression* which describes a set of characters

The three elements must appear in exactly the order listed above.

The following conditions apply to category definitions:

- The order in which category definitions appear is immaterial.
- Each category definition must have a unique name. In other words, no two category definitions can have the same name.
- It is permitted for a category name to be the same as the name of a token or symbol. There is no special significance to such names.

3.6.3 Category Expressions

A *category expression* is a formula that describes a set of characters. The following table shows the forms that may be used in writing a category expression.

`number`	The character whose character code is the given number.
`identifier`	All characters which appear in the given identifier.
`number .. number`	All characters whose character code is greater than or equal to the first number, and less than or equal to the second number.
`identifier .. identifier`	All characters greater than or equal to the first identifier, and less than or equal to the second identifier. Each identifier must be exactly one character long.
`%any`	Any character.
`%none`	No characters.
`%unicode`	Any defined Unicode character.
`%uppercase`	Any uppercase Unicode letter.
`%lowercase`	Any lowercase Unicode letter.
`%titlecase`	Any titlecase Unicode letter.
`%letter`	Any Unicode letter.
`%digit`	Any Unicode digit.
`CatExp – CatExp`	*Difference*. Any character which is included in the first expression, but not included in the second expression.
`CatExp & CatExp`	*Intersection*. Any character which is included both in the first expression, and in the second expression.
`CatExp | CatExp`	*Union*. Any character which is included either in the first expression, or in the second expression, or both.

There are three operators that may be used to combine category expressions. The three operators are organized into two precedence levels as shown below. Within each precedence level, operators are left-associative, that is, they group from left to right.

`-  &`	highest precedence
`|`	lowest precedence

Parentheses may be used to group operators in a category expression.

Technical note: The six keywords `%unicode`, `%uppercase`, `%lowercase`, `%titlecase`, `%letter`, and `%digit` categorize characters using the standard Java functions `Character.isDefined`, `Character.isUpperCase`, `Character.isLowerCase`, `Character.isTitleCase`, `Character.isLetter`, and `Character.isDigit`, respectively. These functions behave differently on different Java implementations. The Invisible Jacc scanner tables reflect the behavior of these functions *on the computer where the scanner tables are generated*. If the scanner tables are generated on one computer, and then the scanner is executed on a second computer, the scanner categorizes characters according to the Java functions defined on the first computer, even if the Java functions on the second computer are different. In other words, once you generate the scanner tables, the run-time behavior of the scanner is identical on all Java implementations. A corollary is that the character categories recognized by the scanner may differ from the character categorizations obtained at run-time from the local `Character` class.

3.6.4 Examples of Category Definitions

The following shows six different ways to define the character category `'decDigit'` to be the decimal digits from 0 through 9.

```
decDigit = '0'..'9';

decDigit = 0x30..0x39;

decDigit = 48..57;

decDigit = '0123456789';

decDigit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

decDigit = 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57;
```

The following shows three different ways to define the character category `'notEol'` to be any character other than carriage return or line feed.

```
notEol = %any – 10 - 13;

notEol = %any – 0x0A – 0x0D;

notEol = %any – (0x0a | 0x0d);
```

The following defines the character category `'letter'` to be any Unicode letter.

```
letter = %letter;
```

The following defines the character category `'letterOrDigit'` to be any Unicode character which is either a letter or a digit.

```
letterOrDigit = %letter | %digit;
```

The following defines the character category `'unicodeNotLetter'` to be any Unicode character which is not a letter.

```
unicodeNotLetter = %unicode - %letter;
```

The following defines the character category `'unicodeNotASCII'` to be any Unicode character whose character code is greater than or equal to 0x100 (in other words, any Unicode character that is not an ASCII character).

```
unicodeNotASCII = %unicode & 0x0100..0xFFFF;
```

3.7 The %conditions Section

The %conditions section lists the scanner's start conditions. *Start conditions* are used to specify which regular expressions are active at any given point during the scan. The current start condition can be changed at any time, allowing you to select a set of candidate regular expressions based on what has been previously seen during the scan. The following productions describe the %conditions section:

```
Section -> ConditionHeader ConditionDefList

ConditionHeader -> '%conditions' ':'

ConditionDefList ->

ConditionDefList -> ConditionDefList ConditionDefinition

ConditionDefinition -> Condition ';'

Condition -> identifier
```

The %conditions section consists of a header, followed by a series of condition definitions. Each condition definition is a statement that is terminated by a semicolon.

3.7.1 The %conditions Header

The %conditions section header consists of the keyword `%conditions` followed by a colon.

3.7.2 Condition Definition

A condition definition consists of an identifier, which is called the *condition name*.

The following rules apply to condition definitions:

- Conditions are assigned consecutive numbers starting with 0, in the order that they appear in the %conditions section.
- Condition 0 is the *initial condition*, that is, the condition in which the scanner starts. Therefore, the first condition listed in the %conditions section is the initial condition.
- Each condition must have a unique name. That is, no two conditions can have the same name.
- It is permitted for a condition name to be the same as the name of a symbol, character category, or token. There is no special significance to such names.
- If no conditions are specified, the generator automatically creates a single condition called `%%normal`.

For example, the following defines three start conditions: ‘initialCondition’ is condition 0, ‘conditionOne’ is condition 1, and ‘conditionTwo’ is condition 2.

```
%conditions:
initialCondition;
conditionOne;
conditionTwo;
```

3.7.3 Example: C-Style Comments

In this example we show how to use start conditions to scan C-style comments. Recall that a C-style comment is an arbitrarily long string of text introduced by `/*` and terminated by `*/`.

As a first attempt, you might try the following:

```
// This is not recommended

%categories:
'/' = '/';
'*' = '*';
any = %any;

%tokens:
comment = '/' '*' (any* ~ '*' '/') '*' '/';
```

The regular expression shown above is straightforward: it recognizes the characters `/*`, followed by any string of characters that does not contain the substring `*/`, followed by the characters `*/`. The regular expression does, in fact, match C-style comments. But it has at least three problems.

First, the regular expression can match an unlimited amount of text. It is not uncommon to encounter C-style comments that are several kilobytes long. The scanner must buffer up the entire matching text, which leads to the scanner consuming a large amount of buffer space.

Second, the regular expression spans line ends. If you are using a token factory to count lines, this regular expression will cause line-ends within comments to be missed. You would have to write a token factory for comments that scans the comment text and updates the line counter.

Third, there is no provision for detecting run-on comments. A *run-on comment* occurs when the user forgets to write the `*/` characters at the end of the comment.

The following shows how to use start conditions to recognize C-style comments. The idea is to switch the start condition when you encounter the beginning of a comment, and then restore the initial condition at the end of the comment. To illustrate the idea, we will keep this example very simple and not very efficient. In the next subsection we will show a more realistic example.

```
// Simple illustration, see next subsection for better version

%categories:
'/' = '/';
'*' = '*';
cr = 13;
lf = 10;
notEol = %any – 10 – 13;

%conditions:
notInComment;
inComment;

%tokens notInComment:
beginComment = '/' '*';

%tokens inComment:
whiteSpace = notEol* ~ '*' '/';
endComment = (notEol* ~ '*' '/') '*' '/';

%tokens:
lineEnd = cr | lf | cr lf;
```

This example defines two start conditions: `'notInComment'` is condition 0, which is the initial condition; and `'inComment'` is condition 1.

There are three %tokens sections. The first %tokens section contains tokens that are active only when the start condition is `'notInComment'`. In this section, we include the token that recognizes the beginning of a comment.

The second %tokens section contains tokens that are active only when the start condition is `'inComment'`. In this section, we include tokens to recognize (a) a single line in the interior of a comment and (b) the end of a comment.

The third %tokens section contains tokens that are always active. (If no conditions are listed in the %tokens section header, that means the tokens are always active.) In this section, we include the token that recognizes the end of a line.

The following code shows the token factories for this example.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("beginComment", "",
         new BeginComment ());
      _scannerTable.linkFactory ("endComment", "",
         new EndComment ());
      _scannerTable.linkFactory ("lineEnd", "",
         new LineEnd ());
      ...
   }
   ...
   public void scannerEOF (Scanner scanner, Token token)
   {
      if (scanner.condition() == 1)
      {
         reportError (token, null, "Run-on comment.");
      }
      return;
   }
   ...

final class BeginComment extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'inComment'

      scanner.setCondition (1);

      // Discard token

      return discard;
   }
}

final class EndComment extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'notInComment'

      scanner.setCondition (0);

      // Discard token

      return discard;
   }
}

final class LineEnd extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Bump the line number

      scanner.countLine ();

      // Discard token

      return discard;
   }
}
...
}  // End class Compiler
```

The token factory for `'beginComment'` sets the start condition to 1 when the start of a comment is recognized. It does this by calling the function `scanner.setCondition` with an argument of 1. This activates the tokens listed in the second %tokens section, and deactivates the tokens listed in the first %tokens section.

The token factory for `'endComment'` sets the start condition back to 0 when the end of a comment is recognized. It does this by calling the function `scanner.setCondition` with an argument of 0. This activates the tokens listed in the first %tokens section, and deactivates the tokens listed in the second %tokens section.

The token factory for `'lineEnd'` counts the line. Since this token is listed in the third %tokens section, it is always active. Therefore, it counts lines both inside and outside comments.

There is no token factory for `'whiteSpace'`, so it is handled by the default token factory, which in this case simply discards the token.

Run-on comments are detected in the function `scannerEOF`. The scanner calls `scannerEOF` when it reaches the end of the source file. The code calls `scanner.condition` to get the current start condition. If the current start condition is `'inComment'`, it issues an error message.

In this example, the numerical values of the start conditions are hard-coded as constants 0 and 1. The next subsection shows how to avoid hard-coding the condition numbers.

3.7.4 Example: Java Comments and White Space

In this example we show how to use start conditions to scan Java comments and white space. This includes: (a) the characters space, tab, and form feed; (b) the ASCII sub character (character code 0x1A) if it is the last character in the file; (c) line comments, which are introduced by `//` and extend to the end of the line; and (d) C-style comments, which are introduced by `/*` and terminated by `*/`.

We also show dynamic binding of start conditions, which eliminates the need to hard-code condition numbers.

The following grammar specification fragment is excerpted from file `Ex1Grammar.jacc`.

```
%categories:
'*' = '*';
'/' = '/';
space = 9 | 12 | 32;       // tab, form feed, and space
sub = 26;                  // ASCII sub
cr = 13;                   // carriage return
lf = 10;                   // line feed
notEol = %any - 10 - 13;   // any character that isn't a line end
any = %any;                // any character

%conditions:
notInComment;              // Normal condition
inComment;                 // Inside a multi-line comment

%tokens notInComment:
whiteSpace = (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
             ('/' '/' notEol*)?;
beginComment = (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
               '/' '*' (notEol* ~ '*' '/');
illegalChar = sub / any;
whiteSpace = sub;

%tokens inComment:
endComment = (notEol* ~ '*' '/') '*' '/'
             (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
             ('/' '/' notEol*)?;
whiteSpace = ((notEol* ~ '*' '/') '*' '/'
              (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
              '/' '*'
             )?
             (notEol* ~ '*' '/');

%tokens:
lineEnd = cr | lf | cr lf;
```

The first %tokens section defines four tokens as follows. These tokens are active only when the start condition is `'notInComment'`.

- `'whiteSpace'` is any number of spaces, tabs, form feeds, and single-line C-style comments, optionally followed by a line comment.
- `'beginComment'` is any number of spaces, tabs, form feeds, and single-line C-style comments, followed by the first line of a multi-line C-style comment.
- `'illegalChar'` is a sub character which is not the last character of the file.
- `'whiteSpace'` is a sub character which is the last character of the file.

The second %tokens section defines two tokens as follows. These tokens are active only when the start condition is `'inComment'`.

- `'endComment'` is the last line of a multi-line C-style comment, followed by any number of spaces, tabs, form feeds, and single-line C-style comments, optionally followed by a line comment.
- `'whiteSpace'` is either (a) a complete line inside a multi-line C-style comment, or (b) the last line of a multi-line C-style comment, followed by any number of spaces, tabs, form feeds, and single-line C-style comments, followed by the first line of a new multi-line C-style comment.

The third %tokens section defines the token `'lineEnd'`, which is the end of a line. This token is always active.

The following code shows how to write the corresponding token factories, and also how to dynamically link the start condition names. This is very similar to code that can be found in file `Ex1Compiler.java`.

```
public class Compiler extends CompilerModel
{
   int _conditionNotInComment;
   int _conditionInComment;
   ...
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("beginComment", "",
         new BeginComment ());
      _scannerTable.linkFactory ("endComment", "",
         new EndComment ());
      _scannerTable.linkFactory ("lineEnd", "",
         new LineEnd ());
      _scannerTable.linkFactory ("illegalChar", "",
         new IllegalChar ());
      ...
      _conditionNotInComment =
         _scannerTable.lookupCondition ("notInComment");
      _conditionInComment =
         _scannerTable.lookupCondition ("inComment");
      ...
   }
   ...
   public void scannerEOF (Scanner scanner, Token token)
   {
      if (scanner.condition() == _conditionInComment)
      {
         reportError (token, null, "Run-on comment.");
      }
      return;
   }
   ...

final class BeginComment extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'inComment'

      scanner.setCondition (_conditionInComment);

      // Discard token
      return discard;
   }
}

final class EndComment extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'notInComment'

      scanner.setCondition (_conditionNotInComment);

      // Discard token

      return discard;
   }
}

final class LineEnd extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Bump the line number

      scanner.countLine ();

      // Discard token

      return discard;
   }
}

final class IllegalChar extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Report this as an illegal token

      scanner.client().scannerUnmatchedToken (scanner, token);

      // Discard token

      return discard;
   }
}
...
}  // End class Compiler
```

In the above code, the variables `_conditionNotInComment` and `_conditionInComment` hold the condition numbers for start conditions `'notInComment'` and `'inComment'` respectively. The function `lookupCondition` is used to obtain the condition number for each start condition. The code uses these two variables, instead of hard-coded numbers, to refer to start conditions.

In the token factory for `'illegalChar'`, the function `scanner.client().scannerUnmatchedToken` is used to report that the ASCII sub is an illegal character.

The remainder of the example code is the same as in the preceding subsection.

3.7.5 Example: Rescanning Input

It is possible for a token factory to change the start condition and then ask the scanner to rescan the input. This is useful if the token factory determines that none of the token definitions in the current start condition are applicable, and the token factory would like the scanner to try again with a different start condition.

As an example, we consider the problem of scanning a Unicode plain text file. A *Unicode plain text file* is a file that contains Unicode characters, with each character represented by two bytes in the file.

According to the Unicode standard, a Unicode plain text file may optionally have a *byte order mark*, or *BOM*, in the first two bytes of the file. The numerical value of the BOM is 0xFEFF. If the BOM is present, it is to be discarded when reading the file. In other words, the BOM is not part of the Unicode text.

The purpose of the BOM is to make sure that the program reading the file uses the same byte ordering as the program that wrote the file. A byte-reversed BOM would have numerical value 0xFFFE, which is never a valid Unicode character. (The Unicode standard itself does not specify a particular byte ordering.)

So in scanning a Unicode plain text file, we would like to do the following:

- If the first character in the file is BOM (value 0xFEFF), then skip over the BOM and beginning scanning with the second character in the file.
- If the first character in the file is a byte-reversed BOM (value 0xFFFE), then abort scanning and report an error.
- Otherwise, begin scanning with the first byte of the file.

The following grammar specification fragment shows how we can achieve this.

```
%options:
%charsetsize 0x10000;      // Character set size for Unicode = 65536

%categories:
BOM = 0xFEFF;
reversedBOM = 0xFFFE;
any = %any;                // any character

%conditions:
checkForBOM;               // At start of file, need to check for BOM
normal;                    // Normal scanning

%tokens checkForBOM:
BOM = BOM;
reversedBOM = reversedBOM;
notBOM = any;

%tokens normal:
// ... Normal token definitions go here ...
```

Notice that `'checkForBOM'` is the initial condition. This means that we check for the BOM character at the start of the file. As soon as we determine whether or not the BOM is present, we change the start condition to 'normal' so that normal scanning can proceed.

The first %tokens section defines three tokens as follows. These tokens are active only when the start condition is `'checkForBOM'`.

- `'BOM'` matches the BOM character.
- `'reversedBOM'` matches the byte-reversed BOM character.
- `'notBOM'` matches any single character other than BOM and byte-reversed BOM.

The following code shows how to write the corresponding token factories, and also how to dynamically link the start condition names.

```
public class Compiler extends CompilerModel
{
   int _conditionCheckForBOM;
   int _conditionNormal;
   ...
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("BOM", "",
         new BOM ());
      _scannerTable.linkFactory ("reversedBOM", "",
         new ReversedBOM ());
      _scannerTable.linkFactory ("notBOM", "",
         new NotBOM ());
      ...
      _conditionCheckForBOM =
         _scannerTable.lookupCondition ("checkForBOM");
      _conditionNormal =
         _scannerTable.lookupCondition ("normal");
      ...
   }
   ...

final class BOM extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'normal'

      scanner.setCondition (_conditionNormal);

      // Discard token

      return discard;
   }
}

final class ReversedBOM extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Report the error

      reportError (token, null, "Incorrect byte order.");

      // Abort the scanning

      throw new SyntaxException ("Incorrect byte order");
   }
}

final class NotBOM extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Set start condition to 'normal'

      scanner.setCondition (_conditionNormal);

      // Tell the scanner to rescan the input

      scanner.setTokenLength (0);

      // Discard token

      return discard;
   }
}
...
}  // End class Compiler
```

In the above code, the variables `_conditionCheckForBOM` and `_conditionNormal` hold the condition numbers for start conditions `'checkForBOM'` and `'normal'` respectively. The function `lookupCondition` is used to obtain the condition number for each start condition. The code uses these two variables, instead of hard-coded numbers, to refer to start conditions.

The token factory for `'BOM'` simply changes the start condition to `'normal'` and then discards the token. This causes normal scanning to begin after the BOM character.

The token factory for `'reversedBOM'` prints an error message, and then throws `SyntaxException` to abort scanning.

The interesting part is the token factory for `'notBOM'`. This token factory changes the start condition to `'normal'` and discards the token. But it also uses the function `scanner.setTokenLength` to set the token length to zero. As a result, when the scanner advances to the next token, it move its internal pointer forward by zero characters. Therefore, normal scanning resumes at the start of the file. In other words, the first character of the file is rescanned, but this time in the `'normal'` start condition.

Note: In this example, we throw an exception if a byte-reversed BOM is found. Instead, we might want to byte-reverse all the characters in the file. If you want to byte-reverse all the characters in the file, you have to put the byte-reversal logic into a prescanner. In other words, you need to write a class that implements the interface `PrescannerChar`, and put code in that class to byte-reverse all the characters before the scanner sees them. If you were writing a real-world program to process Unicode plain text files, it would probably make more sense to put all the BOM handling into the prescanner, instead of using token factories like in this example.

Note: Setting the token length to zero and discarding the token is not the same thing as rejecting the token. When you set the token length to zero and discard the token, the scanner rescans the input from the beginning, using the new start condition. When you reject the token, the scanner does not rescan the input; it just passes the token to the next eligible token factory according to the stored results of the original scan, even if the start condition has been changed.

3.8 The %tokens Section

The %tokens section lists the tokens recognized by the scanner. The following productions describe the %tokens section:

```
Section -> TokenHeader TokenDefList

TokenHeader -> '%tokens' ConditionSet ':'

ConditionSet ->

ConditionSet -> ConditionSet Condition

Condition -> identifier

TokenDefList ->

TokenDefList -> TokenDefList TokenDefinition

TokenDefinition -> Token LinkName Parameter '=' RegExp ';'

TokenDefinition -> Token LinkName Parameter '=' RegExp '/' RegExp ';'

Token -> identifier

LinkName ->

LinkName -> '{' identifier '}'

Parameter ->

Parameter -> '#' number

RegExp -> Category

RegExp -> '(' RegExp ')'

RegExp -> RegExp '*'

RegExp -> RegExp '+'

RegExp -> RegExp '?'

RegExp -> RegExp RegExp

RegExp -> RegExp '-' RegExp

RegExp -> RegExp '&' RegExp

RegExp -> RegExp '~' RegExp

RegExp -> RegExp '@' RegExp

RegExp -> RegExp '|' RegExp
```

```
Category -> identifier
```

The %tokens section consists of a header, followed by a series of token definitions. Each token definition is a statement that is terminated by a semicolon.

3.8.1 The %tokens Header

The %tokens section header consists of:

- The keyword `%tokens`.
- An optional list of start conditions.
- A colon.

If the header includes a list of start conditions, then tokens defined in the section are active only in the specified start conditions. If the header does not include a list of start conditions, then the tokens defined in the section are always active.

The following example defines three start conditions and five %token sections which are active in different start conditions.

```
%conditions:
conditionZero;
conditionOne;
consitionTwo;

%tokens conditionZero:
// tokens that are active only in conditionZero ...

%tokens conditionOne:
// tokens that are active only in conditionOne ...

%tokens conditionTwo:
// tokens that are active only in conditionTwo ...

%tokens conditionZero conditionTwo:
// tokens that are active only in conditionZero or conditionTwo ...

%tokens:
// tokens that are always active ...
```

3.8.2 Token Definition

A token definition contains the following six elements:

1. An identifier. This is called the *token name*.
2. An optional *link name*. If included, it is an identifier enclosed in curly braces. The link name is used to determine which token factory is linked to this token. It has no other significance. (In particular, it is immaterial whether or not the link name is the same as the name of a token or symbol.) If the link name is omitted, it defaults to an empty string.
3. An optional *parameter*. If included, it is a number preceded by the # character. If omitted, the parameter defaults to: (a) zero, if the token name is not the same as the name of a terminal symbol, or (b) the numerical value of the symbol, if the token name is the same as the name of a terminal symbol. The parameter is passed to the token factory whenever this token is recognized. The parameter has no other significance.

4. An equal sign.
5. A regular expression. This is called the *token regular expression*.
6. Optionally, a slash character followed by a second regular expression. The second regular expression is called the *right context regular expression*.

The six elements must appear in exactly the order listed above.

The following conditions apply to token definitions:

- The order in which token definitions appear is significant. If an input string matches more than one token definition, then priority is given to the token definition that appears earliest in the file.
- Every token definition has a *link name*, which is an identifier. The link name, together with the token name, is used to select which token factory to use when that token definition is recognized. If two token definitions have the same token name and the same link name, then the same token factory is used for both token definitions.
- Every token definition has a *parameter*, which is an integer. The parameter is passed to the token factory whenever the token definition is recognized. The meaning of the parameter is client-defined. Typically, the parameter is the number of the token that the token factory creates.
- If a token name is the same as the name of a terminal symbol, then the parameter is automatically set equal to the numerical value of the terminal symbol. In this case, there must not be an explicit parameter in the token definition.
- If a token name is not the same as the name of a terminal symbol, then there may be an explicit parameter in the token definition.
- If a token name is not the same as the name of a terminal symbol, and there is no explicit parameter, then the token definition's parameter defaults to zero.
- It is an error for a token name to be the same as the name of a nonterminal symbol.
- An empty string never matches the token regular expression, even if the token regular expression is written in a way that makes an empty string appear to match it. When processing the token regular expression, Invisible Jacc implicitly adds a requirement that matching strings must be nonempty. However, an empty string can match the right context regular expression.

3.8.3 Regular Expressions

A *regular expression* is a formula that describes a set of strings. A string is said to *match* the regular expression if it belongs to the set of strings described by the regular expression.

The term *catenation* refers to joining strings end-to-end to form a new string. For example, the catenation of "abc" and "de" is the string "abcde".

The following table shows the forms that may be used in writing a regular expression.

`identifier`	*Single character*. Matched by any one-character string, whose single character belongs to the specified category. The identifier must be the name of a character category.
`RegExp *`	*Kleene closure*. Matched by the catenation of zero or more strings, each of which matches the specified regular expression.
`RegExp +`	*Positive closure*. Matched by the catenation of one or more strings, each of which matches the specified regular expression.
`RegExp ?`	*Optional closure*. Matched by the empty string, and by any string that matches the specified regular expression.
`RegExp RegExp`	*Catenation*. Matched by any string which is the catenation of a string that matches the first expression with a string that matches the second expression.
`RegExp – RegExp`	*Difference*. Matched by any string that matches the first expression, but does not match the second expression.
`RegExp & RegExp`	*Intersection*. Matched by any string that matches the first expression, and also matches the second expression.
`RegExp ~ RegExp`	*Exclusion*. Matched by any string that matches the first expression, but does not contain any substring that matches the second expression.
`RegExp @ RegExp`	*Inclusion*. Matched by any string that matches the first expression, and also contains at least one substring that matches the second expression.
`RegExp | RegExp`	*Alternation*. Matched by any string that matches either the first expression, or the second expression, or both.

There are nine operators that may be used to combine regular expressions. The nine operators are organized into four precedence levels as shown below. Within each precedence level, operators are left-associative, that is, they group from left to right.

`* + ?`	highest precedence
`catenation`	second precedence level
`- & ~ @`	third precedence level
`|`	lowest precedence

Parentheses may be used to group operators in a regular expression.

3.8.4 Examples of Regular Expressions

We now present a few regular expressions to illustrate how the regular expression operators work. For these examples, assume the following character categories:

```
%categories:
x = 'x';
y = 'y';
z = 'z';
```

With these assumptions, the following table shows some regular expressions and some of the strings that match them.

`x`	The string *x*.
`x y z`	The string *xyz*.
`x y*`	The strings *x*, *xy*, *xyy*, *xyyy*, *xyyyy*, and so on.
`x y+`	The strings *xy*, *xyy*, *xyyy*, *xyyyy*, and so on; but not the string *x*.
`x y?`	The strings *x* and *xy*.
`x y? z`	The strings *xz* and *xyz*.
`x | y`	The strings *x* and *y*; but not the string *xy*.
`x+ | y+`	The strings *x*, *xx*, *xxx*, and so on; and the strings *y*, *yy*, *yyy*, and so on; but not the string *xy*.
`(x | y)+`	Any string, at least one character long, made up of the letters *x* and *y*. For example, this includes the strings *x*, *y*, *xy*, *yx*, *xx*, *yy*, *xxy*, *xyx*, *yyxy*, *xyxy*, and so forth.
`(x y)+`	Any string made up of one or more repetitions of the string *xy*. For example, this includes the strings *xy*, *xyxy*, *xyxyxy*, and so on; but not the string *yx*.
`(x | y)+ - (x y)+`	Any nonempty string, made up of the letters *x* and *y*, which is not made up of repetitions of *xy*. For example, this includes the strings *x*, *y*, *yx*, *xx*, *yy*, *xxy*, *xyx*, *yyxy*, and so forth; but not the strings *xy*, *xyxy*, *xyxyxy*, and so on.
`(x | y)+ & (y | z)+`	The strings *y*, *yy*, *yyy*, and so on; but not the strings *xy* or *yz*.
`(x | y | z)+ ~ x y`	Any nonempty string made up of the letters *x*, *y*, and *z*, which does not contain the substring *xy*. For example, this includes the strings *x*, *y*, *z*, *yx*, *yz*, *xz*, *xzy*, *yxz*, *zxzy*, and so forth; but not the strings *xy*, *zxy*, *xyy*, *yxyz*, *zxyyxy*, and so on.
`(x | y | z)+ @ x y`	Any nonempty string made up of the letters *x*, *y*, and *z*, which contains at least one occurrence of the substring *xy*. For example, this includes the strings *xy*, *zxy*, *xyy*, *yxyz*, *zxyyxy*, and so forth; but not the strings *x*, *y*, *z*, *yx*, *yz*, *xz*, *xzy*, *yxz*, *zxzy*, and so on.
`x* y x* @ x`	Any string made up of the letters *x* and *y*, which contains at least one *x* and exactly one *y*. For example, this includes the strings *xy*, *yx*, *xxxy*, *xxyx*, *yxxx*, and so on; but not the strings *x*, *y*, *xxyyx*, and so on.

3.8.5 Scanner Operation

We now describe how the scanner uses regular expressions. The function of the scanner is to read an input file and break the input into *tokens*. Each token is a string of one or more consecutive input characters.

The scanner maintains a *current position* in the input. The general idea is to find the longest possible token beginning at the current position. In case of a tie, the scanner gives priority to the token definition that appears earliest in the grammar specification file.

Beginning at a given position in the input, the scanner finds the next token as follows:

1. Each token definition is marked *active* or *inactive*, depending on the scanner's start condition. Inactive token definitions do not participate in the scan.
2. The scanner finds the *context string*.
    - The context string is the longest string, beginning at the current position, that matches any of the active token definitions.
    - If a token definition does not have a right context, we say that a string *matches* the token definition if it matches the token regular expression.

- If a token definition has a right context, we say that a string *matches* the token definition if it matches the catenation of the token regular expression with the right context regular expression.
- If no such string exists, the scanner informs its client that an error occurred. Then, the scanner advances the current position by one character, and goes back to step 1 to begin a new scan.

3. The scanner selects the *current token definition*.
   - The current token definition is the token definition that is matched by the context string.
   - If there is more than one token definition that is matched by the context string, then the scanner chooses the token definition that appears earliest in the grammar specification file.
4. The scanner finds the *token string*.
   - The token string is a string, beginning at the current position, that matches the token regular expression for the current token definition.
   - If the current token definition does not have a right context, then the token string is the same as the context string.
   - If the current token definition has a right context, then the scanner splits the context string into two substrings, one that matches the token regular expression and one that matches the right context regular expression. The first substring becomes the token string. If there is more than one way to split the context string, then the scanner chooses the split that produces the longest token string.
5. The scanner calls the token factory for the current token definition. The scanner makes the token string and context string available to the token factory. The token factory can *assemble* the token, *discard* the token, or *reject* the token.
6. If the token factory assembles the token, the scanner's current position is advanced to the end of the token string. Then, the token is returned to the scanner's client, along with any value supplied by the token factory.
7. If the token factory discards the token, the scanner's current position is advanced to the end of the token string. Then, the scanner goes back to step 1 and begins a new scan.
8. If the token factory rejects the token, the scanner finds the next best combination of context string, token definition, and token string. (In choosing the new combination, the scanner does not re-scan the input, but rather uses the stored results of the previous scan.)
   - If possible, the scanner keeps the same context string and token definition, but uses a shorter token string. This can occur only if the current token definition has a right context, and the context string can be split in more than one way. If this is possible, the scanner chooses the next-longest token string, and goes to step 5.
   - Otherwise, if possible, the scanner keeps the same context string, but uses a different token definition. This can occur only if there is more than one token definition that is matched by the context string. If this is possible, the scanner chooses the next matching token definition in the grammar specification file, and goes to step 4.
   - Otherwise, if possible, the scanner uses a shorter context string. This can occur only if there is more than one string, beginning at the current position, that matches an active token definition. If this is possible, the scanner chooses the next-longest context string, and goes to step 3.
   - Otherwise, there is an error. The scanner reports the error to its client. Then, the scanner advances the current position by one character, and goes to step 1 to begin a new scan.

Technical note: If you are familiar with the Unix program Lex, you should be aware of a subtle difference between Lex and Invisible Jacc in the handling of right contexts. When Lex splits a context string into two substrings, Lex makes the first substring equal to the longest initial substring that matches the token regular expression. With Lex, there is no guarantee that the second substring actually matches the right context regular expression. In contrast, when Invisible Jacc splits a context string into two substrings, the first substring always matches the token regular expression, and the second substring always matches the right context regular expression.

3.8.6 Example: Using Link Names

Link names are used in attaching token factories to token definitions. They let you associate a different token factory object with each token definition. Whenever the scanner recognizes a token, the scanner calls the corresponding token factory.

The following fragment illustrates three token definitions with the same token name and different link names.

```
%tokens:
number {integer} = decDigit+;
number {fixedPoint} = decDigit* '.' decDigit* @ decDigit;
number {illegal} = (decDigit | '.')+;
```

The following code fragment shows how to attach a different token factory class to each of the three token definitions. The function `linkFactory` (which is defined in class `ScannerTable`), accepts three parameters: a string that specifies the token name, a string that specifies the link name, and an object of class `TokenFactory`.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("number", "integer",
         new NumberInteger ());
      _scannerTable.linkFactory ("number", "fixedPoint",
         new NumberFixedPoint ());
      _scannerTable.linkFactory ("number", "illegal",
         new NumberIllegal ());
      ...
   }
   ...

final class NumberInteger extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {
      // Handle token number {integer}
      ...
   }
}

final class NumberFixedPoint extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {
      // Handle token number {fixedPoint}
      ...
   }
}

final class NumberIllegal extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {
      // Handle token number {illegal}
      ...
   }
}
...
}  // End class Compiler
```

Assuming that `'number'` is also a terminal symbol in the grammar, any of the three token factories can assemble a token corresponding to the `'number'` terminal symbol. The token factory `NumberIllegal` would probably print an error message and then assemble a token with `null` value. The other two token factories would probably convert the token text into binary and then assemble a token containing the binary value.

3.8.7 Example: Case-Sensitive Keywords

Many programming languages use keywords. The most sensible way to handle keywords is to define each one as a separate token. Then, each keyword becomes a separate terminal symbol of the grammar.

In Java, keywords are case-sensitive. The following fragment illustrates token definitions for several Java keywords.

```
// Case-sensitive keywords

%categories:
a = 'a';
b = 'b';
c = 'c';
/* and so on for d through y */
z = 'z';

%tokens:
public = p u b l i c;
private = p r i v a t e;
switch = s w i t c h;
do = d o;
```

These keywords are case-sensitive. For example, the token `'switch'` is matched by the string “`switch`” but not by “`Switch`” or “`SWITCH`”.

3.8.8 Example: Case-Insensitive Keywords

If you want to define keywords that are case-insensitive, the easiest way to do so is to create character categories that include both uppercase and lowercase letters. The following fragment illustrates the technique.

```
// Case-insensitive keywords

%categories:
aA = 'aA';
bB = 'bB';
cC = 'cC';
/* and so on for dD through yY */
zZ = 'zZ';
```

```
%tokens:
public = pP uU bB lL iI cC;
private = pP rR iI vV aA tT eE;
switch = sS wW iI tT cC hH;
do = dD oO;
```

These keywords are case-insensitive. For example, the token `'switch'` is matched by the strings “`switch`”, “`Switch`”, “`SWITCH`”, and “`sWItcH`”.

The same technique can be used to allow accented characters or alternate Unicode characters. Simply define each character category to include all the characters that are acceptable forms of the letter.

3.8.9 Example: Numeric Formats

The following example shows how to define tokens that recognize common numeric formats.

```
%categories:
decDigit = '0'..'9';
hexDigit = '0'..'9' | 'a'..'f' | 'A'..'F';
eE = 'eE';
xX = 'xX';
sign = '+-';
'.' = '.';

%tokens:
decimalInteger = decDigit+;
hexInteger = 0 xX hexDigit+;
fixedPoint = decDigit* '.' decDigit* @ decDigit;
floatingPoint = (decDigit* '.'? decDigit* @ decDigit) eE sign? DecDigit+;
```

In this example:

- A decimal integer is defined as a series of one or more decimal digits.
- A hexadecimal integer is defined as the characters “`0x`” or “`0X`”, followed by one or more hexadecimal digits.
- A fixed-point number is defined as a series of one or more decimal digits that contains a decimal point. Notice the use of the inclusion operator `@` to guarantee that the token contains at least one decimal digit.
- A floating-point number is defined as a mantissa followed by an exponent. The mantissa is a series of decimal digits, which optionally contains a decimal point. The exponent is the letter “`e`” or “`E`”, optionally followed by a sign, followed by a series of one or more decimal digits. Again, notice the use of the inclusion operator `@` to guarantee that there is at least one decimal digit in the mantissa.

Many programming languages do not distinguish between fixed-point and floating-point numbers. In this case, you could combine the two into a single token, like this:

```
%tokens:
floating = decDigit* '.' decDigit* @ decDigit
           | (decDigit* '.'? decDigit* @ decDigit) eE sign? DecDigit+;
```

3.8.10 Example: Identifiers

Most programming languages have identifiers, which represent variable names, function names, or other objects. Typically, an identifier is defined as a series of letters or digits, in which the first character is a letter. Programming languages differ in what constitutes a “letter” or a “digit”.

The following example shows how to define identifiers that may contain any Unicode letters or digits.

```
// Unicode identifiers

%categories:
letter = %letter;
digit = %digit;

%tokens:
identifier = letter (letter | digit)*;
```

The following example shows how to define identifiers that may contain ASCII letters or digits.

```
// ASCII identifiers

%categories:
letter = 'a'..'z' | 'A'..'Z';
digit = '0'..'9';

%tokens:
identifier = letter (letter | digit)*;
```

C++ uses ASCII identifiers, but an underscore is considered to be a letter.

```
// C++ identifiers

%categories:
letter = 'a'..'z' | 'A'..'Z' | '_';
digit = '0'..'9';

%tokens:
identifier = letter (letter | digit)*;
```

Java uses Unicode identifiers, but underscore and dollar sign are considered to be letters.

```
// Java identifiers

%categories:
letter = %letter | '_$';
digit = %digit;

%tokens:
identifier = letter (letter | digit)*;
```

3.8.11 Example: Error Tokens

It is often necessary to define *error tokens* to recognize errors in the input. This is necessary because sometimes erroneous input can be broken down into legal tokens in inappropriate ways.

For example, let’s try to define tokens to recognize both identifiers and integers.

```
// This doesn’t work right

%categories:
letter = %letter;
digit = %digit;
decDigit = '0'..'9';
```

```
%tokens:
identifier = letter (letter | digit)*;
decimalInteger = decDigit+;
```

With these definitions, the string “95” will be recognized as a decimal integer, while the string “ABC” will be recognized as an identifier.

But what about the string “95ABC”? This string does not match either token. In this case, the scanner searches for the longest initial substring that matches a token. The substring “95” matches 'decimalInteger', so the scanner splits off the substring “95” and considers it to be a token. Then, the scanner recognizes the remaining substring “ABC” as a second token which matches 'identifer'.

So, the scanner splits the string “95ABC” into two legal tokens, “95” and “ABC”. This is almost certainly not what we want the scanner to do. We would like for the scanner to report an error when it encounters “95ABC”.

To detect the error, we must introduce an *error token* into the specification. The error token detects combinations of letters and digits that are not legal identifiers or integers. Here is how to do it:

```
// Example of error token

%categories:
letter = %letter;
digit = %digit;
decDigit = '0'..'9';

%tokens:
identifier = letter (letter | digit)*;
decimalInteger = decDigit+;
badIdentifier = (letter | digit)+;
```

We have added a third token called 'badIdentifier' which matches *any* string of letters and digits. In particular, 'badIdentifier' matches legal identifiers and integers, as well as illegal combinations. This is OK because when an input string matches more than one token definition, the scanner gives priority to the token definition that appears earliest in the specification file.

In this case, we have listed 'badIdentifier' *after* the token definitions for legal identifier and integers. Therefore, legal identifiers and integers match up with 'identifier' and 'decimalInteger', while every other combination of letters and digits “falls through” to match 'badIdentifier'.

With these definitions, the string “95” will be recognized as a decimal integer, the string “ABC” will be recognized as an identifier, and the string “95ABC” will be recognized as invalid.

As this example illustrates, error tokens should appear *after* the corresponding legal tokens. That way, the error tokens won’t interfere with the legal tokens.

Every error token should have a token factory that issues an error message. After issuing an error message, the token factory could either discard the token or assemble the token. Assembling the token would be reasonable if the form of the error token is close enough to a legal token so that you can guess what the user intended to write. If you can guess the type of token but not the token’s value, you could assemble a token with null value, which is the signal for an *error insertion symbol*.

3.8.12 Example: Numeric Suffixes

In a Java program, you can append the suffix L to a literal integer to indicate that it is of type long rather than type int. For example, 23 is a literal int, but 23L is a literal long.

C++ supports the suffix L, and also supports the suffix U to indicate that the literal integer is of type unsigned rather than signed.

Suppose you're writing a Java compiler. Given that many Java programmers are also C++ programmers, you might expect that a common error in Java code is to write a suffix `U`. This example shows how to use error tokens to detect this particular error, and issue a specific error message. Here is how to do it:

```
// Detect an illegal suffix U

%categories:
decDigit = '0'..'9';
lL = 'lL';
uU = 'uU';

%tokens:
intLiteral = decDigit+;
longLiteral = decDigit+ lL;
intLiteral {unsigned} = decDigit+ uU;
longLiteral {unsigned} = decDigit+ (lL uU | uU lL);
```

We have use the link name `'unsigned'` to identify the tokens that contain the illegal suffix `U`. This lets us write separate token factories for these tokens.

The token factory for an `'unsigned'` token should issue an error message saying that the suffix `U` is not allowed in Java. Then, it should go ahead and assemble an `'intLiteral'` or `'longLiteral'` token, since it is clear what the user intended to write.

3.8.13 Example: Java Strings

In a Java program, you can write a literal string that is enclosed in double quotes. You can also use *escape sequences* to insert special characters into the string. The following escape sequences are supported:

`\b`	backspace
`\t`	tab
`\n`	line feed
`\f`	form feed
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash
`\digits`	octal escape, characters `0x00` through `0xFF`

The following shows the tokens to recognize strings. This is excerpted from file `Ex1Grammar.jacc`.

```
// Java string literals

%categories:
'"' = '"';
'\' = '\';
stringChar = %any - '"' - '\' - 10 - 13;

%tokens:
stringLiteral =
   '"' (stringChar | '\' '\' | '\' '"' | '\' stringChar)* '"';
stringLiteral {runOn} =
   '"' (stringChar | '\' '\' | '\' '"' | '\' stringChar)* '\'?;
```

The first token recognizes Java string literals. The second token is an error token that catches *run-on strings*, that is, strings that do not have a closing double quote. Notice the optional backslash at the end of the second token, to catch run-on strings that end in the middle of an escape sequence.

The following code demonstrates how to write the token factories for these tokens. This code is very similar to code in `Ex1Compiler.java`. The token factory for legal strings processes all the escape codes, and assembles a token whose value is the resulting `String` object. If an invalid escape sequence is encountered, the token factory prints an error message and assembles a token with `null` value. The token factory for run-on strings prints an error message, and then assembles a token with `null` value. Recall that `null` value is the signal for an error-insertion symbol.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("stringLiteral", "",
         new StringLiteral ());
      _scannerTable.linkFactory ("stringLiteral", "runOn",
         new StringLiteralRunOn ());
      ...
   }
   ...

final class StringLiteral extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Get the contents of the string literal into a char array

      char[] stringChars = new char[scanner.tokenLength() - 2];
      scanner.tokenToChars (1, scanner.tokenLength() - 2, stringChars, 0);

      // Process escape sequences

      boolean badEscape = false;
      int srcIndex = 0;
      int dstIndex = 0;
      while (srcIndex < stringChars.length)
      {

         // Get the next character

         char c = stringChars[srcIndex++];

         // If it's the start of an escape sequence ...

         if (c == '\\')
         {

            // Get the next character. Note that we can't be past the end
            // of the array because the regular expression won't match a
            // string that ends in the middle of an escape sequence.
            c = stringChars[srcIndex++];

            // Switch on the escape code ...

            switch (c)
            {
            case 'b':
               c = 0x0008;
               break;

            case 't':
               c = 0x0009;
               break;

            case 'n':
               c = 0x000A;
               break;

            case 'f':
               c = 0x000C;
               break;

            case 'r':
               c = 0x000D;
               break;

            case '\"':
            case '\'':
            case '\\':
               break;

            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':

               // First octal digit

               c = (char)(c - '0');

               if ((srcIndex < stringChars.length)
                  && (stringChars[srcIndex] >= '0')
                  && (stringChars[srcIndex] <= '7') )
               {

                  // Second octal digit

                  c = (char)((c << 3) + (stringChars[srcIndex++] - '0'));

                  if ((c <= 0x001F)
                     && (srcIndex < stringChars.length)
                     && (stringChars[srcIndex] >= '0')
                     && (stringChars[srcIndex] <= '7') )
                  {

                     // Third octal digit

                     c = (char)((c << 3)
                         + (stringChars[srcIndex++] - '0'));
                  }
               }
               break;

            default:

               // Invalid escape sequence

               token.column += (srcIndex - 1);
               reportError (token, null,
                  "Invalid escape sequence '"
                  + new String (stringChars, srcIndex - 2, 2) + "'." );
               token.column -= (srcIndex - 1);
               badEscape = true;
               break;

            }  // end switch on escape code

         }  // end if escape sequence

         // Write the character at the destination index

         stringChars[dstIndex++] = c;

      }  // end loop over source characters

      // If not a bad escape sequence, convert value to String

      if (!badEscape)
      {
         token.value = new String (stringChars, 0, dstIndex);
      }

      // Assembled token

      return assemble;
   }
}

final class StringLiteralRunOn extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Report the error

      reportError (token, null, "Unterminated string.");

      // Assemble error-insertion token (with null value)
```

```
        return assemble;
    }
}
...
}  // End class Compiler
```

3.8.14 Example: Include Files

Most programming languages have an *include* command that allows you to incorporate text from other source files into the current compilation. For example, in C++ this is done with the `#include` directive. Java is somewhat unusual in lacking an include command.

The best way to handle an include command is to define a token to recognize the command. The semantics of the include token are defined as follows: the included source file is scanned into tokens, and all the tokens of the included file are inserted into the token stream in place of the original include token. Notice that an individual token is not allowed to extend from one source file into another.

In other words, an include token says: “take all the tokens in this file, and insert them into the token stream here.”

Of course, include commands can be nested. That is, the included source file may itself contain include commands, and so on to any depth.

In this example, we show how to implement an include command. The following shows how to define a token to recognize an include command. This is excerpted from file `Ex1Grammar.jacc`.

```
// Include command

%categories:
c = 'c';
d = 'd';
e = 'e';
i = 'i';
l = 'l';
n = 'n';
u = 'u';
'*' = '*';
'/' = '/';
space = 9 | 12 | 32;       // tab, form feed, and space
notEol = %any - 10 - 13;   // any character that isn't a line end
'<' = '<';
'>' = '>';
filenameChar = 0x0020..0xFFFF - '<>';
filenameCharNotBlank = 0x0021..0xFFFF - '<>';

%tokens:
include =
        i n c l u d e (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
        '<' space* filenameCharNotBlank filenameChar* space* '>';
illegalInclude =
        i n c l u d e (space | '/' '*' (notEol* ~ '*' '/') '*' '/')*
        ('<'? space* filenameCharNotBlank* | (notEol* ~ '>') '>');
```

In this example, an include command consists of the keyword “`include`”, followed by a filename enclosed in angle brackets. There may be any amount of white space and/or comments between the “`include`” keyword and the left angle bracket. There may be white space (but not comments) before and after the filename. The filename may contain any characters from `0x0020` through `0xFFFF`, except for the angle bracket characters. The entire include command must be written on a single line.

The following are some examples of include commands as they might appear in the source file:

```
include<MyFile.txt>
include  <MyFile.txt>
include  <  MyFile.txt  >
include  /* silly comment */  <MyFile.txt>
include  <File name with spaces.txt>
```

We have defined two tokens. The token `'include'` represents legal include commands. The token `'illegalInclude'` is an error token that attempts to catch likely malformations of the include command. There is no hard-and-fast rule for how to write the error token for a case like this, since there can be many different malformations, and it is hard to know how much additional input beyond the keyword "`include`" ought to be sucked up by the error token. Here, the error token extends to the first `'>'` character on the current line. If there is a no `'>'` character on the current line, then the error token will suck up a `'<'` character and/or a word that resembles a filename.

The following code demonstrates how to write the token factories for these tokens. This code is very similar to code in `Ex1Compiler.java`.

```
public class Compiler extends CompilerModel
{
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("include", "",
         new Include ());
      _scannerTable.linkFactory ("illegalInclude", "",
         new IllegalInclude ());
      ...
   }
   ...

final class Include extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Get the token text as a string

      String includeText = scanner.tokenToString ();

      // Extract the substring that contains the filename

      String filename = includeText.substring (
         includeText.lastIndexOf('<') + 1,
         includeText.lastIndexOf('>') ).trim();

      // Make a scanner for this filename

      Scanner includeScanner = makeScanner (filename);

      // If we couldn't open the file, report error and throw exception

      if (includeScanner == null)
      {
         reportError (token, null,
            "Unable to open include file '" + filename + "'." );
         throw new FileNotFoundException (filename);
      }

      // Construct an include-file escape token

      token.number = Token.escapeInsertStream;
      token.value = includeScanner;

      // Assembled token

      return assemble;
   }
}

final class IllegalInclude extends TokenFactory
{
   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Report the error

      reportError (token, null, "Invalid 'include' statement.");

      // Discard token

      return discard;
   }
}
...
}  // End class Compiler
```

The token factory for `'include'` first extracts the filename from the matching token text. Then, it creates a scanner for the included file. The scanner is created by the function `makeScanner`, which is defined in class `CompilerModel`. If there is some error in creating the scanner, the token factory prints an error message and then throws an exception to abort scanning, since there is no point in continuing to scan the current input if the include file is missing.

If the scanner is created successfully, the token factory must then insert all its tokens into the token stream. It does this by assembling a special token called an *insert stream escape token*. It does this by setting `token.number` to the special value `Token.escapeInsertStream`, and setting `token.value` to the scanner object.

The escape token is intercepted by the preprocessor. (The *preprocessor* is an object that sits in between the scanner and the parser, where it can filter or modify the token stream.) When the preprocessor sees the escape token, it suspends reading from the current scanner, and begins reading from the scanner for the included file. When the included file has been completely scanned, the preprocessor returns to the original source file, resuming right after the `'include'` token.

3.8.15 Example: The Fortran DO Statement

This is the first of three examples that illustrate the use of right context.

Perhaps the most famous example of right context is the Fortran `DO` statement. Consider the following two Fortran statements.

```
    DO 10 I = 1.25
    DO 10 I = 1, 25
```

These are both valid Fortran. The first statement is an assignment statement, which assigns the value `1.25` to the variable `DO10I`. (In Fortran, blanks are ignored, so variable names can be written with embedded blanks.) The second is a `DO` statement, introducing a loop in which the variable `I` will range from `1` to `25`.

`DO` is a keyword in Fortran, and we would like to have a token that recognizes the `DO` keyword. Unfortunately, when we see the letters `D` and `O`, we don't know that it's a `DO` keyword until later, when we see the comma in the `DO` statement.

The following shows how to write a token to recognize the Fortran `DO` keyword:

```
// Fortran DO

%categories:
letter = 'a'..'z' | 'A'..'Z';
digit = '0'..'9';
',' = ',';
'=' = '=';
D = 'dD';
O = 'oO';

%tokens:
DO = D O / (letter | digit)+ '=' (letter | digit)+ ',';
```

The token definition for `DO` contains two regular expressions, separated by a slash. The first is the regular expression for the keyword `DO` itself, which consists of the letters `D` and `O`.

The second regular expression is the *right context*. When the scanner sees the letters `D` and `O`, it matches them to the `DO` token only if the following text matches the right context. In this case, the scanner matches the letters `D` and `O` to the `DO` token only if they are followed by more letters or digits, then an equal sign, then more letters or digits, and then a comma. This is sufficient to ensure that the letters `D` and `O` actually introduce a `DO` statement, and not an assignment statement.

In the two Fortran statements given previously, the `DO` token would match the letters `D` and `O` appearing in the second statement, but not the first statement.

3.8.16 Example: The Range Operator

A number of programming languages use the operator '`..`' to indicate a range of values. For example, the expression "`7..23`" would denote an integer that can range from `7` to `23`.

You have already seen that Invisible Jacc uses the operator '`..`' to denote a range of values in a category expression. Other languages can use this operator to denote array bounds, switch cases, or loop index limits.

A problem arises in scanning the range operator if the language also allows fixed-point or floating-point numbers. Consider the following:

```
// This doesn't work

%categories:
decDigit = '0'..'9';
'.' = '.';

%tokens:
'..' = '.' '.';
decimalInteger = decDigit+;
fixedPoint = decDigit* '.' decDigit* @ decDigit;
```

Now look at what happens if the scanner encounters the expression "`7..23`". The scanner looks for the longest initial substring that matches any token definition. In this case, the scanner finds that "`7.`" matches the token definition for `'fixedPoint'`. Then the scanner looks at the remaining input and finds that "`.23`" also matches the token definition for `'fixedPoint'`. So the scanner interprets the input "`7..23`" as two fixed-point numbers: "`7.`" and "`.23`". This is not what we want.

The problem is that when decimal digits are followed by two dots, we want them to be interpreted as an integer, not as a fixed-point number. We can achieve this with a right context, like this:

```
// Correctly scans the range operator

%categories:
decDigit = '0'..'9';
'.' = '.';

%tokens:
'..' = '.' '.';
decimalInteger = decDigit+;
decimalInteger = decDigit+ / '.' '.';
fixedPoint = decDigit* '.' decDigit* @ decDigit;
```

We've added a second token definition for `'decimalInteger'`, this one with a right context consisting of two dots.

Now let's see what happens when the scanner encounters the expression "`7..23`". The scanner looks for the longest initial substring that matches any token definition. In this case, the scanner finds that "`7..`" matches the second token definition for `'decimalInteger'`. Notice that in choosing the longest substring, the input that matches the right context is included in the calculation. This is an important point. When right contexts are involved, the scanner chooses the token definition that looks furthest ahead, even if that token definition does not ultimately produce the longest token.

After matching "`7..`" to the second token definition for `'decimalInteger'`, the scanner strips off the right context and returns "`7`" as the first token. Then the scanner looks at the remaining input, which is "`..23`". The longest initial substring that matches any token definition is "`..`", which matches the token definition for the range operator. The remaining input is now "`23`", which matches the first token definition for `'decimalInteger'`.

So, the scanner interprets the input "`7..23`" as three tokens: the decimal integer "`7`", the operator "`..`", and the decimal integer "`23`". This is exactly what we want.

As a final remark, note that we could combine the two token definitions for `'decimalInteger'` into one, by using the optional closure operator as follows:

```
decimalInteger = decDigit+ / ('.' '.')?;
```

3.8.17 Example: Detecting End-of-File

In some cases it is necessary to determine whether or not a token is the last token in the file. For example, in some operating systems (notably MS-DOS), it is common for text editors to append an ASCII sub character at the end of each text file. The *sub character* is character code 0x1A, and is also referred to as *control-Z.* If you're scanning such files, you will probably want to treat a sub character occurring at the end of a file differently from a sub character occurring elsewhere in the file.

We can use a right context to distinguish between a sub character at the end of a file, and a sub character elsewhere in the file.

```
%categories:
sub = 0x1A;
any = %any;
```

```
%tokens:
subNotAtEOF = sub / any;
subAtEOF = sub;
```

A sub character that is not at the end of the file always matches `'subNotAtEOF'`. That’s because the right context `'any'` matches whatever character happens to appear after the sub character. The scanner chooses `'subNotAtEOF'` instead of `'subAtEOF'`, because the former matches a two-character string, while the latter matches a one-character string. Remember, the scanner always chooses the token definition that is matched by the longest string, and the right context counts in determining the longest string.

A sub character that is at the end of the file always matches `'subAtEOF'`. That’s because there is no succeeding character to match the right context ‘any’.

3.8.18 Example: Rejecting a Token

Regular expressions are very powerful, but they can’t recognize every possible token. Invisible Jacc has several facilities for scanning tokens that can’t be recognized by regular expressions alone. The general idea is to write a regular expression to recognize *candidate* tokens. Then, the token factory can make the final determination as to which input strings are valid tokens.

This is the first of three examples of recognizing tokens that aren’t defined by regular expressions. In each example, we will show a way to recognize palindromes. A *palindrome* is a series of letters that reads the same forwards and backwards. For example, “`abcba`” is a palindrome, but “`abcbb`” is not a palindrome. It is well known that regular expressions cannot recognize palindromes.

In this example, we write a regular expression that recognizes any series of letters.

```
%categories:
letter = %letter;

%tokens:
palindrome = letter+;
```

Next, we write a token factory. The token factory checks to see if the token text is, in fact, a palindrome. If the token text is a palindrome, the token factory assembles the token. If the token text is not a palindrome, the token factory rejects the token. Here is the code for the token factory:

```
final class Palindrome extends TokenFactory
{
   private static boolean isPalindrome (String str)
   {
      for (int i = 0; i < str.length()/2; ++i)
      {
         if (str.charAt(i) != str.charAt(str.length() - 1 – i))
         {
            return false;
         }
      }
      return true;
   }

   public int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException
   {

      // Get the token text as a string
```

```
        String tokenText = scanner.tokenToString ();

        // If string is a palindrome, assemble the token

        if (isPalindrome (tokenText))
        {
           token.value = tokenText;
           return assemble;
        }

        // Otherwise, reject the token

        return reject;
     }
  }
```

When a token factory rejects a token, the scanner behaves exactly as if the input string did not match the token definition. If the input string matches some other token definition, the scanner calls the token factory for the other token definition. Otherwise, the scanner searches for a shorter input string that matches some token definition.

Given the input string "`abcbadde`", the scanner does the following:

1. The scanner calls the token factory with string "`abcbadde`". Since this isn't a palindrome, the token factory rejects the token.
2. There are no other token definitions in this example, so the scanner tries a shorter input string. The scanner calls the token factory with string "`abcbadd`". This isn't a palindrome, so the token factory rejects it.
3. The scanner calls the token factory with string "`abcbad`". Again, the token factory rejects it.
4. The scanner calls the token factory with string "`abcba`". This is a palindrome, and the token factory assembles the token.
5. The scanner calls the token factory with the remaining input, which is "`dde`". This isn't a palindrome, and the token factory rejects it.
6. The scanner calls the token factory with string "`dd`". This is a palindrome, and the token factory assembles the token.
7. The scanner calls the token factory with the remaining input, which is "`e`". This is a palindrome, and the token factory assembles the token.

So, given input string "`abcbadde`", the scanner interprets it as three tokens: "`abcba`", "`dd`", and "`e`". Notice that each token is a palindrome.

3.8.19 Example: Setting the Token Length

Continuing the previous example, we now write a different token factory to recognize palindromes. The previous token factory merely checked the string to see if it is a palindrome. Our new token factory will determine *how much* of the string is a palindrome. The token factory then calls the function `scanner.setTokenLength` to tell the scanner the actual length of the token.

```
  final class Palindrome extends TokenFactory
  {
     private static boolean isPalindrome (String str)
     {
        ...
```

```
      }

      public int makeToken (Scanner scanner, Token token)
         throws IOException, SyntaxException
      {

         // Get the token text as a string

         String tokenText = scanner.tokenToString ();

         // Loop to determine the actual length of a palindrome

         for (int actualLength = tokenText.length(); ; --actualLength)
         {

            // Get the initial substring with length actualLength

            String actualToken = tokenText.substring (0, actualLength);

            // If the substring is a palindrome, assemble the token

            if (isPalindrome (actualToken))
            {
               scanner.setTokenLength (actualLength);
               token.value = actualToken;
               return assemble;
            }
         }
      }
   }
```

When a token factory assembles or discards a token, the scanner advances its current position by the length of the token text. Normally, this is the length of the string that matched the token regular expression. But a token factory can change this length by calling `scanner.setTokenLength`. Then, the scanner advances its current position by whatever length is specified by the token factory. This can be used in situations where the regular expression cannot detect the actual length of the token.

Given the input string “`abcbadde`”, the scanner does the following:

1. The scanner calls the token factory with string “`abcbadde`”.
2. The token factory determines that the first 5 characters form a palindrome. So it sets the token length to 5, and assembles the token “`abcba`”.
3. The scanner advances its current position by 5 characters. Then it calls the token factory with the remaining input, which is “`dde`”.
4. The token factory determines that the first 2 characters form a palindrome. So it sets the token length to 2, and assembles the token “`dd`”.
5. The scanner advances its current position by 2 characters. Then it calls the token factory with the remaining input, which is “`e`”.
6. The token factory determines that the first 1 character forms a palindrome. So it sets the token length to 1, and assembles the token “`e`”.
7. The scanner advances its current position by 1 character.

So, given input string “`abcbadde`”, the scanner interprets it as three tokens: “`abcba`”, “`dd`”, and “`e`”. Notice that each token is a palindrome.

3.8.20 Example: Assembling Different Types of Tokens

Continuing the previous example, we now approach our palindrome problem in a different way. Suppose that we have a grammar with two different terminal symbols: `'palindrome'`, which represents a word that is a palindrome, and `'notPalindrome'`, which represents a word that is not a palindrome.

We want to scan the input file and locate all the words. A *word* is a series of letters. For each word, we want to generate the terminal symbol `'palindrome'` if the word is a palindrome, or `'notPalindrome'` if it isn’t. To do this, we will define a single token factory that can return two different types of tokens. In all of our previous examples, each token factory has returned only a single type of token.

Here is part of the grammar specification for this example:

```
%terminals:
palindrome;
notPalindrome;

%categories:
letter = %letter;

%tokens:
word = letter+;
```

Next, we show the token factory and part of the compiler. The compiler uses the function `lookupSymbol` (which is defined in class `ParserTable`) to obtain the numerical values for the two terminal symbols `'palindrome'` and `'notPalindrome'`. The token factory checks the token to see if it is a palindrome, and assembles the appropriate terminal symbol.

```
public class Compiler extends CompilerModel
{
   int _symbolPalindrome;
   int _symbolNotPalindrome;
   ...
   public Compiler ()
   {
      ...
      _scannerTable.linkFactory ("word", "", new Word ());
      ...
      _symbolPalindrome =
         _parserTable.lookupSymbol ("palindrome");
      _symbolNotPalindrome =
         _parserTable.lookupSymbol ("notPalindrome");
      ...
   }
   ...

final class Word extends TokenFactory
{
   private static boolean isPalindrome (String str)
   {
      ...
   }
```

```
      public int makeToken (Scanner scanner, Token token)
         throws IOException, SyntaxException
      {

         // Get the token text as a string

         String tokenText = scanner.tokenToString ();

         // If string is a palindrome, assemble 'palindrome' symbol

         if (isPalindrome (tokenText))
         {
            token.number = _symbolPalindrome;
            token.value = tokenText;
         }

         // Otherwise, assemble 'notPalindrome' symbol

         else
         {
            token.number = _symbolNotPalindrome;
            token.value = tokenText;
         }

         // Assembled the token

         return assemble;
      }
   }
   ...
   }  // End class Compiler
```

The variable `token.number` contains the numerical value of the token. By setting this value, the token factory can return any type of token, and therefore any terminal symbol.

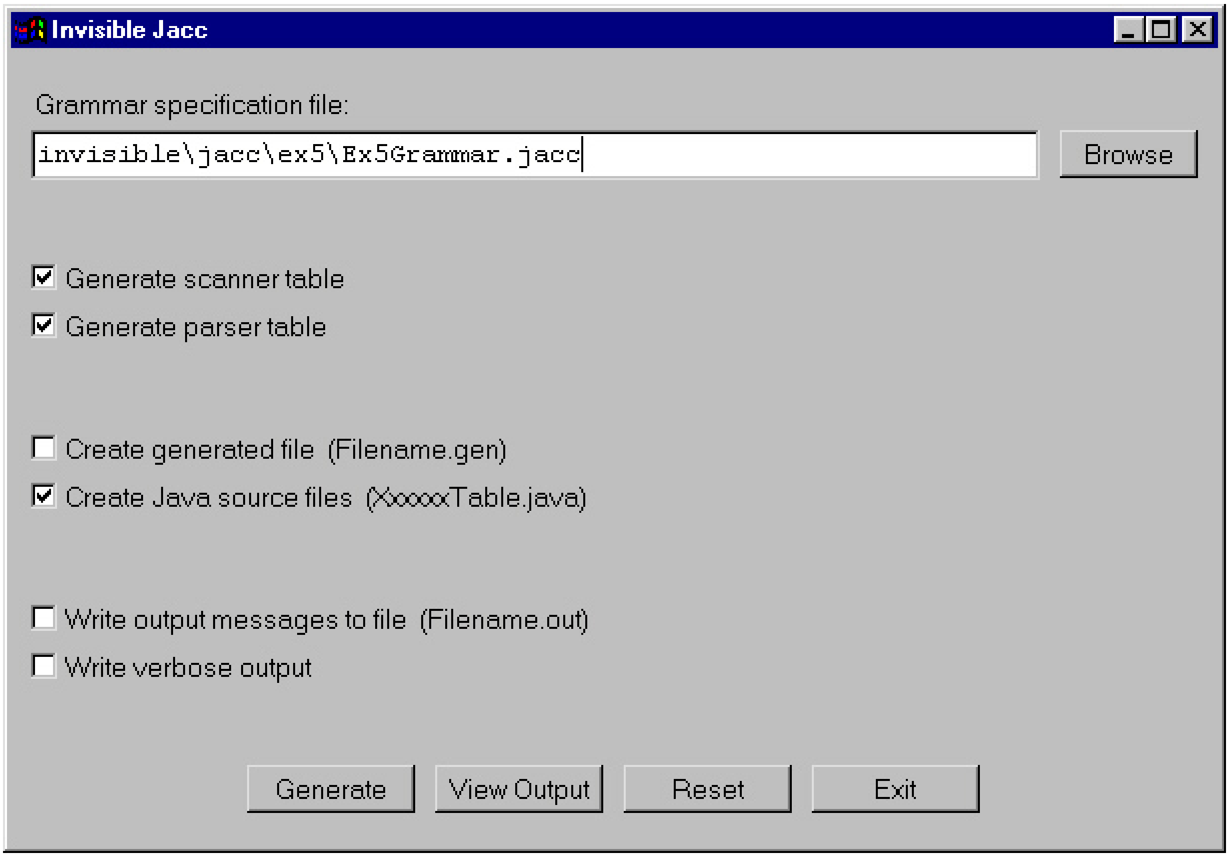
4 Running the Parser Generator

There are two ways to run the parser generator. You can use the Invisible Jacc graphical user interface, or you can use the Invisible Jacc command line.

The graphical interface is usually more convenient, but you can use whichever method you prefer.

4.1 The Invisible Jacc Graphical User Interface

The class `invisible.jacc.gen.GenGUI` provides the graphical user interface. To start the graphical user interface, execute class `invisible.jacc.gen.GenGUI`. The following window appears:


Invisible Jacc

Grammar specification file: invisible\jacc\ex5\Ex5Grammar.jacc

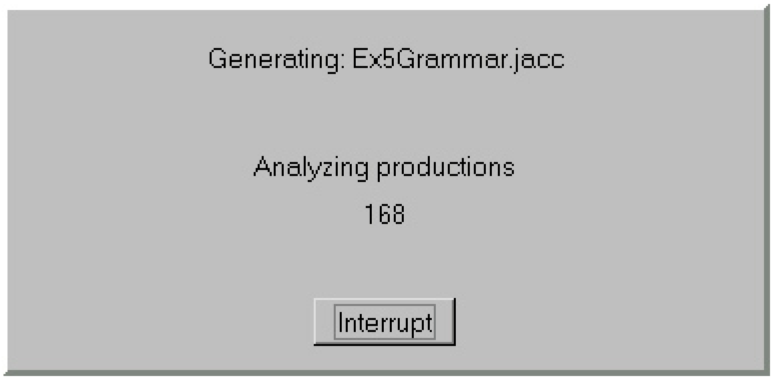
- [✓] Generate scanner table
- [✓] Generate parser table
- [ ] Create generated file (Filename.gen)
- [✓] Create Java source files (XxxxxTable.java)
- [ ] Write output messages to file (Filename.out)
- [ ] Write verbose output

The window starts by displaying the *input panel*. On the input panel, you can do the following:

- Enter the name of the grammar specification file in the text field. On Microsoft Windows, the file name is not case-sensitive. (On other operating systems, the file name may or may not be case-sensitive.) You may optionally include the extension "`.jacc`" on the filename; if you don't type an extension, Invisible Jacc appends the extension "`.jacc`" automatically.
- Click **Browse** to bring up a dialog box for you to select the grammar specification file.
- Check **Generate scanner table** if you want to generate scanner tables for your grammar.
- Check **Generate parser table** if you want to generate parser tables for your grammar.

- Check **Create generated file** if you want to save your scanner and parser tables to disk in binary format. The name of the generated file is obtained by appending the extension "`.gen`" to the name of the grammar specification file. The resulting file contains first the scanner table, and then the parser table. You can use the method `CompilerModel.readGenFile` to read the generated file. (You can also use `ScannerTable.readFromStream` and `ParserTable.readFromStream`.)
- Check **Create Java source files** if you want to save your scanner and parser tables to disk as Java source files. If you use this option, then you must have a `%java` statement in your grammar specification file. The Java package is the package name from the `%java` statement. The class name for the scanner table is the class name from the `%java` statement, concatenated with "`ScannerTable`". The class name for the parser table is the class name from the `%java` statement, concatenated with "`ParserTable`". The source files are written in the same directory as the grammar specification file, and each source file has the extension "`.java`".
- Check **Write output messages to file** if you want to save the output messages to a text file on disk. The name of the output file is obtained by appending the extension "`.out`" to the name of the grammar specification file. (The output messages are also displayed on-screen, regardless of whether or not you check this box.)
- Check **Write verbose output** if you want Invisible Jacc to write out a complete description of the scanner and parser tables.
- After you've entered the file name and selected the options you want, click **Generate** to start Invisible Jacc and generate the tables.
- Click **View Output** to switch to the output panel.
- Click **Reset** to clear the text field and reset all the checkboxes to their default values.
- Click **Exit** to exit from Invisible Jacc.

When you click **Generate**, Invisible Jacc begins to generate the scanner and parser tables. While it is working, it displays a *progress window* to let you know what it is doing. The progress window looks like this:

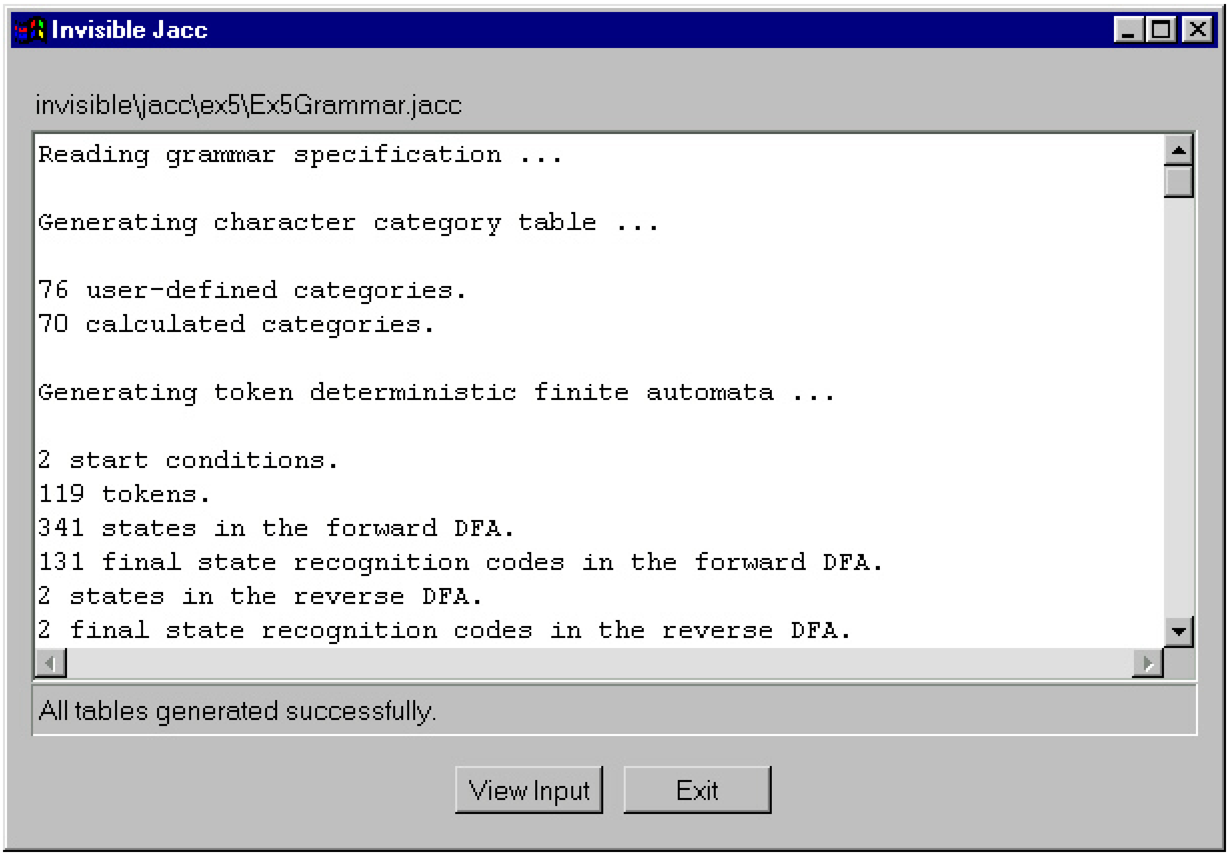

Generating: Ex5Grammar.jacc

Analyzing productions

168

While Invisible Jacc is busy generating tables, you can click **Interrupt** to stop it.

When table generation is complete, Invisible Jacc displays the *output panel* to show you the results. The output panel looks like this:



invisible\jacc\ex5\Ex5Grammar.jacc

```
Reading grammar specification ...

Generating character category table ...

76 user-defined categories.
70 calculated categories.

Generating token deterministic finite automata ...

2 start conditions.
119 tokens.
341 states in the forward DFA.
131 final state recognition codes in the forward DFA.
2 states in the reverse DFA.
2 final state recognition codes in the reverse DFA.
```

All tables generated successfully.

The output panel shows the name of the grammar specification file at the top, the output messages in a large text area, and a one-line summary at the bottom.

On the output panel, you can do the following:

- Use the scroll bars to scroll through the output messages.
- Use the keyboard to scroll through the output messages. The following keys are supported:

Up Arrow	Scroll up one line.
Down Arrow	Scroll down one line.
Ctrl + Up Arrow	Scroll up one screen.
Ctrl + Down Arrow	Scroll down one screen.
Page Up	Scroll up one screen.
Page Down	Scroll down one screen.
Ctrl + Page Up	Go to the start of the messages.
Ctrl + Page Down	Go to the end of the messages.
Right Arrow	Scroll right one column.
Left Arrow	Scroll left one column.
Ctrl + Right Arrow	Scroll right 20 columns.
Ctrl + Left Arrow	Scroll left 20 columns.
Home	Go to the leftmost column.
End	Go to the rightmost column.

- Click **View Input** to switch back to the input panel.
- Click **Exit** to exit from Invisible Jacc.

Note: On the output panel, the keyboard only works when the text area has the input focus. If the keyboard doesn't work, try clicking the mouse in the text area; this should give the input focus to the text area.

Note: By default, Invisible Jacc uses 14-point fonts in the graphical user interface. If these fonts are uncomfortable on your system, either too large or too small, you can change the font size. Simply enter the desired font size on the `invisible.jacc.gen.GenGUI` command line. For example, on Microsoft Windows the following command starts the Invisible Jacc graphical user interface with 12-point fonts:

```
jview invisible.jacc.gen.GenGUI 12
```

4.2 The Invisible Jacc Command Line

Invisible Jacc includes a simple command-line interface to the parser generator. The Invisible Jacc command line is provided by class `invisible.jacc.gen.GenMain`.

To invoke the parser generator, execute class `invisible.jacc.gen.GenMain` with the following command line:

```
[-v] [-o] [-g] [-s] [-p] [-j] grammar-file
```

Square brackets indicate options. The options may appear on the command line in any order; you don't have to write them in the order shown. The following table explains each item on the command line.

`grammar-file`	The name of the grammar specification file. On Microsoft Windows, the file name is not case-sensitive. (On other operating systems, the file name may or may not be case-sensitive.) You may optionally include the extension "`.jacc`" on the filename; if you don't type an extension, Invisible Jacc appends the extension "`.jacc`" automatically.
`-v`	Verbose mode. If you include this option, Invisible Jacc writes out a complete description of the scanner and parser tables.
`-o`	Create output file. If you include this option, Invisible Jacc writes all output messages to an output file. The name of the output file is obtained by appending the extension "`.out`" to the name of the grammar specification file. If you don't include this option, Invisible Jacc writes output messages to the standard output.
`-g`	Create generated file. If you include this option, the scanner and parser tables are written, in binary form, to a generated file. The name of the generated file is obtained by appending the extension "`.gen`" to the name of the grammar specification file. The resulting file contains first the scanner table, and then the parser table. You can use the method `CompilerModel.readGenFile` to read the generated file. (Or, you can use `ScannerTable.readFromStream` and `ParserTable.readFromStream`.) If you don't include this option, then no generated file is created.
`-s`	Create only the scanner table. Do not create a parser table.
`-p`	Create only the parser table. Do not create a scanner table.
`-j`	Create Java source files for the scanner and parser tables. If you include this option, then you must have a `%java` statement in your grammar specification file. The Java package is the package name from the `%java` statement. The class name for the scanner table is the class name from the `%java` statement, concatenated with "`ScannerTable`". The class name for the parser table is the class name from the `%java` statement, concatenated with "`ParserTable`". The source files are written in the same directory as the grammar specification file, and each source file has the extension "`.java`".

You can use any combination of options, with one exception: you can't use both the `-s` and the `-p` option.

For example, suppose you write the following command line:

```
–o –g –j acme\compiler\MyGrammar
```

In Microsoft Windows, you could use the `jview` command at the `C:>` prompt to issue the command line, like this:

```
jview invisible.jacc.gen.GenMain –o –g –j acme\compiler\MyGrammar
```

Let's also suppose that you have the following `%java` statement in your grammar specification file:

```
%options:
%java acme.compiler.MyGrammar;
```

Then:

- The parser generator reads the grammar specification from the file `acme\compiler\MyGrammar.jacc`.
- The parser generator writes all output messages to the file `acme\compiler\MyGrammar.out`.
- The parser generator writes binary forms of the scanner and parser tables to the generated file `acme\compiler\MyGrammar.gen`.
- The parser generator creates a Java source file for the scanner table. The Java source file name is `acme\compiler\MyGrammarScannerTable.java`, the Java package name is `acme.compiler`, and the Java class name is `MyGrammarScannerTable`.
- The parser generator creates a Java source file for the parser table. The Java source file name is `acme\compiler\MyGrammarParserTable.java`, the Java package name is `acme.compiler`, and the Java class name is `MyGrammarParserTable`.

5 Additional Examples

In this chapter we describe the additional examples that are included with Invisible Jacc. Because of their length, we won't print out all the examples. Instead, we will refer you to the disk files, and mention some of the highlights.

5.1 Verbose Output

File `invisible\jacc\ex2\Ex2GrammarVerbose.out` contains a verbose output listing for the simple calculator language presented at the start of this manual. You can look at this file for an example of what a verbose output listing looks like.

5.2 Debug Output

File `invisible\jacc\ex2\Ex2InputDebug.out` contains debug output for the simple calculator language presented at the start of this manual. You can look at this file for an example of what debug output looks like.

To create debug output, you need to edit `Ex2Compiler.java` and change the value of the `_debug` field to `true`. Then, recompile `Ex2Compiler.java` and run it on the sample input file `Ex2Input.txt`.

The debug output contains a trace of all scanner and parser activity. Every time the scanner recognizes a token, it outputs a message containing:

- The name of the input file.
- The line number and column number where the token is located.
- The name of the token (from the grammar specification).
- The token's link name (from the grammar specification).
- The token's parameter (from the grammar specification).
- The token text (from the input file).

Every time the parser reduces a production, it outputs a message containing:

- The name of the input file.
- The current line number and column number.
- The symbol on the production's left hand side (from the grammar specification).
- The production's link name (from the grammar specification).
- The production's parameter (from the grammar specification).

Trace output is useful in debugging a grammar specification. You can use trace output to watch the operation of the scanner and parser, and make sure they are recognizing tokens and reducing productions as you expect them to.

5.3 Scientific Calculator

Directory `invisible\jacc\ex1` contains a compiler for a scientific calculator language. The compiler does all calculations in double-precision floating point, and supports:

- Addition, subtraction, multiplication, and division.
- Exponentiation.
- Square root.
- Named variables.

- Ability to print both numerical values and literal strings.
- Both line comments and C-style comments.

This example illustrates the following techniques:

- Using `%shift` and `%reduce` clauses to define operator precedence and associativity.
- Using production parameters to allow a single nonterminal factory to handle several different productions.
- Assigning error repair costs to terminal symbols.
- Using start conditions to process multi-line C-style comments.
- Writing token definitions to recognize keywords.
- Writing token definitions to recognize various numeric formats.
- Writing token definitions and token factories to recognize Java-style string literals.
- Implementing an “include” directive.
- Counting lines.
- Writing a compiler class that does not inherit from `CompilerModel`.
- Maintaining a table of variables and their values.

5.4 Parsing Conflicts

Directory `invisible\jacc\ex3` contains several grammar specifications with parsing conflicts. They are provided to illustrate how the parser generator reports conflicts.

We will briefly describe one of the examples. File `Ex3Grammar1.jacc` defines a grammar that contains a “dangling else” construction.

```
%terminals:
';';
'(';
')';
if;
then;
else;
identifier;

%productions:
Goal -> Statement;
Statement -> Expression ';';
Statement -> if '(' Expression ')' then Statement;
Statement -> if '(' Expression ')' then Statement else Statement;
Expression -> identifier;
```

When you run this file through the parser generator, it produces the following output file. This output is located in file `Ex3Grammar1.out`.

```
Reading grammar specification ...
```

```
Generating LALR(1) configuration finite state machine ...

LALR(1) configuration finite state machine:

  State 15:
    Statement -> if ( Expression ) then Statement _
         {%%EOF, else}
    Statement -> if ( Expression ) then Statement _ else Statement
         {%%EOF, else}

    Conflict (shift/reduce 2): else
    Reduce 2: %%EOF
    Unwind: Reduce 2

12 symbols.
6 productions.
18 LALR(1) machine states.

error: 1 unresolved conflicts.

***** Error generating parser table. *****

There were 1 error and 0 warnings.
```

Notice the line that begins with the word “Conflict”. It says that the parser has encountered a shift-reduce conflict in this state. Specifically, it says that when the next input symbol is the terminal symbol `'else'`, the parser can either shift the `'else'` symbol, or reduce production number 2. Productions are numbered consecutively beginning with zero, so in this case production number 2 is the *if-then* rule.

The productions shown in the output provide further information about the cause of the conflict. The underscore character denotes the current parsing position. The symbols in curly braces are *lookahead symbols*; they indicate which terminal symbols could legally appear as the next input symbol after the end of the corresponding production. So this state represents the situation where:

1. The parser has seen the input “`if ( Expression ) then Statement`”, and
2. The parser could be parsing either the *if-then* rule or the *if-then-else* rule, in a context where it is legal for the next input symbol after the end of the rule to be the either the `'else'` symbol or the end-of-file symbol.

Sure enough, in this situation there is no way for the parser to know what to do when the next input symbol is `'else'`. So it reports a conflict.

5.5 Scanning Java Source Code

Directory `invisible\jacc\ex4` contains a compiler that reads a Java source code file. Then, it prints out a list of all the identifiers used in the code, along with the number of times that each identifier appears. It supports Java’s Unicode escape sequences.

It turns out that you don’t need a parser to do this job. The scanner alone is powerful enough to do it. So this example uses just the scanner, not the parser.

This example illustrates the following techniques:

- Using the scanner alone, without the parser.
- Writing token definitions to recognize all Java keywords and operators.

- Writing token definitions to recognize all Java numeric literals, including decimal integers, hexadecimal integers, octal integers, and floating-point formats.
- Writing token definitions to recognize Java white space and comments.
- Writing token definitions to recognize all Java character and string literals, including all legal escape sequences.
- Writing token definitions to recognize Java identifiers.
- Using the prescanner class `PrescannerJavaSource`.

5.6 Advanced Example: A Java Class Summarizer

Directory `invisible\jacc\ex5` contains a compiler that reads a Java source code file. Then, it prints out a summary of the file. The summary includes the signatures of the classes, interfaces, methods, constructors, and fields defined in the Java code. The resulting class summary is similar to the class summaries you can find in this manual and other Java API documents.

This is a major example. It is larger and more complicated than the other examples. We have included it to illustrate some of the techniques used to construct real-world compilers.

This example contains an LALR(1) parser for the full Java programming language, including Java 1.1 language extensions. The grammar specification file is based on the LALR(1) grammar given in reference [3], the Java Language Specification.

This example illustrates the following advanced techniques:

- Working with a grammar for a real-world programming language. This example uses the full grammar of the Java 1.1 language.
- Using the analysis-synthesis technique of compiler design. This example first builds up a data structure representing the entire contents of a top-level class, and then walks the data structure to produce the output.
- Collecting a sequence of items into a collection object on the parser's stack. This shows how to handle sequences whose length is unpredictable and possibility unlimited. For instance:
    - A sequence of modifiers (such as `public`, `static`, `final`, etc.) is collected into a set of flag bits, which is stored on the parser's stack. As each modifier is encountered, the appropriate flag bit is set.
    - A sequence of names (as in `invisible.jacc.gen.GenGUI`) is collected into a `String`, which is stored on the parser's stack. As each name is encountered, it is concatenated onto the `String`.
    - A sequence of variable declarations is collected into a `Vector`, which is stored on the parser's stack. As each new variable declaration is encountered, the variable is added to the `Vector`.
- Using polymorphism to represent language constructs of different types. In this example, we define one class to hold the signature of a method or constructor, a second class to hold the signature of a field, and a third class to hold the signature of a class or interface. All three of these inherit from a common abstract superclass; so the abstract superclass can represent a signature of any type.
- Using the parser's stack to construct a complicated data structure, working from the bottom up. This example begins with the lowest-level items in Java code: individual identifiers and keywords. It gradually combines these into larger and larger structures, until finally it has constructed a tree-like data structure that represents an entire top-level class. The parser's stack is used to hold the pieces of the data structure as they are being assembled and combined.

The following sections describe some of this in more detail.

5.6.1 Special Requirements for Class Summaries

First, we'll mention a few special requirements for our class summaries.

The most important special requirement is that we must handle nested classes and interfaces. In Java 1.1, class and interface definitions can be nested inside other class and interface definitions. These definitions can be nested to any depth. Therefore, we must keep track of nesting, and indent everything by the appropriate amount, according to how deeply it is nested.

Another special requirement has to do with array dimensions. For example, in Java code the following three field declarations are equivalent:

```
int[][] x;
int[] x[];
int x[][];
```

We accept any of these forms, but we convert them all into the first form, so that in the summary, array brackets always appear immediately after the type. Likewise, we convert any of the following three equivalent method declarations into the first form:

```
int[][] f(int y);
int[] f(int y)[];
int f(int y)[][];
```

Also, field declarations can define several fields in one statement. For example, the first declaration below is equivalent to the other three:

```
int x, y, z[];

int x;
int y;
int[] z;
```

When we encounter a statement that defines several fields, we split it up into separate declarations. So, in the summary each field is defined in a separate statement. That is, we always output the second form shown above.

5.6.2 Signatures

To follow this example, you need to understand the concept of signatures. The term *signature* refers to the following:

- The *signature of a field* is a set of modifiers (`public`, `static`, `final`, etc.), the type of the field, and the name of the field.
- The *signature of a method* is a set of modifiers, the return type of the method, the name of the method, a list containing the types and names of the method's formal parameters, and a list of the exceptions that the method can throw.
- The *signature of a constructor* is a set of modifiers, the name of the constructor, a list containing the types and names of the constructor's formal parameters, and a list of the exceptions that the constructor can throw.
- The *signature of a class* is a set of modifiers, the name of the class, the name of the superclass that the class extends, a list of the interfaces that the class implements, and a list containing the signatures of all the class members. The term "class member" includes all the fields, methods, constructors, nested classes, and nested interfaces located in the class.
- The *signature of an interface* is a set of modifiers, the name of the interface, a list of the superinterfaces that the interface extends, and a list containing the signatures of all the interface members. The term "interface member" includes all the fields, methods, nested classes, and nested interfaces located in the interface.

As you see, the signature of a class or interface can be very complicated. That's because the signature of a class or interface includes a list of the signatures of all its members. And some of the members can be nested classes or interfaces, each with its own members.

The function of our class summarizer is to read a Java source file, extract all the signatures, and print them out.

5.6.3 Design Overview

This compiler follows the *analysis-synthesis* design. Analysis-synthesis is very commonly used in production compilers.

An analysis-synthesis compiler works in two stages. First, it reads the input file and builds up a data structure in memory that represents the entire contents of the file. This is called the "analysis" stage. Then, it walks the data structure to produce the compiler's output. This is called the "synthesis" stage.

In this compiler, we read the source code for each top-level class or interface and produce a data structure that represents the complete signature of the class or interface. The signature of a class or interface includes the class or interface header, plus the signatures of all its fields, methods, constructors, nested classes, and nested interfaces. Of course, if there are nested classes or interfaces, their signatures include their own fields, methods, constructors, and further nested classes and interfaces.

Because of the possibility of nesting, our data structure is a tree. Each leaf in the tree contains the signature of a field, method, or constructor. Each internal node in the tree contains the signature of a class or interface. The "children" of a class or interface are all its fields, methods, constructors, and nested classes or interfaces. Since classes and interfaces can be nested to any depth, our tree can have any depth. The root node of the tree represents the top-level class.

The following diagram shows the classes we use to build the tree:

```
(PSig)
  |
  +----- (PSigContainer)
  |             |
  |             +---------- PSigClass
  |
  +----- PSigMethod
  |
  +----- PSigField
```

Parentheses indicate abstract classes, and lines indicate subclasses.

Abstract class `PSig` represents any node on the tree. Abstract class `PSigContainer` (a subclass of `PSig`) represents an internal node on the tree, that is, a node that can have children.

Concrete class `PSigClass` (a subclass of `PSigContainer`) represents the signature of a class or interface.

Concrete class `PSigMethod` (a subclass of `PSig`) represents the signature of a method or constructor. Concrete class `PSigField` (a subclass of `PSig`) represents the signature of a field.

After building up this tree, we produce the summary by walking the tree and printing out the signatures contained in the tree's nodes.

To construct the tree, we build it from the bottom up, using the parser's stack to hold the various parts of the tree under construction. To see how this is done, consider a few productions from the grammar specification file:

```
ClassDeclaration ->
      ModifiersOpt class identifier SuperOpt InterfacesOpt ClassBody;
```

```
ClassBody -> '{' ClassBodyDeclarationsOpt '}';

ClassBodyDeclarationsOpt -> ClassBodyDeclarations;
ClassBodyDeclarationsOpt -> ;

ClassBodyDeclarations -> ClassBodyDeclaration;
ClassBodyDeclarations -> ClassBodyDeclarations ClassBodyDeclaration;

ClassBodyDeclaration -> ClassMemberDeclaration;
ClassBodyDeclaration -> StaticInitializer;
ClassBodyDeclaration -> ConstructorDeclaration;

ClassMemberDeclaration -> FieldDeclaration;
ClassMemberDeclaration -> MemberDeclaration;
ClassMemberDeclaration -> ClassDeclaration;
ClassMemberDeclaration -> InterfaceDeclaration;
```

Because Invisible Jacc is an LR parser, the production for `ClassDeclaration` won't be reduced until everything on its right hand side is reduced. That means the entire class body is parsed before `ClassDeclaration` is reduced.

As we parse the class body, we discover the signatures for all the class members. When we reduce `FieldDeclaration`, we put a `PSigField` object on the parser's stack. When we reduce `MemberDeclaration` or `ConstructorDeclaration`, we put a `PSigMethod` object on the parser's stack. When we reduce `ClassDeclaration` or `InterfaceDeclaration`, we put a `PSigClass` object on the parser's stack.

By the time we reduce `ClassBodyDeclaration`, we will have a `PSig` object on the parser's stack; it could be any of the concrete types that inherit from `PSig`.

When we reduce `ClassBodyDeclarations`, we put a `Vector` of `PSig` objects on the parser's stack. In the first production, we create a new `Vector` and add the `PSig` object from `ClassBodyDeclaration` to it. In the second production, we take the existing `Vector` from `ClassBodyDeclarations`, and add the `PSig` object from `ClassBodyDeclaration`.

By the time we reduce `ClassBody`, the parser's stack will contain a `Vector` of `PSig` objects, which holds all the signatures in the class body.

Finally, when we reduce `ClassDeclaration` we can obtain the `Vector` of `PSig` objects from the parser's stack. We then use this `Vector` to construct a `PSigClass` object, which we leave on the parser's stack. The `Vector` contains all of the class's child signatures.

5.6.4 Running the Class Summarizer

Class `invisible.jacc.ex5.Ex5Main` provides a simple command-line interface for the class summarizer. You can run `invisible.jacc.ex5.Ex5Main` with the following command line:

```
[option] java-file [java-file]...
```

The following options are supported:

`-1`	Summarize `public` classes, interfaces, methods, constructors, and fields.
`-2`	Summarize `public` and `protected` classes, interfaces, methods, constructors, and fields.
`-3`	Summarize non-`private` classes, interfaces, methods, constructors, and fields.
`-4`	Summarize all classes, interfaces, methods, constructors, and fields.

If you omit the option, it defaults to `-2`, which selects `public` and `protected` declarations.

The `java-file` is the name of a Java source file. You can include more than one Java source file on the command line, in which case the program creates summaries of all of them.

The summary is written to the standard output, which is `System.out`.

For example, in Micosoft Windows you can use the following command at the `C:>` prompt to create a summary of the Java source file `Ex5Compiler.java`, including all declarations:

```
jview invisible.jacc.ex5.Ex5Main -4 invisible\jacc\ex5\Ex5Compiler.java
```

The result of running this command is in file `Ex5CompilerSummary.out`.

5.7 Invisible Jacc Input Syntax

File `invisible\jacc\gen\JaccGrammar.jacc` contains the Invisible Jacc input syntax, written in the form of an Invisible Jacc grammar specification file.

File `invisible\jacc\gen\JaccGrammar.out` is a verbose output listing.

6 Package invisible.jacc.gen

The package `invisible.jacc.gen` contains the classes to generate scanner and parser tables.

The classes and interfaces in this package fall into two categories.

- **User Interface:** Class `GenGUI` provides the Invisible Jacc graphical user interface. Class `GenMain` provides the Invisible Jacc command-line interface.
- **Programming Interface:** Class `GenFrontEnd` provides functions you can use to invoke the parser generator from within your own code. Interface `GenObserver` lets you receive notifications about the parser generator's progress. Class `GenGUIPanel` is an AWT component that lets you embed the Invisible Jacc graphical user interface into your own application or applet.

6.1 Class GenFrontEnd

```
public class GenFrontEnd
{
   // Constructors
   public GenFrontEnd ();

   // Methods for starting the parser generator
   public void generate (GenObserver genObserver, boolean async,
      ErrorOutput errOut, boolean verbose, InputStream inStream,
      String filename, boolean makeScan, boolean makeParse);
   public void generate (GenObserver genObserver, boolean async,
      ErrorOutput errOut, boolean verbose, String filename,
      boolean makeScan, boolean makeParse,
      boolean makeOut, boolean makeGen, boolean makeJava);

   // Methods for monitoring and interrupting the parser generator
   public synchronized void requestInterrupt ();
   public synchronized void waitUntilDone ();
   public synchronized boolean isDone ();

   // Methods for retrieving error information
   public int errorFlags ();
   public String summary ();
   public int errorCount ();
   public int warningCount ();

   // Methods for retrieving the results of table generation
   public ScannerTable scannerTable ();
   public ParserTable parserTable ();
   public String javaName ();

   // Methods for retrieving file names
   public String jaccFilename ();
   public String shortFilename ();
   public String outFilename ();
   public String scannerJavaFilename ();
   public String parserJavaFilename ();
   public String genFilename ();
}
```

Class `GenFrontEnd` is the front end for the parser generator. It can perform the following operations:

- Open the grammar specification file.
- Read and interpret the grammar specification file.
- Generate the scanner table.
- Generate the parser table.
- Produce output messages.
- Create an output file, and write output messages to the file.
- Write verbose output.

- Create a generated file, and write the scanner and parser tables to the generated file.
- Create Java source files, and write the scanner and parser tables to the Java source files.
- Run the parser generator in a separate background thread.
- Interrupt the parser generator.
- Send periodic reports on the progress of the parser generator.
- Produce a summary of the parser generator's results.

This class provides an integrated front end that lets you perform the entire process of creating scanner and parser tables, with a single function call.

Notice that there are two different versions of the `generate` method:

- The `generate` method with eight parameters reads the grammar specification from an input stream, and generates scanner and parser tables. This method is recommended if you are going to use the scanner and parser tables immediately, rather than saving them to disk.
- The `generate` method with ten parameters reads the grammar specification from a disk file, generates scanner and parser tables, and then saves the tables to disk files. This method is recommended if you are going to save the scanner and parser tables for later use.

This class contains several methods for retrieving the results of the parser generator. These "retrieval" methods are documented as being callable "after the parser generator has finished." It is safe to call these retrieval methods under any of the following conditions:

- If you call `generate` with the `async` parameter set to `false`, then you can call the retrieval methods any time after `generate` returns.
- If you provide a `GenObserver` object, then you can call the retrieval methods from within your implementation of `GenObserver.generatorEnd`, or any time after `GenObserver.generatorEnd` has been called.
- If you call `waitUntilDone`, then you can call the retrieval methods any time after `waitUntilDone` returns.
- If `isDone` returns `true`, then you can call the retrieval methods.

```
public GenFrontEnd ();
```

Create a new `GenFrontEnd` object.

```
public void generate (GenObserver genObserver, boolean async,
   ErrorOutput errOut, boolean verbose, InputStream inStream,
   String filename, boolean makeScan, boolean makeParse)
```

Invoke the parser generator, and produce scanner and parser tables.

The `genObserver` parameter is an observer that receives notifications of the parser generator's progress. This parameter can be `null` if notifications are not required.

If the `async` parameter is `true`, then the parser generator executes in a separate thread, and this function returns immediately without waiting for the parser generator finish. If the `async` parameter is `false`, then the parser generator executed in the current thread, and this function does not return until the parser generator is finished.

The `errOut` parameter is the destination for output messages. It can be `null`.

If the `verbose` parameter is `true`, the parser generator produces verbose output (which, like all output, is sent to the `errOut` object). The verbose output includes complete listings of the scanner and parser tables.

The `inStream` parameter is the input source. Typically, this input stream is connected to a disk file that contains the grammar specification. This parameter cannot be `null`.

The `filename` parameter is used in error messages, to identify the grammar specification file. It is *not* used to open the file. This parameter is allowed to be `null`.

If the `makeScan` parameter is `true`, then the scanner tables are generated.

If the `makeParse` parameter is `true`, then the parser tables are generated.

```
public void generate (GenObserver genObserver, boolean async,
   ErrorOutput errOut, boolean verbose, String filename,
   boolean makeScan, boolean makeParse,
   boolean makeOut, boolean makeGen, boolean makeJava)
```

Open the grammar specification file, invoke the parser generator, produce scanner and parser tables, and save the generated tables into disk files.

The `genObserver` parameter is an observer that receives notifications of the parser generator's progress. This parameter can be `null` if notifications are not required.

If the `async` parameter is `true`, then the parser generator executes in a separate thread, and this function returns immediately without waiting for the parser generator finish. If the `async` parameter is `false`, then the parser generator executed in the current thread, and this function does not return until the parser generator is finished.

The `errOut` parameter is the destination for output messages. It can be `null`.

If the `verbose` parameter is `true`, the parser generator produces verbose output (which, like all output, is sent to the `errOut` object and the output file). The verbose output includes complete listings of the scanner and parser tables.

The `filename` parameter is the name of the grammar specification file. It cannot be `null`. This file name is used to open the grammar specification file. However, in the interest of making this function "user friendly", the file name is modified as follows:

- If there is no extension on the `filename`, then the extension "`.jacc`" is appended automatically.
- This function reads the disk directory and attempts to determine the correct case of the file name. If a matching entry is found in the disk directory listing, then the supplied file name is replaced by the matching disk directory entry. This makes the `filename` parameter case-insensitive on most operating systems.
- On Microsoft Windows, this function also handles certain peculiarities of the Windows file system, such as the fact that `'\'` and `'/'` are both recognized by Windows as path separator characters.

If the `makeScan` parameter is `true`, then the scanner tables are generated.

If the `makeParse` parameter is `true`, then the parser tables are generated.

If the `makeOut` parameter is `true`, then an output file is created. All output messages are written to the output file, in addition to being sent to the `errOut` object. The name of the output file is created by appending "`.out`" to the base name of the grammar specification file.

If the `makeGen` parameter is `true`, then a generated file is created. The scanner and parser tables are written to the generated file, in binary format. The name of the output file is created by appending "`.gen`" to the base name of the grammar specification file.

If the `makeJava` parameter is `true`, then the scanner and parser tables are saved to disk as Java source files. The Java source files are stored in the same disk directory as the grammar specification file. The names of the Java source files are determined by the `%java` statement in the grammar specification file. The class name of the scanner table is created by appending “`ScannerTable`” to the class name in the `%java` statement. The class name of the parser table is created by appending “`ParserTable`” to the class name in the `%java` statement. The parser generator issues a warning if the class name in the `%java` statement does not match the name of the grammar specification file.

`public synchronized void` **`requestInterrupt`** `()`

Interrupts the parser generator. This function may be called by any thread.

This function sets a flag and then returns immediately. The parser generator is actually interrupted at some later time, when the flag is polled.

`public synchronized void` **`waitUntilDone`** `()`

Blocks the current thread until the parser generator is finished. This function may be called by any thread.

If there is an observer, this function does not return until after `GenObserver.generatorEnd` has been called.

`public synchronized boolean` **`isDone`** `()`

Returns `true` if the parser generator has finished. Returns `false` if the parser generator is still running. This function may be called by any thread.

If there is an observer, this function does not return `true` until after `GenObserver.generatorEnd` has been called.

`public int` **`errorFlags`** `()`

After the parser generator is finished, you can call this function to obtain the resulting error flags. Refer to interface `GenObserver` for documentation on the error flags.

The function’s return value is the same set of flags that is passed to `GenObserver.generatorEnd`.

`public String` **`summary`** `()`

After the parser generator is finished, you can call this function to obtain a one-line summary of the results. The return value is never `null`. Refer to interface `GenObserver` for further information.

The function’s return value is the same summary that is passed to `GenObserver.generatorEnd`.

`public int` **`errorCount`** `()`

After the parser generator is finished, you can call this function to obtain the number of error messages that were written to the output.

public int **warningCount** ()

After the parser generator is finished, you can call this function to obtain the number of warning messages that were written to the output.

public ScannerTable **scannerTable** ()

After the parser generator is finished, you can call this function to obtain the scanner tables, in the form of a `ScannerTable` object. The return value is `null` if no scanner tables were generated.

public ParserTable **parserTable** ()

After the parser generator is finished, you can call this function to obtain the parser tables, in the form of a `ParserTable` object. The return value is `null` if no parser tables were generated.

public String **javaName** ()

After the parser generator is finished, you can call this function to obtain the value of the `%java` statement in the grammar specification file. The value is returned as a `String`, which contains a fully-qualified Java package and class name. The return value is `null` if there is no `%java` statement in the grammar specification file.

public String **jaccFilename** ()

After the parser generator is finished, you can call this function to obtain the name of the grammar specification file. The return value can be `null` if the name is unspecified or unknown. The return value is the same file name that is passed to the function `GenObserver.generatorBegin`.

If the parser generator is invoked via the `generate` function with eight parameters, then `jaccFilename` returns the value of the `filename` parameter of the `generate` function.

If the parser generator is invoked via the `generate` function with ten parameters, then `jaccFilename` returns the file name that was actually used to open the grammar specification file. This may differ from the `filename` parameter of the `generate` function, for example by appending the “`.jacc`” extension and by correcting the case.

public String **shortFilename** ()

After the parser generator is finished, you can call this function to obtain a shortened form of the name of the grammar specification file. The return value can be `null` if the name is unspecified or unknown. The return value is the same shortened file name that is passed to the function `GenObserver.generatorBegin`.

If the parser generator is invoked via the `generate` function with eight parameters, then `shortFilename` returns the value of the `filename` parameter of the `generate` function.

If the parser generator is invoked via the `generate` function with ten parameters, then `shortFilename` returns the last component of the file name that was actually used to open the grammar specification file. This may differ from the last component of the `filename` parameter of the `generate` function, for example by appending the “`.jacc`” extension and by correcting the case.

`public String` **`outFilename`** `()`

After the parser generator is finished, you can call this function to obtain the name of the output file. The return value is `null` if no output file was created.

`public String` **`scannerJavaFilename`** `()`

After the parser generator is finished, you can call this function to obtain the name of the Java source file that contains the scanner tables. The return value is `null` if no scanner tables were generated, or if no Java source files were created.

`public String` **`parserJavaFilename`** `()`

After the parser generator is finished, you can call this function to obtain the name of the Java source file that contains the parser tables. The return value is `null` if no parser tables were generated, or if no Java source files were created.

`public String` **`genFilename`** `()`

After the parser generator is finished, you can call this function to obtain the name of the generated file. The return value is `null` if no generated file was created.

6.2 Class GenGUI

```
public class GenGUI
{
   // Methods
   public static void main (String[] args);
}
```

Class `GenGUI` implements the parser generator graphical user interface. Since its only method is `static`, this class should not be instantiated.

```
public static void main (String[] args)
```

Create a window on the screen and display the Invisible Jacc graphical user interface.

The argument is an array of strings that specify the command-line parameters. `GenGUI` only accepts one command-line parameter: a number which is the desired font size, in points. If not specified, the font size defaults to 14 points.

The graphical user interface is documented in chapter 4.

6.3 Class GenGUIPanel

```
public class GenGUIPanel extends java.awt.Panel
{
   // Constructors
   public GenGUIPanel (Frame parent, int fontSize);
}
```

Class `GenGUIPanel` is a Java AWT component that implements the parser generator graphical user interface. You can use this class to embed the parser generator graphical user interface into your own Java application or applet.

For example, class `GenGUI` works by simply creating an instance of `java.awt.Frame`, and then inserting a `GenGUIPanel` into it.

Caution: This class contains Java 1.1 specific code. If you use this class in your program, then your program will require a Java 1.1 compliant VM in order to run.

```
public GenGUIPanel (Frame parent, int fontSize)
```

Create an instance of GenGUIPanel.

The `parent` parameter is the frame (top-level window) which contains this instance of `GenGUIPanel`. This parameter cannot be `null`. The specified frame is used as the parent for any dialog boxes that are created.

The `fontSize` parameter is the font size, in points, to be used for this instance of `GenGUIPanel`. A value of 14 is recommended.

6.4 Class GenMain

```
public class GenMain
{
   // Methods
   public static void main (String[] args);
}
```

Class `GenMain` implements the parser generator command-line interface. Since its only method is `static`, this class should not be instantiated.

```
public static void main (String[] args)
```

Read the command line and perform the required operations. The argument is an array of strings that specify the command-line parameters.

The format of the command line is documented in chapter 4.

6.5 Interface GenObserver

```
public interface GenObserver
{
   // Methods
   public void generatorBegin (String filename, String shortFilename);
   public void generatorEnd (String summary, int errorFlags);
   public void generatorStage (String stage);
   public void generatorWork (int amount);

   // Error flag values
   public static final int efInterrupted = 0x0001;
   public static final int efAborted = 0x0002;
   public static final int efBadFilename = 0x0004;
   public static final int efJaccOpen = 0x0008;
   public static final int efJaccRead = 0x0010;
   public static final int efGrammarSpec = 0x0020;
   public static final int efParserTable = 0x0040;
   public static final int efScannerTable = 0x0080;
   public static final int efInternalError = 0x0100;
   public static final int efOutWrite = 0x0200;
   public static final int efGenWrite = 0x0400;
   public static final int efParserJavaWrite = 0x0800;
   public static final int efScannerJavaWrite = 0x1000;
   public static final int efNoJavaName = 0x2000;
}
```

Interface `GenObserver` describes an object that can receive notifications about the progress of the parser generator.

When the parser generator begins, it calls `generatorBegin`, passing the name of the grammar specification file. When the parser generator ends, it calls `generatorEnd`, passing a summary and a set of error flags. It is guaranteed that there is exactly one call to `generatorBegin` and exactly one call to `generatorEnd`.

In between the beginning and the end, the parser generator proceeds through a series of stages. At the beginning of each stage, the parser generator calls `generatorStage`, passing a description of the stage. If a stage is lengthy, the parser generator calls `generatorWork` repeatedly during the stage, passing the amount of work done since the start of the stage.

For example, the graphical user interface uses `GenObserver` to manage the progress window. The progress window is created during the call to `generatorBegin`, and disposed of during the call to `generatorEnd`. Each call to `generatorStatus` updates the status line in the progress window, and each call to `generatorWork` updates the numerical counter in the progress window.

Refer to class `GenFrontEnd` for information on how to invoke the parser generator.

```
public void generatorBegin (String filename, String shortFilename)
```

Signal the beginning of parser generation.

The `filename` parameter specifies the name of the grammar specification file. It can be `null`.

The `shortFilename` parameter is a possibly-shortened form of the name of the grammar specification file. For example, if the `filename` parameter is a complete path, then the `shortFilename` parameter may be the final component of the path. However, it is possible that `shortFilename` is the same as the `filename` parameter. This parameter can be `null`.

Refer to class `GenFrontEnd` for additional information about how filenames are handled.

`public void` **`generatorEnd`** `(String summary, int errorFlags)`

Signal the end of parser generation.

The `summary` parameter is a one-line summary of the results. It must be non-`null`.

The `errorFlags` parameter is a set of error flags. If `errorFlags` is zero, then the parser generation is completely successful. Otherwise, bits are set in `errorFlags` to indicate which errors occurred. The possible error flag bits are defined in this interface.

If the `efAborted` bit is set in `errorFlags`, it means that there was an error that made it impossible to execute the parser generator (for example, if the grammar specification file could not be opened). In this case, the `summary` parameter contains an error message describing the error.

For example, the graphical user interface uses `efAborted` as a hint for how to display the results. If `efAborted` is zero, the GUI displays the results in the output panel, with the summary appearing in a status line. If `efAborted` is one, the GUI displays the summary in a pop-up dialog box.

`public void` **`generatorStage`** `(String stage)`

Signal the start of a new stage.

The `stage` parameter is a one-line description of the new stage. It must be non-`null`.

`public void` **`generatorWork`** `(int amount)`

Signal that some work has been performed in the current stage.

If a stage is lengthy, the parser generator calls this function repeatedly during the stage. The `amount` parameter is `1` for the first call to `generatorWork` during a given stage, and is incremented by `1` on each succeeding call during the stage. Thus, the `amount` parameter is an indication of how much work has been done during the current stage.

The units of work are not specified; they do not directly correspond to anything in the grammar specification. However, the number of calls to `generatorWork` is fixed for a given grammar specification and set of generation options; the number of calls does not vary from run-to-run.

`public static final int` **`efInterrupted`** `= 0x0001`

This error flag bit is set if the parser generator was interrupted. (The parser generator can be interrupted by calling `GenFrontEnd.requestInterrupt`.)

`public static final int` **`efAborted`** `= 0x0002`

This error flag bit is set if the parser generator could not be executed at all because of an error. For example, this bit is set if the grammar specification file cannot be opened.

This bit has special significance to the `generatorEnd` function. Refer to `generatorEnd` for details.

`public static final int` **`efBadFilename`** `= 0x0004`

This error flag bit is set if the grammar specification file name is missing or invalid.

`public static final int` **`efJaccOpen`** `= 0x0008`

This error flag bit is set if the grammar specification file cannot be opened. For example, this bit is set if the file does not exist, or if system security does not allow access to the file.

`public static final int` **`efJaccRead`** `= 0x0010`

This error flag bit is set if an I/O error occurs while reading the grammar specification file.

`public static final int` **`efGrammarSpec`** `= 0x0020`

This error flag bit is set if there is a syntax error in the grammar specification file.

`public static final int` **`efParserTable`** `= 0x0040`

This error flag bit is set if an error is detected while generating the parser tables. For example, this bit is set if the grammar contains unresolved parsing conflicts.

`public static final int` **`efScannerTable`** `= 0x0080`

This error flag bit is set if an error is detected while generating the scanner tables.

`public static final int` **`efInternalError`** `= 0x0100`

This error flag bit is set if Invisible Jacc detects an internal error.

`public static final int` **`efOutWrite`** `= 0x0200`

This error flag bit is set if the output file cannot be written. For example, this bit is set if an I/O error occurs while creating or writing the output file, or if system security does not allow the file to be created.

`public static final int` **`efGenWrite`** `= 0x0400`

This error flag bit is set if the generated file cannot be written. For example, this bit is set if an I/O error occurs while creating or writing the generated file, or if system security does not allow the file to be created.

`public static final int` **`efParserJavaWrite`** `= 0x0800`

This error flag bit is set if the Java source file for the parser tables cannot be written. For example, this bit is set if an I/O error occurs while creating or writing the Java source file, or if system security does not allow the file to be created.

`public static final int` **`efScannerJavaWrite`** `= 0x1000`

This error flag bit is set if the Java source file for the scanner tables cannot be written. For example, this bit is set if an I/O error occurs while creating or writing the Java source file, or if system security does not allow the file to be created.

`public static final int` **`efNoJavaName`** `= 0x2000`

This error flag bit is set if Java source files could not be created because there is no `%java` statement in the grammar specification.

7 Package invisible.jacc.parse

The package `invisible.jacc.parse` contains the classes and interfaces which are used to construct a compiler.

This package is designed with the idea that a compiler works in four stages. First, a *prescanner* reads the source file into a buffer, and performs any preliminary transformations on the characters of the source file. You can think of the prescanner as a “filter” which can modify the stream of characters that passes through it. For example, a prescanner that reads Java code would recognize Unicode escape sequences and convert them into the corresponding character values. A prescanner that reads Fortran code would remove all space characters outside of literals. In many cases, the prescanner performs no transformations at all. The prescanner’s output is a stream of characters.

Second, a *scanner* reads a stream of characters from the prescanner, and groups them into *tokens*. Every time the scanner recognizes a token, it calls a user-supplied routine called a *token factory*. The token factory can perform whatever operations are necessary for that particular token. The scanner’s output is a stream of tokens.

Third, a *preprocessor* reads a stream of tokens from the scanner, and performs any required transformations on the token stream. You can think of the preprocessor as a “filter” which can modify the stream of tokens that passes through it. For example, the preprocessor can recognize “include” directives, which cause another source file to be included within the current source file. The preprocessor can also perform macro expansion. The preprocessor’s output is a stream of tokens.

Finally, a *parser* reads a stream of tokens from the preprocessor, and structures them according to the language grammar. Every time the parser recognizes a production, it calls a user-supplied routine called a *nonterminal factory*. The nonterminal factory can perform whatever actions are associated with that particular production. The parser does not have any output of it own; it is executed for the purpose of calling the nonterminal factories.

The classes and interfaces in this package fall into several categories.

- **Prescanner:** The interfaces `Prescanner`, `PrescannerByte`, and `PrescannerChar` define the functions that are performed by a prescanner. The class `PrescannerByteStream` is a prescanner that processes a stream of bytes and makes no modifications to the input. The class `PrescannerCharReader` is a prescanner that processes a stream of characters and makes no modifications to the input. The class `PrescannerJavaSource`, and the interface `PrescannerJavaSourceClient`, form a prescanner that processes Java source code and converts Unicode escape sequences.

- **Scanner:** The class `Scanner` is the Invisible Jacc scanner. The class `ScannerTable` holds the set of tables that are used to control the operation of the scanner. The interface `ScannerClient` is used by the scanner to makes calls back to the client. The abstract class `TokenFactory` defines the functions that are performed by a token factory.

- **Preprocessor:** The interface `Preprocessor` defines the functions that are performed by a preprocessor. The class `PreprocessorInclude` is a simple preprocessor that can recognize “include” directives.

- **Parser:** The class `Parser` is the Invisible Jacc parser. The class `ParserTable` holds the set of tables that are used to control the parser. The interface `ParserClient` is used by the parser to make calls back to the client. The abstract class `NonterminalFactory` defines the functions that are performed by a nonterminal factory.

- **Tokens:** The class `Token` defines the contents of a token. The interface `TokenStream` defines the idea of a stream of tokens.

- **Error Reporting:** The abstract class `ErrorOutput` defines a common set of functions that are used by the compiler to report errors. Class `ErrorOutputStream` is a simple implementation that sends each error message to an output stream. Class `ErrorOutputWriter` is a simple implementation that sends each error message to an output writer. Class `ErrorOutputMulticaster` is an implementation that broadcasts error messages to multiple receivers.

- **Exceptions:** Classes `SyntaxException`, `InternalCompilerException`, and `InterruptedCompilerException` are exceptions that may be thrown to indicate a condition that makes it impossible to continue a compilation.
- **Framework:** The abstract class `CompilerModel` is a simple framework that combines the various elements listed above. You can inherit from this class to create a simple compiler.
- **Version Information:** The class `ProductInfo` contains information about Invisible Jacc.

To construct a compiler, you must combine a prescanner class, a scanner class, a preprocessor class, and a parser class. The four classes are chained together, with each class sending its output to the next.

In Invisible Jacc, the scanner and parser classes are standardized. You never need to write your own scanner or parser. Instead, you use the grammar specification file, along with token factories and nonterminal factories, to customize the behavior of the scanner and parser. The Invisible Jacc scanner is in class `Scanner`, and the Invisible Jacc parser is in class `Parser`; you should never need to modify or replace these classes.

In contrast, prescanner and preprocessor classes are not standardized. Invisible Jacc comes with two different prescanner classes, and one preprocessor class. The classes included with Invisible Jacc are sufficient for many cases. But, you may find it necessary to replace the prescanner or preprocessor with one that is custom-written for your language. To replace the prescanner, you must write a class that implements the `Prescanner` interface. Likewise, to replace the preprocessor, you must write a class that implements the `Preprocessor` interface.

7.1 Class CompilerModel

```
public abstract class CompilerModel implements ScannerClient, ParserClient
{
   // Fields initialized during construction
   protected ErrorOutput _errOut;
   protected ScannerTable _scannerTable;
   protected ParserTable _parserTable;

   // Fields initialized at the start of each compilation
   protected boolean _error;
   protected Scanner _scanner;
   protected Preprocessor _preprocessor;
   protected Parser _parser;

   // Constructor
   public CompilerModel ();

   // Methods
   public boolean compile (String filename);
   public boolean error ();
   public void reportError (Token token, String code, String message);
   public void reportError (String code, String message);
   public void reportWarning (Token token, String code, String message);
   public void reportWarning (String code, String message);
   public Scanner makeScanner (String filename);
   protected boolean readGenFile (String filename);
   protected boolean setDebugMode (boolean traceScanner,
      boolean traceParser);

   // Methods that implement the ScannerClient interface
   public void scannerEOF (Scanner scanner, Token token);
   public void scannerUnmatchedToken (Scanner scanner, Token token);

   // Methods that implement the ParserClient interface
   public void parserIOException (Parser parser, IOException e);
   public void parserSyntaxException (Parser parser, SyntaxException e);
   public void parserErrorRepair (Parser parser, Token errorToken,
      int[] insertions, int insertionLength,
      int[] deletions, int deletionLength);
   public void parserErrorFail (Parser parser, Token errorToken);
}
```

`CompilerModel` is an abstract class that provides most of the implementation for a simple compiler.

When you first start using Invisible Jacc, you can use `CompilerModel` to create a simple compiler with minimum programming effort. As you become more experienced and write more sophisticated compilers, you will probably stop using `CompilerModel`.

The way you use `CompilerModel` is to write a concrete subclass that extends `CompilerModel`. At a minimum, the concrete subclass needs to provide a constructor that creates the scanner and parser tables. The concrete subclass can also override methods of `CompilerModel` to customize its behavior.

See class `Ex2Compiler` in package `invisible.jacc.ex2` for an example of how to write a subclass of `CompilerModel`. For a more advanced example of how to write a subclass of `CompilerModel`, see class `Ex5Compiler` in package `invisible.jacc.ex5`.

What a Concrete Subclass Must Do

At a minimum, a concrete subclass must include a constructor that creates scanner and parser tables. The constructor must set the variables `_scannerTable` and `_parserTable` to point to these tables.

What a Concrete Subclass May Optionally Do

Any method in `CompilerModel` may be overridden to customize its behavior. However, there are three methods that are most likely to be overridden.

The `compile` method may be overridden to provide additional initialization at the start of a parse, and additional processing after the end of a parse. It may also be overridden to install a different preprocessor, or to implement a different method for finding the source file.

The `makeScanner` method may be overridden to install a different prescanner, or to implement a search path for source files.

The `scannerEOF` method may be overridden to perform processing at the end of each source file, for example to detect run-on comments.

Invoking the Compiler

To invoke the compiler, first create an instance of the concrete subclass, and then call the `compile` method.

`protected ErrorOutput` **`_errOut`**`;`

The `ErrorOutput` object that is used as the destination for error messages. The `CompilerModel` constructor sets this to send error messages to `System.out`. A concrete subclass can change this.

`protected ScannerTable` **`_scannerTable`**`;`

The compiler's scanner table. The `CompilerModel` constructor sets this to `null`. The constructor of the concrete subclass must build the scanner table and set this variable.

`protected ParserTable` **`_parserTable`**`;`

The compiler's parser table. The `CompilerModel` constructor sets this to `null`. The constructor of the concrete subclass must build the parser table and set this variable.

`protected boolean` **`_error`**`;`

This flag is `true` if an error occurred during the compilation.

The `compile` method initializes this variable to `false` at the start of each compilation. If the concrete subclass overrides `compile`, then the concrete subclass is responsible for initializing this variable.

`protected Scanner **_scanner**;`

The scanner used for the current compilation.

The `compile` method initializes this variable at the start of each compilation. If the concrete subclass overrides `compile`, then the concrete subclass is responsible for initializing this variable.

`protected Preprocessor **_preprocessor**;`

The preprocessor used for the current compilation.

The `compile` method initializes this variable at the start of each compilation. If the concrete subclass overrides `compile`, then the concrete subclass is responsible for initializing this variable.

`protected Parser **_parser**;`

The parser used for the current compilation.

The `compile` method initializes this variable at the start of each compilation. If the concrete subclass overrides `compile`, then the concrete subclass is responsible for initializing this variable.

`public **CompilerModel** ()`

Create a compiler object.

This constructor sets `_errOut` to point at the standard output, and sets `_scannerTable` and `_parserTable` to `null`. Here is the code for this constructor:

```
public CompilerModel ()
{
   super ();
   _errOut = new ErrorOutputStream (System.out, null);
   _scannerTable = null;
   _parserTable = null;
   return;
}
```

The concrete subclass must construct scanner and parser tables, and must set `_scannerTable` and `_parserTable` to point at the constructed tables.

Optionally, the concrete subclass can change `_errOut` to point at a different `ErrorOutput` object.

`public boolean **compile** (String filename)`

Compile a source file.

The return value is `true` if there was an error.

This implementation first calls `makeScanner` to create a scanner for the given filename. Then it creates a parser object and a preprocessor object. (The preprocessor object is of class `PreprocessorInclude`.) Finally, it calls the parser to parse the source file. Here is the code for this method:

```
   public boolean compile (String filename)
   {
      _error = false;
      _scanner = makeScanner (filename);
      if (_scanner == null)
      {
         reportError (null,
            "Unable to open source file '" + filename + "'.");
         return _error;
      }
      _parser = new Parser (this, _parserTable, null);
      _preprocessor = new PreprocessorInclude (_scanner);
      _parser.parse (_preprocessor);
      _scanner = null;
      _preprocessor = null;
      _parser = null;
      return _error;
   }
```

Many concrete subclasses need to override this method. A common reason for overriding this method is to perform additional initialization before the parse begins, and additional calculations after the parse is over.

`public boolean` **`error`** `()`

The return value is `true` if an error occurred during the compilation.

The default implementation returns the current value of the `_error` field. A concrete subclass may override this method to provide a different behavior.

`public void` **`reportError`** `(Token token, String code, String message)`

Issue an error message at the specified position. The fields `token.file`, `token.line`, and `token.column` provide the file name, line number, and column number where the error occurred. The `code` argument is an error code, and the `message` argument is an error message. Either `code` or `message`, or both, may be null.

The default implementation sets the `_error` flag to true, and then writes an error message to the `_errOut` object. This is the code for the default implementation:

```
   public void reportError (Token token, String code, String message)
   {
      _error = true;
      _errOut.reportError (ErrorOutput.typeError, null, token.file,
         token.line, token.column, code, message );
      return;
   }
```

A concrete subclass may override this method to provide a different behavior.

`public void` **`reportError`** `(String code, String message)`

Issue an error message with no position specified. The `code` argument is an error code, and the `message` argument is an error message. Either `code` or `message`, or both, may be null.

The default implementation sets the `_error` flag to true, and then writes an error message to the `_errOut` object. This is the code for the default implementation:

```
public void reportError (String code, String message)
{
   _error = true;
   _errOut.reportError (ErrorOutput.typeError, null, null,
      ErrorOutput.noPosition, ErrorOutput.noPosition, code, message );
   return;
}
```

A concrete subclass may override this method to provide a different behavior.

`public void` **`reportWarning`** `(Token token, String code, String message)`

Issue a warning message at the specified position. The fields `token.file`, `token.line`, and `token.column` provide the file name, line number, and column number where the error occurred. The `code` argument is an error code, and the `message` argument is an error message. Either `code` or `message`, or both, may be null.

The default implementation writes a warning message to the `_errOut` object. This is the code for the default implementation:

```
public void reportWarning (Token token, String code, String message)
{
   _errOut.reportError (ErrorOutput.typeWarning, null, token.file,
      token.line, token.column, code, message );
   return;
}
```

A concrete subclass may override this method to provide a different behavior.

`public void` **`reportWarning`** `(String code, String message)`

Issue a warning message with no position specified. The `code` argument is an error code, and the `message` argument is an error message. Either `code` or `message`, or both, may be null.

The default implementation writes a warning message to the `_errOut` object. This is the code for the default implementation:

```
public void reportWarning (String code, String message)
{
   _errOut.reportError (ErrorOutput.typeWarning, null, null,
      ErrorOutput.noPosition, ErrorOutput.noPosition, code, message );
   return;
}
```

A concrete subclass may override this method to provide a different behavior.

`public Scanner` **`makeScanner`** `(String filename)`

Given a filename, this function creates a `Scanner` object for scanning the file. The return value is `null` if the file could not be opened.

The `compile` method calls `makeScanner` to create the scanner for the original source file. A token factory that recognizes an “include” directive can use this function to create a scanner for the include file; the scanner can then be used as the value of an insert-stream escape token.

The default implementation first opens a `FileInputStream` to read the file. Then, it creates a prescanner of class `PrescannerByteStream`. This assumes that the source file contains ASCII text, which is fed directly to the scanner with no modifications. Finally, it calls `Scanner.makeScanner` to create the `Scanner` object. This is the code for the default implementation:

```
public Scanner makeScanner (String filename)
{
   try
   {
      InputStream stream = new FileInputStream (filename);
      Prescanner scannerSource = new PrescannerByteStream (stream);
      Scanner scanner = Scanner.makeScanner (
         this, scannerSource, _scannerTable, filename, 1, 1, 4000, null );
      return scanner;
   }
   catch (IOException e)
   {
      return null;
   }
}
```

A concrete subclass can override this method to provide a different behavior. Possible reasons for overriding this method include (a) using a different prescanner, or (b) implementing a search path.

`protected boolean` **`readGenFile`** `(String filename)`

Given a filename, this function reads a generated file and creates the `_scannerTable` and `_parserTable` objects. The file must be a “`.gen`” file as created by Invisible Jacc.

The return value is `false` if the function is successful, or `true` if an error occurred. Note that this function catches all I/O exceptions internally, and returns `true` if an I/O exception occurs.

If the function is successful, then on return `_scannerTable` and `_parserTable` contain the scanner and parser tables, respectively. Here is the code for this method:

```
protected boolean readGenFile (String filename)
{
   try
   {
      DataInputStream stream =
         new DataInputStream (new FileInputStream (filename));
      _scannerTable = new ScannerTable ();
      _scannerTable.readFromStream (stream);
      _parserTable = new ParserTable ();
      _parserTable.readFromStream (stream);
      stream.close ();
      return false;
   }
   catch (IOException e)
   {
      return true;
   }
}
```

This function is provided as a convenience for concrete subclasses. If the scanner and parser tables are stored in a generated file, then the constructor of a concrete subclass may call `readGenFile` to read in the tables.

```
protected boolean setDebugMode (boolean traceScanner,
   boolean traceParser);
```

Activate debugging features.

This checks the scanner and parser tables for internal consistency. It also checks all the token names, nonterminal names, and link names passed to the `linkFactory` functions, to ensure that they match names that appear in the grammar specification. If any errors are found, messages are sent to the `_errOut` object.

If `traceScanner` is true, then scanner tracing is enabled. This sends a message to the `_errOut` object on every call to a token factory.

If `traceParser` is true, then parser tracing is enabled. This sends a message to the `_errOut` object on every call to a nonterminal factory.

This function should be used only while you are debugging your code. The call to `setDebugMode` should be placed in the constructor of your concrete subclass, after you set up the scanner and parser tables, make all your calls to `linkFactory`, and set up `_errOut` to the destination for debug output.

The release version of your code should not call `setDebugMode` at all.

The return value is `true` if an error was detected in the scanner and/or parser tables.

Here is the code for this method:

```
protected boolean setDebugMode (boolean traceScanner, boolean traceParser)
{
   if (traceScanner)
   {
      _scannerTable.setTrace (_errOut);
   }
   if (traceParser)
   {
      _parserTable.setTrace (_errOut);
   }
   String sinv = _scannerTable.checkInvariant();
   if (sinv != null)
   {
      _errOut.reportError (ErrorOutput.typeError, null, null,
         ErrorOutput.noPosition, ErrorOutput.noPosition, null,
         "Invalid scanner table: " + sinv );
      _errOut.flush();
   }
   String pinv = _parserTable.checkInvariant();
   if (pinv != null)
   {
      _errOut.reportError (ErrorOutput.typeError, null, null,
         ErrorOutput.noPosition, ErrorOutput.noPosition, null,
         "Invalid parser table: " + pinv );
      _errOut.flush();
   }
   return (sinv != null) || (pinv != null);
}
```

This method is provided as a convenience for concrete subclasses. The constructor of a concrete subclass can activate debugging features with a single call to `setDebugMode`, instead of making several calls to methods of `ScannerTable` and `ParserTable`.

`public void` **`scannerEOF`** `(Scanner scanner, Token token)`

The scanner calls this routine when it reaches end-of-file.

The default implementation does nothing. This is the code for the default implementation:

```
public void scannerEOF (Scanner scanner, Token token)
{
   return;
}
```

A concrete subclass can override this method to provide a different behavior. A common reason for overriding this method is to check for run-on comments.

Implements the `scannerEOF` method of `ScannerClient`.

`public void` **`scannerUnmatchedToken`** `(Scanner scanner, Token token)`

The scanner calls this routine when it cannot match a token.

The default implementation calls `reportError` to write an error message. This is the code for the default implementation:

```
public void scannerUnmatchedToken (Scanner scanner, Token token)
{
   reportError (token, null,
      "Illegal character or unrecognized token in input." );
   return;
}
```

A concrete subclass can override this method to provide a different behavior.

Implements the `scannerUnmatchedToken` method of `ScannerClient`.

`public void` **`parserIOException`** `(Parser parser, IOException e)`

The parser calls this routine when an I/O exception occurs.

The default implementation calls `reportError` to write an error message. This is the code for the default implementation:

```
public void parserIOException (Parser parser, IOException e)
{
   reportError ("I/O Exception", e.toString() + ".");
   return;
}
```

A concrete subclass can override this method to provide a different behavior.

Implements the `parserIOException` method of `ParserClient`.

```
public void **parserSyntaxException** (Parser parser, SyntaxException e)
```

The parser calls this routine when a syntax exception occurs.

The default implementation calls `reportError` to write an error message. This is the code for the default implementation:

```
   public void parserSyntaxException (Parser parser, SyntaxException e)
   {
      if (e instanceof InterruptedCompilerException)
      {
         reportError (null, "Interrupted.");
      }
      else
      {
         reportError ("Syntax Exception", e.toString() + ".");
      }
      return;
   }
```

A concrete subclass can override this method to provide a different behavior.

Implements the `parserSyntaxException` method of `ParserClient`.

```
public void **parserErrorRepair** (Parser parser, Token errorToken,
   int[] insertions, int insertionLength,
   int[] deletions, int deletionLength)
```

The parser calls this routine when it repairs a syntax error.

The default implementation calls `reportError` to write an error message. This is the code for the default implementation:

```
   public void parserErrorRepair (Parser parser, Token errorToken,
      int[] insertions, int insertionLength,
      int[] deletions, int deletionLength)
   {
      if ((insertionLength + deletionLength) <= 6)
      {
         for (int i = 0; i < insertionLength; ++i)
         {
            reportError (errorToken, null,
               "Expected '" + parser.symbolName(insertions[i]) + "'." );
         }
         for (int i = 0; i < deletionLength; ++i)
         {
            reportError (errorToken, null,
               "Unexpected '" + parser.symbolName(deletions[i]) + "'." );
         }
      }
      else
      {
         reportError (errorToken, null, "Syntax error.");
      }
      return;
   }
```

A concrete subclass can override this method to provide a different behavior.

Implements the `parserErrorRepair` method of `ParserClient`.

`public void` **`parserErrorFail`** `(Parser parser, Token errorToken)`

The parser calls this routine when it is unable to repair a syntax error.

The default implementation calls `reportError` to write an error message. This is the code for the default implementation:

```
public void parserErrorFail (Parser parser, Token errorToken)
{
   reportError (errorToken, null, "Syntax error - unable to continue.");
   return;
}
```

A concrete subclass can override this method to provide a different behavior.

Implements the `parserErrorRepair` method of `ParserClient`.

7.2 Class ErrorOutput

```
public abstract class ErrorOutput
{
   // Message type codes
   public static final int typeInformational = 0;
   public static final int typeWarning = 1;
   public static final int typeError = 2;

   // Constant that defines an unknown or unspecified line or column.
   public static final int noPosition = 0x80000000;

   // Constructor
   protected ErrorOutput (String program);

   // Methods for reporting errors
   public final void reportError (int type, String module, String file,
      int line, int column, String code, String message);
   public final int errorCount ();
   public final int warningCount ();
   public final void resetCounters ();
   public void flush ();

   // Methods for subclasses
   protected final String program ();
   protected String formatMessage (int type, String module, String file,
      int line, int column, String code, String message);
   protected abstract void handleError (int type, String module, String file,
      int line, int column, String code, String message);
}
```

`ErrorOutput` is an abstract class that represents a destination for error messages.

`ErrorOutput` provides base functionality which includes converting an error message into a text string, and maintaining counts of the number of error and warning messages.

The public interface of `ErrorOutput` consists primarily of a method called `reportError` which is used to report error messages, warning messages, and informational messages. It also includes routines to read and reset the message counters.

The protected interface of `ErrorOutput` consists primarily of an abstract method called `handleError` which a concrete subclass overrides to define how a message is handled. It also provides a method called `formatMessage` which provides a default method for converting an error message into a text string.

See `ErrorOutputStream` for an example of a concrete subclass of `ErrorOutput`. The subclass `ErrorOutputStream` just prints each error message on an output stream. A more sophisticated subclass might store all the error messages in a table, and use the table to display the source code where each error occurred.

```
public static final int typeInformational = 0;
```

The type code for informational messages.

```
public static final int typeWarning = 1;
```

The type code for warning messages.

```
public static final int typeError = 2;
```

The type code for error messages.

```
public static final int noPosition = 0x80000000;
```

An integer value that may be used to indicate an unknown line number, or an unknown column number.

```
protected ErrorOutput (String program)
```

Create an error output object, and initialize the internal error and warning counters to zero.

The `program` argument is a string that is prepended to the start of each message; it can be `null`. The intended use of the `program` argument is identify the program that issued the messages, in situations where messages from several different programs may be intermixed on the screen. Normally messages are not intermixed, and so normally you would set `program` to `null`.

```
public final void reportError (int type, String module, String file,
   int line, int column, String code, String message)
```

Report an error message, a warning message, or an informational message.

The `type` argument is an integer identifying the type of message. It can be `typeInformational`, `typeWarning`, or `typeError`.

The `module` argument is a string identifying the program module that issued the message. This can be `null`.

The `file` argument is a string identifying the source file. This can be `null`.

The `line` argument is an integer identifying the line within the specified source file where the error occurred. A value of `noPosition` indicates that the line number is unknown or unspecified.

The `column` argument is an integer identifying the column within the specified line where error occurred. A value of `noPosition` indicates that the column number is unknown or unspecified. It is expected that column numbers (and sometimes line numbers as well) are approximations.

The `code` argument is a string containing an error code. This can be `null`.

The `message` argument is a string containing the text of the message. This can be `null`.

This method does the following: If the `type` argument is invalid, an exception is thrown. If the `type` argument is `typeError`, the internal error counter is incremented. If the `type` argument is `typeWarning`, the internal warning counter is incremented. Then, the `handleError` method is called.

`public final int` **`errorCount`** `()`

Returns the number of error messages issued since the `ErrorOutput` object was created, or since the last call to `resetCounters`.

`public final int` **`warningCount`** `()`

Returns the number of warning messages issued since the `ErrorOutput` object was created, or since the last call to `resetCounters`.

`public final void` **`resetCounters`** `()`

Resets the internal error and warning counters to zero.

`public void` **`flush`** `()`

Flushes any buffered data to the output sink. The default implementation of `flush` does nothing.

`protected final String` **`program`** `()`

Returns the program name that was originally passed to the constructor.

```
protected String formatMessage (int type, String module, String file,
   int line, int column, String code, String message)
```

Create a string that contains the complete message, and return the resulting `String` object.

See the method `reportError` for an explanation of the parameters.

The default implementation creates a string by concatenating the following elements:

1. The program name specified in the constructor, if it is not `null` and not empty.
2. The `module` argument, if it is not `null` and not empty.
3. The `file` argument, if it is not `null` and not empty.
4. The `line` argument, if it is not `noPosition`.
5. The `column` argument, if it is not `noPosition`.
6. The string “`error`” if the `type` argument is `typeError`, or “`warning`” if the `type` argument is `typeWarning`.
7. The `code` argument, if it is not `null` and not empty.
8. The `message` argument, if it is not `null` and not empty.

The default implementation also inserts appropriate punctuation between the elements. The message format produced by this method is intended to be similar to the error messages typically produced by commercial compilers.

A subclass may override this method to provide a different message format.

```
protected abstract void handleError (int type, String module, String file,
   int line, int column, String code, String message)
```

Handle an error message.

This is an abstract method. A subclass must override this method to provide an implementation for handling error messages.

The method `reportError` calls this method after validating the `type` parameter and incrementing the appropriate counter.

See the method `reportError` for an explanation of the parameters.

7.3 Class ErrorOutputMulticaster

```
public class ErrorOutputMulticaster extends ErrorOutput
{
   // Constructor
   public ErrorOutputMulticaster ();

   // Methods
   public void add (ErrorOutput target);
   public void remove (ErrorOutput target);
   protected void handleError (int type, String module, String file,
      int line, int column, String code, String message);
   public void flush ();
}
```

`ErrorOutputMulticaster` is an implementation of the `ErrorOutput` abstract class that broadcasts each message to a set of targets.

An `ErrorOutputMulticaster` contains a set of `ErrorOutput` objects. Each message is forwarded to all the `ErrorOutput` objects in the target set.

`public` **`ErrorOutputMulticaster`** `()`

Create a new `ErrorOutputMulticaster` object. Initially, the set of targets is empty.

`public void` **`add`** `(ErrorOutput target)`

Add a receiver to the set of targets.

If the `target` argument is non-`null`, then the `ErrorOutput` object it refers to is added to the set of targets.

If the `target` argument is `null`, or if the `target` argument refers to an `ErrorOutput` object that is already in the set of targets, then this method does nothing.

`public void` **`remove`** `(ErrorOutput target)`

Remove a receiver from the set of targets.

If the `target` argument is non-`null`, then the `ErrorOutput` object it refers to is removed from the set of targets.

If the `target` argument is `null`, or if the `target` argument refers to an `ErrorOutput` object that is not in the set of targets, then this method does nothing.

```
protected void handleError (int type, String module, String file,
   int line, int column, String code, String message)
```

Handle an error message.

This method forwards the message to all the receivers in the target set. It calls `ErrorOutput.reportError` on each object in the target set.

Refer to the `reportError` method in class `ErrorOutput` for an explanation of the parameters.

`public void` **`flush`** `()`

Flushes any buffered data to the output sink. This method calls `ErrorOutput.flush` on each object in the target set.

7.4 Class ErrorOutputStream

```
public class ErrorOutputStream extends ErrorOutput
{
   // Constructors
   public ErrorOutputStream (PrintStream stream, String program);
   public ErrorOutputStream (PrintStream stream, String program,
      String separator);

   // Methods
   protected void handleError (int type, String module, String file,
      int line, int column, String code, String message);
   public void flush ();
}
```

`ErrorOutputStream` is a simple implementation of the `ErrorOutput` abstract class that immediately writes each message to an output stream.

```
public ErrorOutputStream (PrintStream stream, String program)
```

Create an error output object. Each message is written to the specified `stream`.

The `program` argument is a string that is prepended to the start of each message; it can be `null`. The intended use of the `program` argument is identify the program that issued the messages, in situations where messages from several different programs may be intermixed on the screen. Normally messages are not intermixed, and so normally you would set `program` to `null`.

Each message is terminated with the default line separator. In Java 1.0, the default line separator is the newline character `"\n"`. In Java 1.1, the default line separator is the value of the system property `line.separator`, which is `"\r\n"` on Microsoft Windows.

```
public ErrorOutputStream (PrintStream stream, String program,
   String separator)
```

Create an error output object. Each message is written to the specified `stream`.

The `program` argument is a string that is prepended to the start of each message; it can be `null`. The intended use of the `program` argument is identify the program that issued the messages, in situations where messages from several different programs may be intermixed on the screen. Normally messages are not intermixed, and so normally you would set `program` to `null`.

If the `separator` argument is non-`null`, then each message is terminated with the specified `separator`.

If the `separator` argument is `null`, then each message is terminated with the default line separator. In Java 1.0, the default line separator is the newline character `"\n"`. In Java 1.1, the default line separator is the value of the system property `line.separator`, which is `"\r\n"` on Microsoft Windows.

```
protected void handleError (int type, String module, String file,
   int line, int column, String code, String message)
```

Handle an error message.

This method calls `formatMessage` to construct a string that contains the complete message. Then it writes the string to the output stream.

If no separator was specified when this `ErrorOutputStream` was created, then `PrintStream.println` is used to write the message. If a separator was specified when this `ErrorOutputStream` was created, then `PrintStream.print` is used to write the message followed by the specified line separator.

Refer to the `reportError` method in class `ErrorOutput` for an explanation of the parameters.

```
public void flush ()
```

Flushes any buffered data to the output sink. This method calls `PrintStream.flush` on the output stream.

7.5 Class ErrorOutputWriter

```
public class ErrorOutputWriter extends ErrorOutput
{
   // Constructors
   public ErrorOutputWriter (PrintWriter writer, String program);
   public ErrorOutputWriter (PrintWriter writer, String program,
      String separator);

   // Methods
   protected void handleError (int type, String module, String file,
      int line, int column, String code, String message);
   public void flush ();
}
```

`ErrorOutputWriter` is a simple implementation of the `ErrorOutput` abstract class that immediately writes each message to an output writer.

Caution: This class contains Java 1.1 specific code. If you use this class in your program, then your program will require a Java 1.1 compliant VM in order to run.

```
public ErrorOutputWriter (PrintWriter writer, String program)
```

Create an error output object. Each message is written to the specified `writer`.

The `program` argument is a string that is prepended to the start of each message; it can be `null`. The intended use of the `program` argument is identify the program that issued the messages, in situations where messages from several different programs may be intermixed on the screen. Normally messages are not intermixed, and so normally you would set `program` to `null`.

Each message is terminated with the default line separator. The default line separator is the value of the system property `line.separator`, which is `"\r\n"` on Microsoft Windows.

```
public ErrorOutputWriter (PrintWriter writer, String program,
   String separator)
```

Create an error output object. Each message is written to the specified `writer`.

The `program` argument is a string that is prepended to the start of each message; it can be `null`. The intended use of the `program` argument is identify the program that issued the messages, in situations where messages from several different programs may be intermixed on the screen. Normally messages are not intermixed, and so normally you would set `program` to `null`.

If the `separator` argument is non-`null`, then each message is terminated with the specified `separator`.

If the `separator` argument is `null`, then each message is terminated with the default line separator. The default line separator is the value of the system property `line.separator`, which is `"\r\n"` on Microsoft Windows.

```
protected void handleError (int type, String module, String file,
   int line, int column, String code, String message)
```

Handle an error message.

This method calls `formatMessage` to construct a string that contains the complete message. Then it writes the string to the output stream.

If no separator was specified when this `ErrorOutputWriter` was created, then `PrintWriter.println` is used to write the message. If a separator was specified when this `ErrorOutputWriter` was created, then `PrintWriter.print` is used to write the message followed by the specified line separator.

Refer to the `reportError` method in class `ErrorOutput` for an explanation of the parameters.

```
public void flush ()
```

Flushes any buffered data to the output sink. This method calls `PrintWriter.flush` on the output writer.

7.6 Class InternalCompilerException

```
public class InternalCompilerException extends SyntaxException
{
   // Constructors
   public InternalCompilerException ()
   public InternalCompilerException (String s)
}
```

`InternalCompilerException` is a subclass of `SyntaxException`. It can be thrown when a compiler detects an error that is best described as being internal to the compiler.

For example, `InternalCompilerException` can be thrown when a nonterminal factory receives an invalid production parameter.

`public` **`InternalCompilerException`** `()`

Creates a new exception object with `null` as its error message string.

`public` **`InternalCompilerException`** `(String s)`

Creates a new exception object with `s` as its error message string.

7.7 Class InterruptedCompilerException

```
public class InterruptedCompilerException extends SyntaxException
{
   // Constructors
   public InterruptedCompilerException ()
   public InterruptedCompilerException (String s)
}
```

`InterruptedCompilerException` is a subclass of `SyntaxException`. It can be thrown when a compiler detects that the user is requesting an interrupt or abort.

`public` **`InterruptedCompilerException`** `()`

Creates a new exception object with `null` as its error message string.

`public` **`InterruptedCompilerException`** `(String s)`

Creates a new exception object with `s` as its error message string.

7.8 Class NonterminalFactory

```
public abstract class NonterminalFactory
{
   // Factory Method
   public abstract Object makeNonterminal (Parser parser, int param)
      throws IOException, SyntaxException;
}
```

`NonterminalFactory` is used by the parser to obtain the values of nonterminal symbols.

Every production has an associated object of class `NonterminalFactory`. Whenever the parser reduces a production, it calls the `makeNonterminal` method for the corresponding nonterminal factory. The parser makes available the production's parameter, and the portion of the parser's stack that contains the values for the right hand side of the production. The `makeNonterminal` method returns the value of the nonterminal on the left hand side of the production; this value is stored on the parser's stack.

A compiler typically defines several subclasses of `NonterminalFactory`, one subclass for each production in the grammar that requires special action. Each subclass of `NonterminalFactory` is called a *nonterminal factory*.

It is recommended that a nonterminal factory object should not store any state information. State information should be stored either in a "global" client object, or in the parser's `clientParams` variable. In Java 1.1, a nonterminal factory could be declared as an inner class nested inside the global client, which would give the nonterminal factory convenient access to the global client's variables and methods.

```
public abstract Object makeNonterminal (Parser parser, int param)
   throws IOException, SyntaxException
```

Calculate the value of a nonterminal symbol. Typically, this function also has side effects, such as updating the compiler's internal data structures.

Assume that this `NonterminalFactory` corresponds to the production `Y -> X1 ... Xn`. Whenever the parser reduces this production, it calls the `makeNonterminal` method. During this call, `parser.rhsValue(0)` through `parser.rhsValue(n-1)` return the values of `X1` through `Xn` respectively. These values come from the parser's stack. Note that it is allowed for values to be `null`.

The integer parameter associated with this production is passed in the argument `param`. If there is more than one production with `Y` on the left hand side that has the same link name, the integer parameter could be used to indicate which production is being reduced.

The function returns the value of `Y`. This value is saved on the parser's stack. Note that the return value is allowed to be `null`.

During this call, `parser.token()` returns the most recently shifted `Token` object. The nonterminal factory may examine the `file`, `line`, and `column` fields of this `Token` object. This can be used to relate semantic errors to a specific position within the source file.

Also, `parser.client()` may be used to retrieve the `ParserClient` object.

In addition, the nonterminal factory may access the `public` variable `parser.clientParams`, which it may use for any purpose.

Note 1: If error repair is enabled, then any terminal symbol on the right hand side can be the result of an error insertion. An error insertion symbol has `null` value; that is, `parser.rhsValue` returns `null` if the terminal symbol was inserted during error repair. Therefore, if error repair is enabled, then the nonterminal factory must be prepared to accept a `null` value for any terminal symbol on the right hand side.

Note 2: If error repair is enabled, and the grammar is ambiguous, then error insertions may cause productions to be reduced differently than specified by the precedence rules. Therefore, the nonterminal factory should not contain code that crashes if the precedence rules are not followed. In practice, this is unlikely to require any special code.

7.9 Class Parser

```
public class Parser
{
   // Field
   public Object clientParams;

   // Constructor
   public Parser (ParserClient client, ParserTable parserTable,
      Object params);

   // Methods for use by clients
   public boolean parse (Preprocessor source);
   public final String symbolName (int symbolNumber);

   // Methods for use by nonterminal factories
   public final Object rhsValue (int offset);
   public final Token token ();
   public final ParserClient client ();
   public final ParserTable parserTable ();
}
```

`Parser` is the Invisible Jacc parser. It is an LR(1) parser with LM error repair.

Creation

When you create a `Parser` object, you must provide the following:

- An object of type `ParserClient`. This provides callback functions that the parser can use to inform the client of significant events. These include parsing errors (whether repaired or not), I/O exceptions, and syntax exceptions.
- An object of type `ParserTable`. This contains all the tables required for the LR(1) parsing algorithm and the LM error repair algorithm. In addition, it contains pointers to the nonterminal factories that are called during the parse.
- An `Object` which is used to initialize the public variable `clientParams`. This can be `null`.

After creating a `Parser` object, the client should not alter the `ParserTable` object that was used to create it.

The `clientParams` variable may be used by the client and the nonterminal factories for any desired purpose. It is provided as a convenient place for the client to store per-parser information that needs to be accessed during the parse.

Invoking the Parser

Once the `Parser` object is created, you invoke it by calling the `parse` method. When you call the `parse` method, you must pass in an object of type `Preprocessor`. The preprocessor provides the stream of tokens that the parser is to parse.

The `parse` method automatically reads all the tokens from the preprocessor. The return value is a boolean that indicates if the parse was successful. Parsing terminates when one of the following occurs:

- The end-of-file token is shifted onto the parser’s stack. In this case, `parse` returns `false`.

- An unrepairable parser error occurs. In this case, `parse` calls `ParserClient.parserErrorFail` and then returns `true`.
- An I/O exception occurs. In this case, `parse` calls `ParserClient.parserIOException` and then returns `true`.
- A syntax exception occurs. In this case, `parse` calls `ParserClient.parserSyntaxException` and then returns `true`.

Note that `parse` returns `true` if any error occurs that prevents the entire input from being parsed. On the other hand, if errors occur but are repaired, `parse` returns `false`. Also, note that `parse` catches all checked exceptions, so the client doesn't have to do it.

After the `parse` method returns, you can call `parse` again with another preprocessor, to parse another source file using the same parser tables. It is not necessary to create a new `Parser` object for each source file.

Parser Operation

The parser operates using the LR(1) parsing algorithm. A description of the algorithm can be found in references [1] and [2].

The parser maintains an internal stack, which stores the values of terminal and nonterminal symbols. Each entry in the stack can store an object of type `Object`, or `null`.

During the parse, the parser performs a series of actions. There are only two possible actions: the parser can *shift* a terminal symbol, or it can *reduce* a production.

When the parser shifts a terminal symbol, it reads one token from the preprocessor. Each token corresponds to one terminal symbol of the grammar. The parser obtains the symbol's value from the `value` field of the `Token` object, and pushes that value onto the parser's stack. (If the input tokens are coming from a `Scanner` object, then the token value is the value supplied by the token factory.)

When the parser reduces a production, the parser calls the nonterminal factory for that production. Suppose that the production is `Y -> X1 ... Xn`. Then, the top `n` elements on the parser's stack are the values of the symbols `X1` through `Xn`. The nonterminal factory can call the method `rhsValue` to retrieve these values. The nonterminal factory can also call the `token` method to get the current position in the source file. Refer to the description of class `NonterminalFactory` for additional documentation.

After the nonterminal factory returns, the values of the right hand side are popped off the parser's stack, and the value returned by the nonterminal factory is pushed onto the parser's stack. Thus, the nonterminal factory returns the value of the production's left hand side. As a result, the size of the parser's stack is reduced by `n-1` elements. (If the production has an empty right hand side, so that `n` equals zero, then the parser's stack actually increases in size by one element.)

In a successful parse, the last production reduced is a production with the goal symbol on the left hand side. This means that the entire input has been reduced to the goal symbol.

Error Repair

A *parsing error* occurs when the parser encounters an input symbol that cannot be shifted onto the parser's stack. When this happens, the parser uses a modified version of the LM error repair algorithm to attempt to repair the error and continue parsing, provided that error repair is enabled in the parser tables. A description of the LM algorithm can be found in reference [2]. (We have made two alterations in the algorithm, one to ensure that it cannot enter an infinite loop, and one to make it work with ambiguous grammars.)

An *error repair* consists of deleting zero or more terminal symbols from the front of the input, and then inserting zero or more terminal symbols onto the front of the input. All repairs are *validated*; that is, the parser checks to make sure that after the repair is done, several additional input symbols can be shifted.

When an error is repaired, the parser calls `ParserClient.parserErrorRepair` to inform the client.

When a terminal symbol is inserted during error repair, the value is set to `null`, and the position information is set to the same position as the symbol that originally caused the error. Therefore, if error repair is enabled, then every nonterminal factory must be prepared to accept a `null` value for any terminal symbol on the right hand side.

There is one special consideration when LM error repair is used with an ambiguous grammar. When the inserted terminal symbols are processed, the resulting reductions always respect the grammar, but may not respect the precedence rules that were used to resolve parsing conflicts. It appears that this rarely occurs in practice. Nonetheless, nonterminal factories should be designed so that they at least don't crash if productions are reduced differently than specified in the precedence rules. (In practice, this is unlikely to require any special code in the nonterminal factories.)

```
public Object clientParams;
```

The client parameters. Client code and nonterminal factories may use this `public` variable for any desired purpose.

```
public Parser (ParserClient client, ParserTable parserTable,
   Object params)
```

Create a new parser. The `client` argument is the object that the parser will call whenever an error occurs. The `parserTable` argument contains the parsing tables for the grammar. The `params` argument is the initial value of the `clientParams` field; it can be `null`.

Refer to the documentation following the class summary for additional information.

```
public boolean parse (Preprocessor source)
```

Parse the input. The parser reads input from the `source` argument, and attempts to parse it according to the grammar specification.

Refer to the documentation following the class summary for a description of how the parser parses the input.

```
public final String symbolName (int symbolNumber)
```

Obtain the name of a symbol, given its number.

This method may be called by a nonterminal factory or by a client. This is typically used for error reporting. For example, the method `ParserClient.parserErrorRepair` can call `symbolName` to convert the supplied error repair information into human-readable form.

```
public final Object rhsValue (int offset)
```

Read a value from the right hand side of a production.

A nonterminal factory may call this function to retrieve a value from the parser's stack.

Assume that a nonterminal factory is being called to reduce the production `Y -> X1 ... Xn`. Then, `rhsValue(0)` through `rhsValue(n-1)` return the values of `X1` through `Xn` respectively.

Negative offsets are permitted to access values of the right hand sides of containing productions that are not yet reduced. Doing this requires detailed knowledge of the properties of the grammar, and is not recommended.

`public final Token` **`token`** `()`

Obtain the most recently processed token.

A nonterminal factory may call this function to get the current position in the source file. This is typically used for error reporting.

The nonterminal factory may not modify the returned `Token` object. The contents of the `Token` object remain valid only during the call to the nonterminal factory (after that, the fields in the `Token` object may be overwritten).

`public final ParserClient` **`client`** `()`

Returns the `ParserClient` object that was used to create this parser.

`public final ParserTable` **`parserTable`** `()`

Returns the `ParserTable` object that was used to create this parser.

7.10 Interface ParserClient

```
public interface ParserClient
{
   // Methods
   public void parserIOException (Parser parser, IOException e);
   public void parserSyntaxException (Parser parser, SyntaxException e);
   public void parserErrorRepair (Parser parser, Token errorToken,
      int[] insertions, int insertionLength,
      int[] deletions, int deletionLength);
   public void parserErrorFail (Parser parser, Token errorToken);
}
```

`ParserClient` is an interface that represents the client of a `Parser` object.

The parser uses this interface to call back to its client to inform the client of errors.

```
public void parserIOException (Parser parser, IOException e)
```

Report an I/O exception.

The parser calls this function to inform the client that an I/O exception has occurred. This function is called immediately before `Parser.parse` returns. Typically, this function would generate an error message.

The `IOException` object is the exception object that was thrown. There is no reliable way for this function to determine the position in the source file where the exception occurred. Therefore, the code that throws the exception should include an error message in the exception object, or otherwise inform the user of where the error occurred.

See the documentation for method `parserIOException` in class `CompilerModel` for an example of how to implement this function.

```
public void parserSyntaxException (Parser parser, SyntaxException e)
```

Report a syntax exception.

The parser calls this function to inform the client that a syntax exception has occurred. This function is called immediately before `Parser.parse` returns. Typically, this function would generate an error message.

The `SyntaxException` object is the exception object that was thrown. There is no reliable way for this function to determine the position in the source file where the exception occurred. Therefore, the code that throws the exception should include an error message in the exception object, or otherwise inform the user of where the error occurred.

See the documentation for method `parserSyntaxException` in class `CompilerModel` for an example of how to implement this function.

```
public void parserErrorRepair (Parser parser, Token errorToken,
   int[] insertions, int insertionLength,
   int[] deletions, int deletionLength);
```

Report an error repair.

The parser calls this function to inform the client that a parser error was detected and repaired. An error repair consists of first deleting zero or more terminal symbols from the front of the input, then inserting zero or more terminal symbols onto the front of the input.

The number of symbols inserted is `insertionLength`. The symbols inserted are in `insertions[0]` through `insertions[insertionLength-1].` Note that `insertions[0]` is the last symbol pushed back onto the input, so it will be the first symbol processed when parsing resumes.

The number of symbols deleted is `deletionLength`. The symbols deleted are in `deletions[0]` through `deletions[deletionLength-1]`. Note that `deletions[0]` appeared first in the original input.

Note that `insertionLength` does not equal `insertions.length`. Likewise, `deletionLength` does not equal `deletions.length`. Therefore, you must rely on the two integer arguments, not the lengths of the arrays, to determine how many symbols were inserted and deleted.

It is guaranteed that at least one of `insertionLength` and `deletionLength` will be nonzero.

In the `insertions` and `deletions` arrays, terminal symbols are identified by their symbol numbers. You can call `parser.symbolName` to get the name of any symbol, given its number.

The input token that caused the error is `errorToken`. The fields `errorToken.file`, `errorToken.line`, and `errorToken.column` contain the source file name, line number, and column number where the error occurred. Note that if `deletionLength` is nonzero then `deletions[0]` equals `errorToken.number`.

See the documentation for method `parserErrorRepair` in class `CompilerModel` for an example of how to implement this function.

`public void` **`parserErrorFail`** `(Parser parser, Token errorToken)`

Report an error that the parser was not able to repair.

The parser calls this function to inform the client that a parser error occurred, and the parser was not able to repair it. This may happen either because the parser table has error repairs disabled, or because the parser tried all possible repairs and was unable to validate any of them.

The input token that caused the error is `errorToken`. The fields `errorToken.file`, `errorToken.line`, and `errorToken.column` contain the source file name, line number, and column number where the error occurred.

See the documentation for method `parserErrorFail` in class `CompilerModel` for an example of how to implement this function.

7.11 Class ParserTable

```
public class ParserTable implements Cloneable
{
   //Constructor
   public ParserTable ();

   // Methods
   public int lookupSymbol (String name);
   public void linkFactory (String nonterminalName, String linkName,
      NonterminalFactory factory);
   public void setTrace (ErrorOutput traceOut);
   public void writeToStream (DataOutput stream) throws IOException;
   public void readFromStream (DataInput stream) throws IOException;
   public String checkInvariant ();
   public boolean writeToJavaSource (PrintStream stream, String packageName,
      String className, boolean useRLE);
   public Object clone ();
}
```

`ParserTable` holds the tables required for `Parser`.

Generating Tables

The class `GenFrontEnd` contains functions to generate parser tables. Using `GenFrontEnd`, you supply a specification of the grammar, and then `GenFrontEnd` creates the resulting `ParserTable` object.

Once you have a `ParserTable` object, you will probably want to save it for future use. There are two ways to do this.

- You can use `writeToStream` to write the `ParserTable` to an output stream. This can be used to save the tables into a file.
- Alternatively, you can use `writeToJavaSource` to create a subclass of `ParserTable` that contains all the tables as literals. This produces a Java source file that you can compile.

The class `GenGUI` provides a graphical user interface that invokes the parser generator, and then writes the parser tables either to a file, or to Java source code. The class `GenMain` provides a simple command-line interface that invokes the parser generator, and then writes the parser tables either to a file, or to Java source code.

Using the Tables

To use the tables, you first have to retrieve them from where they are stored.

- If the tables are stored in a file, you can create a new `ParserTable` object and then call `readFromStream` to load the tables from an input stream. Refer to method `readGenFile` in class `CompilerModel` for an example of how to do this.
- If the tables are stored as literals in a subclass, you can create a new instance of the subclass, and cast the result to type `ParserTable`. When you create the subclass object, the tables are automatically loaded from the literals.

After retrieving the tables, you need to link in the nonterminal factories. To do this, call `linkFactory` once for each combination of left hand side and link name that has a nonterminal factory. `ParserTable` automatically constructs an internal table containing all the nonterminal factories. (If you don't supply factories for all productions, `ParserTable` automatically supplies default factories. If a production's right hand side is nonempty, the default factory returns the value of the first symbol on the right hand side; otherwise, the default factory returns `null`.)

If you want to enable tracing, call `setTrace` and specify the destination for tracing output. If you enable tracing, `ParserTable` inserts code to write a message every time a production is reduced. This is useful for debugging.

Finally, pass the `ParserTable` object to `Parser`.

Other Functions

`ParserTable` defines other functions that can be used for special purposes.

You can call `clone` to create a copy of a `ParserTable` object. This is a shallow copy which contains the same tables. This is useful if you want to create two different `ParserTable` objects, with the same parsing tables but different sets of nonterminal factories. A multi-pass compiler might need to do this, supplying a different set of nonterminal factories for each pass. Note that for this to work, you must call `clone` before making any calls to `linkFactory`.

You can call `lookupSymbol` to get the number of any symbol in the grammar, given its name. You can use this to link token factories to their corresponding token numbers. Normally, this linkage is done automatically by the parser generator; each token factory is passed a parameter which is normally the token number. In some cases you may have to perform the linkage yourself, for example, if a given token factory can return two or more different types of tokens. Also, if the tokens are coming from a source other than `Scanner`, you should use this function to get the token numbers to use.

You can call `checkInvariant` to make `ParserTable` check its tables for internal consistency. This can be used after retrieving the tables from a file, if you want to check that the tables were read properly. It can also be used after linking factories to verify that every factory actually corresponds to at least one production.

```
public ParserTable ()
```

Create a new `ParserTable` object. The object contains no parsing tables, and it has no linked nonterminal factories. Tracing is disabled.

```
public int lookupSymbol (String name)
```

Given the name of a symbol, this method returns the corresponding symbol number. If the name is invalid, this method returns `-1`.

```
public void linkFactory (String nonterminalName, String linkName,
   NonterminalFactory factory)
```

Links a nonterminal factory into the parser tables.

The `nonterminalName` argument is the name of the nonterminal symbol on the production's left hand side.

The `linkName` argument is the production's link name. If the production does not have a link name, this should be an empty string (not `null`).

The `factory` argument is the nonterminal factory for the production. Note that if there is more than one production with the same left hand side and link name, this factory will be used for all of them.

If no factory is provided for a given production, then a default factory is used. If the production's right hand side is nonempty, the default factory returns the value of the first symbol on the right hand side. If the right hand side is empty, the default factory returns `null`.

Note that there is no way to "unlink" a nonterminal factory once it has been linked.

`public void` **`setTrace`** `(ErrorOutput traceOut)`

Sets the destination for tracing output. If the argument is `null`, then tracing is disabled.

If this function is never called, then tracing is disabled.

When tracing is enabled, the production's left hand side, link name, parameter, and position are written out every time a production is reduced. This lets you view the operation of the parser. Caution: This produces a large amount of output!

`public void` **`writeToStream`** `(DataOutput stream) throws IOException`

Write the parser tables to the `stream`, in binary format.

The binary format is fairly complicated. If you need to know the format, look at the source code for the class `ParserTable`.

`public void` **`readFromStream`** `(DataInput stream) throws IOException`

Read the parser tables from the `stream`, in binary format.

This function performs only a cursory check of data integrity. If you want to fully check data integrity, call `checkInvariant` after calling `readFromStream`.

`public String` **`checkInvariant`** `()`

This method checks the object to make sure that its contents are internally consistent.

If the contents are consistent, the method returns `null`.

If any inconsistency is found, the method returns a `String` describing the problem.

`public boolean` **`writeToJavaSource`** `(PrintStream stream, String packageName, String className, boolean useRLE)`

This method writes a Java source file for a subclass of `ParserTable`. The subclass contains all the generated parser tables, written as literal constants.

The constructor for the subclass automatically sets all the parser tables to the literal constants. This saves the time that would otherwise be required to read the tables in from a file.

The Java source file is written to the specified `stream`. The `packageName` argument is the name of the subclass's package, and the `className` argument is the name of the subclass itself.

If `useRLE` is `true`, the tables are compressed with RLE encoding. The tables are automatically expanded the first time an object of this class is constructed. This reduces the size of the class file, at the expense of the time required for RLE decoding. (Note, however, that the Java VM Specification implies a limit of approximately 8000 array initializer elements per class. Thus, RLE encoding may be mandatory for large tables.)

The return value is `stream.checkError`, which is `true` if there was an I/O error.

`public Object` **`clone`** `()`

Creates a copy of the tables. This is a shallow copy. The clone object shares its contained tables with the original.

7.12 Interface Preprocessor

```
public interface Preprocessor extends TokenStream
{
   // Methods
   public Token nextToken () throws IOException, SyntaxException;
   public void pushBackToken (Token token);
   public Token peekAheadToken (int distance)
      throws IOException, SyntaxException;
   public void close ();
}
```

A `Preprocessor` object represents a stream of tokens, with push-back and peek-ahead capability. Reading a token automatically advances the internal stream pointer to the next token. Pushing back a token inserts it into the stream at the current position, so it can later be read. Peeking ahead examines upcoming tokens without removing them from the stream.

Each token is packaged in a `Token` object.

End-of-stream is indicated by returning token number `0`, which is indicated symbolically as `Token.EOF`.

Typically, a preprocessor reads input from a `TokenStream` object, and filters the token stream in some way. For example, class `PreprocessorInclude` is an implementation of `Preprocessor` that recognizes “include” directives. A more sophisticated preprocessor might perform macro expansion.

Note that a `Preprocessor` can also be used as a `TokenStream`. This allows multiple `Preprocessor` objects to be chained together.

`public Token nextToken () throws IOException, SyntaxException`

Gets the next token in the stream, and removes it from the stream.

If any tokens were pushed back, this method retrieves the token most recently pushed back, and removes it from the push-back queue.

When the end of the stream is reached, this function should return with `token.number` equal to `Token.EOF`. The end-of-stream token may be retrieved an arbitrary number of times.

In order to avoid the overhead of creating a `Token` object for each token, this function may return the same `Token` object repeatedly, and simply update the fields of the `Token` object for each call. Therefore, the caller must treat the returned `Token` object as read-only, and may use the returned `Token` object only until the next call to `nextToken` or `peekAheadToken`.

`public void pushBackToken (Token token)`

Pushes back the specified token, so that the next call to `nextToken` retrieves the pushed-back token.

An implementation of `Preprocessor` is required to support an unlimited number of push-backs. A typical implementation would copy the `Token` object for each pushed-back token, and store the copies in a queue.

This method is not allowed to retain a reference to the supplied `Token` object. It is permitted to pass in the `Token` object that was returned from the last call to `nextToken` or `peekAheadToken`.

Implementations may assume that calls to this function are relatively rare, perhaps occurring only during error recovery. In particular, a parser should not use this function to push back “lookahead” tokens.

```
public Token peekAheadToken (int distance)
   throws IOException, SyntaxException;
```

Retrieves a token from the stream, without removing it from the stream.

The `distance` argument specifies how far ahead in the stream to look. If `distance` is `0`, the function returns the token that will be returned by the next call to `nextToken`. The effect of this function is the same as making `distance+1` calls to `nextToken`, saving the last token, and then making `distance+1` calls to `pushBackToken`.

If `distance` is past the end of the stream, this function should return with `token.number` equal to `Token.EOF`. If `distance` is negative, this function should throw `IllegalArgumentException`.

An implementation of `Preprocessor` is required to support peeking ahead an unlimited number of tokens. A typical implementation would create a `Token` object for each peek-ahead token, and store the upcoming tokens in a queue.

In order to avoid the overhead of creating a `Token` object for each call, this function may return a `Token` object that is also stored internally. Therefore, the caller must treat the returned `Token` object as read-only, and may use the returned `Token` object only until the next call to `nextToken` or `peekAheadToken`.

Implementations may assume that calls to this function are relatively rare, perhaps occurring only during error recovery. In particular, a parser should not use this function to retrieve “lookahead” tokens.

```
public void close ()
```

Closes the token stream.

Typically, this function closes the `TokenStream` from which tokens are being obtained.

Note that this function is not allowed to throw `IOException`.

7.13 Class PreprocessorInclude

```
public class PreprocessorInclude implements Preprocessor
{
   // Constructor
   public PreprocessorInclude (TokenStream stream);

   // Method
   public void pushBackStream (TokenStream stream);

   // Methods that implement the Preprocessor interface
   public Token nextToken () throws IOException, SyntaxException;
   public void pushBackToken (Token token);
   public Token peekAheadToken (int distance)
      throws IOException, SyntaxException;
   public void close ();
}
```

`PreprocessorInclude` is an implementation of the `Preprocessor` interface that supports "include" directives. `PreprocessorInclude` reads input from a `TokenStream`, adds the push-back and peek-ahead functions required by `Preprocessor`, and also adds the "include" functionality.

`PreprocessorInclude` supports stacking of token streams. At any time, a new token stream can be pushed on the stack. `PreprocessorInclude` then takes succeeding tokens from the new stream. When the new stream returns end-of-file, `PreprocessorInclude` closes the new stream, pops the stack, and continues reading tokens from the original stream. This allows for processing of "include" files, transparently to the client.

Stacking is integrated with the push-back and peek-ahead functions to be sure they interact correctly.

`PreprocessorInclude` recognizes two special token types:

- The *end-of-file token*, denoted by `Token.EOF`. When an end-of-file token is received from any stream other than the original stream, the token is deleted and the stream stack is popped. An end-of-file token obtained from the original stream is returned to the client unchanged.
- The *insert-stream escape token*, denoted by `Token.escapeInsertStream`. When this token is received, the token's `value` object is cast to a `TokenStream`, and the stream is pushed onto the stream stack. The effect is that the entire contents of the stream is inserted at the point where the single token would have been inserted.

All other token types are passed through to the client unchanged.

Tokens that are pushed back are not given any special treatment, even if they are end-of-file or insert-stream tokens. Tokens that are pushed back are always returned to the client unchanged.

`public` **`PreprocessorInclude`** `(TokenStream stream)`

Create a new preprocessor that reads input from the specified `stream`.

`public void` **`pushBackStream`** `(TokenStream stream)`

Push the specified stream onto the internal stream stack.

The new stream is inserted before any buffered tokens. This means that `nextToken` will return all the tokens in the new stream before it returns any tokens that have been buffered by push-back or peek-ahead operations. In other words, the new stream is inserted at the client's current position.

Warning: A token factory should never call this function. A token factory that recognizes an "include" directive should assemble an insert-stream escape token.

```
public Token nextToken () throws IOException, SyntaxException
```

Implements the `nextToken` method of `Preprocessor`.

```
public void pushBackToken (Token token)
```

Implements the `pushBackToken` method of `Preprocessor`.

```
public Token peekAheadToken (int distance)
   throws IOException, SyntaxException;
```

Implements the `peekAheadToken` method of `Preprocessor`.

```
public void close ()
```

Implements the `close` method of `Preprocessor`. This method closes all stacked `TokenStream` objects, as well as the original `TokenStream` object.

7.14 Interface Prescanner

```
public interface **Prescanner**
{
   // Method
   public void close () throws IOException;
}
```

`Prescanner` is a common superinterface for `PrescannerByte` and `PrescannerChar`. `Prescanner` abstracts away from the data type of the prescanner, and allows all prescanners to be handled polymorphically.

Note: You should never write a class that implements `Prescanner` directly. When you write a prescanner class, you should implement either `PrescannerByte` or `PrescannerChar`.

```
public void close () throws IOException
```

Closes the prescanner.

7.15 Interface PrescannerByte

```
public interface PrescannerByte extends Prescanner
{
   // Methods
   public int read (byte[] dstArray, int dstOffset, int dstLength,
      Token token) throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

`PrescannerByte` represents a prescanner that stores each character in a `byte`. The prescanner is an input source for the scanner. Typically, the prescanner reads input from a file, filters the character stream in some way, and then delivers the resulting characters.

An implementation of `PrescannerByte` is required to deliver characters into a buffer array provided by the scanner. The buffer array has type `byte[]`.

Note: A prescanner that stores each character in a variable of type `byte` should implement `PrescannerByte`. A prescanner that stores each character in a variable of type `char` should implement `PrescannerChar` instead. `PrescannerBye` is appropriate for ASCII input sources, while `PrescannerChar` is appropriate for Unicode input sources.

For an example of how to implement the `PrescannerByte` interface, refer to the source code for class `PrescannerByteStream`.

```
public int read (byte[] dstArray, int dstOffset, int dstLength,
   Token token) throws IOException, SyntaxException
```

Read `byte`s from the source. The `byte`s are read into `dstArray`, beginning at element `dstOffset` (that is, the first `byte` read goes into array element `dstArray[dstOffset]`). A maximum of `dstLength` `byte`s may be read. The value of `dstLength` must be a positive integer.

If the return value is positive, it is the number of `byte`s actually read. The prescanner is not required to fill the buffer on each call, even if there is sufficient input data available to fill the buffer.

If the return value is zero, it means that there is no more input data.

If the return value is negative, it means that the prescanner could not deliver any data because the buffer is too small. The negative of the return value is the minimum buffer size required. The scanner must allocate a larger buffer and retry the operation. If the prescanner cannot determine how large a buffer is required, it may simply return `-1`, since the scanner always increases the buffer size by at least a factor of 3/2 following the return of any negative value.

It is recommended that a prescanner not produce error messages. But if it must produce an error message, it can use the `token` argument to identify the error's location. The current file name is in `token.file`. Estimates of the current line and column number (that is, the line and column of the first character to be read) are in `token.line` and `token.column`.

It is recommended that the line count should be maintained in the token factories. However, it is possible for the prescanner to maintain the line count. To do this, the prescanner should deliver one line of input (or partial line) on each call. Whenever a new line is delivered, the prescanner should increment `token.line` and set `token.column` to `1`.

`public void close () throws IOException`

Closes the prescanner.

7.16 Class PrescannerByteStream

```
public class PrescannerByteStream implements PrescannerByte
{
   // Constructor
   public PrescannerByteStream (InputStream stream)

   // Methods that implement the PrescannerByte interface
   public int read (byte[] dstArray, int dstOffset, int dstLength,
      Token token) throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

`PrescannerByteStream` is a prescanner that reads from an `InputStream`, and delivers the raw byte stream to the scanner, with no modifications.

`PrescannerByteStream` implements the `PrescannerByte` interface. Therefore, each input character is stored in a `byte`. This prescanner is suitable for ASCII input files.

```
public PrescannerByteStream (InputStream stream)
```

Creates a new prescanner that reads input from the specified `stream`.

```
public int read (byte[] dstArray, int dstOffset, int dstLength,
   Token token) throws IOException, SyntaxException
```

Implements the `read` method of interface `PrescannerByte`.

```
public void close () throws IOException
```

Closes the contained `InputStream`.

7.17 Interface PrescannerChar

```
public interface PrescannerChar extends Prescanner
{
   // Methods
   public int read (char[] dstArray, int dstOffset, int dstLength,
      Token token) throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

PrescannerChar represents a prescanner that stores each character in a char. The prescanner is an input source for the scanner. Typically, the prescanner reads input from a file, filters the character stream in some way, and then delivers the resulting characters.

An implementation of PrescannerChar is required to deliver characters into a buffer array provided by the scanner. The buffer array has type char[].

Note: A prescanner that stores each character in a variable of type char should implement PrescannerChar. A prescanner that stores each character in a variable of type byte should implement PrescannerByte instead. PrescannerBye is appropriate for ASCII input sources, while PrescannerChar is appropriate for Unicode input sources.

For an example of how to implement the PrescannerChar interface, refer to the source code for class PrescannerJavaSource.

Caution: If you use PrescannerChar, you need to set the character set size to 65536. The following statement in the grammar specification file sets the character set size to 65536:

```
   %options:
   %charsetsize 0x10000;
```

```
public int read (char[] dstArray, int dstOffset, int dstLength,
   Token token) throws IOException, SyntaxException
```

Read chars from the source. The chars are read into dstArray, beginning at element dstOffset (that is, the first char read goes into array element dstArray[dstOffset]). A maximum of dstLength chars may be read. The value of dstLength must be a positive integer.

If the return value is positive, it is the number of chars actually read. The prescanner is not required to fill the buffer on each call, even if there is sufficient input data available to fill the buffer.

If the return value is zero, it means that there is no more input data.

If the return value is negative, it means that the prescanner could not deliver any data because the buffer is too small. The negative of the return value is the minimum buffer size required. The scanner must allocate a larger buffer and retry the operation. If the prescanner cannot determine how large a buffer is required, it may simply return -1, since the scanner always increases the buffer size by at least a factor of 3/2 following the return of any negative value.

It is recommended that a prescanner not produce error messages. But if it must produce an error message, it can use the token argument to identify the error's location. The current file name is in token.file. Estimates of the current line and column number (that is, the line and column of the first character to be read) are in token.line and token.column.

It is recommended that the line count should be maintained in the token factories. However, it is possible for the prescanner to maintain the line count. To do this, the prescanner should deliver one line of input (or partial line) on each call. Whenever a new line is delivered, the prescanner should increment `token.line` and set `token.column` to `1`.

```
public void close () throws IOException
```

Closes the prescanner.

7.18 Class PrescannerCharReader

```
public class PrescannerCharReader implements PrescannerChar
{
   // Constructor
   public PrescannerCharReader (Reader reader)

   // Methods that implement the PrescannerChar interface
   public int read (char[] dstArray, int dstOffset, int dstLength,
      Token token) throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

`PrescannerCharReader` is a prescanner that reads from a `Reader`, and delivers the raw character stream to the scanner, with no modifications.

`PrescannerCharReader` implements the `PrescannerChar` interface. Therefore, each input character is stored in a `char`. This prescanner is suitable for whatever sort of input files are supported by the `Reader` class. The supported files vary from platform to platform.

Caution: If you use `PrescannerCharReader`, you need to set the character set size to 65536. The following statement in the grammar specification file sets the character set size to 65536:

```
%options:
%charsetsize 0x10000;
```

Caution: This class contains Java 1.1 specific code. If you use this class in your program, then your program will require a Java 1.1 compliant VM in order to run.

```
public PrescannerCharReader (Reader reader)
```

Creates a new prescanner that reads input from the specified `reader`.

```
public int read (char[] dstArray, int dstOffset, int dstLength,
   Token token) throws IOException, SyntaxException
```

Implements the `read` method of interface `PrescannerChar`.

```
public void close () throws IOException
```

Closes the contained `Reader`.

7.19 Class PrescannerJavaSource

```
public class PrescannerJavaSource implements PrescannerChar
{
   // Constructor
   public PrescannerJavaSource (PrescannerJavaSourceClient client,
      InputStream stream, int bufSize);

   // Methods that implement the PrescannerChar interface
   public int read (char[] dstArray, int dstOffset, int dstLength,
      Token token) throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

`PrescannerJavaSource` is a prescanner that reads a Java source file from an `InputStream`. The Java source file is presumed to be stored as one character per byte (that is, the Java source file is an ASCII file). `PrescannerJavaSource` converts each byte to a `char`, and also recognizes and converts Unicode escape sequences.

`PrescannerJavaSource` implements the `PrescannerChar` interface.

When you create a `PrescannerJavaSource`, you must supply both an `InputStream` and a `PrescannerJavaSourceClient`. When the prescanner encounters an invalid Unicode escape sequence, it calls the `PrescannerJavaSourceClient` to report the error.

Note: The presence of Unicode escapes `\u000A` and `\u000D` (line feed and carriage return) will throw off the scanner's line count, because the token factories consider them to be line ends, but the text editor does not consider them to be line ends. However, the Visual J++ compiler (version 1.01) also exhibits this behavior.

For an example of how to use `PrescannerJavaSource`, refer to file `Ex4Compiler.java`.

Caution: If you use `PrescannerJavaSource`, you need to set the character set size to 65536. The following statement in the grammar specification file sets the character set size to 65536:

```
   %options:
   %charsetsize 0x10000;
```

```
public PrescannerJavaSource (PrescannerJavaSourceClient client,
   InputStream stream, int bufSize)
```

Creates a new prescanner that reads input from the specified `stream`.

The prescanner uses an internal buffer to read data from the `stream`. The `bufSize` argument specifies the size of the internal buffer, in bytes. A typical value for the `bufSize` argument is `2000`.

The `client` argument is used to report invalid Unicode escape sequences. Whenever the prescanner encounters an invalid escape sequence, it calls the `client` object to report the error.

```
public int read (char[] dstArray, int dstOffset, int dstLength,
   Token token) throws IOException, SyntaxException
```

Implements the `read` method of interface `PrescannerChar`.

`public void **close** () throws IOException`

Closes the contained `InputStream`.

7.20 Interface PrescannerJavaSourceClient

```
public interface PrescannerJavaSourceClient
{
   // Method
   public void javaSourceInvalidEscape (Token token);
}
```

PrescannerJavaSourceClient is an interface that represents the client of a PrescannerJavaSource object.

For an example of how to use PrescannerJavaSourceClient, refer to file Ex4Compiler.java.

```
public void javaSourceInvalidEscape (Token token);
```

The prescanner calls this routine when it encounters an invalid Unicode escape code.

The fields token.file, token.line, and token.column contain the file name, line number, and column number where the invalid escape sequence is located.

A typical implementation of this method would print an error message.

7.21 Class ProductInfo

```
public class ProductInfo
{
   // Fields
   public static final String product = "Invisible Jacc";
   public static final String copyright =
      "Copyright 1997-1998 Invisible Software, Inc.";
   public static final String version = "1.1";
   public static final int majorVersion = 1;
   public static final int minorVersion = 1;
   public static final String author = "Invisible Software, Inc.";
   public static final String description =
      "Invisible Jacc is a Java class library for scanning and parsing.";
}
```

Class `ProductInfo` contains information about Invisible Jacc. The information includes the product name, copyright notice, version number, author, and description.

Since all the fields of `ProductInfo` are `static`, this class should not be instantiated.

7.22 Class Scanner

```
public abstract class Scanner implements TokenStream
{
   // Field
   public Object clientParams;

   // Factory method
   public static Scanner makeScanner (ScannerClient client,
      Prescanner source, ScannerTable scannerTable, String file,
      int line, int column, int bufSize, Object params);

   // Methods that implement the TokenStream interface
   public abstract Token nextToken () throws IOException, SyntaxException;
   public abstract void close () throws IOException;

   // Methods for start conditions
   public final int condition ();
   public final void setCondition (int newCondition);

   // Methods used by token factories to obtain the token string
   public final int tokenLength ();
   public final void setTokenLength (int newTokenLength);
   public final String tokenToString ();
   public final String tokenToString (int off, int len);
   public final char tokenCharAt (int off);
   public final void tokenToChars (int off, int len,
      char[] dst, int dstOff);

   // Methods used by token factories to obtain the context string
   public final int contextLength ();
   public final String contextToString ();
   public final String contextToString (int off, int len);
   public final char contextCharAt (int off);
   public final void contextToChars (int off, int len,
      char[] dst, int dstOff);

   // Methods used by token factories to obtain additional input text
   public final int textLength ();
   public final String textToString (int off, int len);
   public final char textCharAt (int off);
   public final void textToChars (int off, int len,
      char[] dst, int dstOff);
   public abstract int readData () throws IOException, SyntaxException;

   // Methods used by token factories for direct access to the input buffer
   public abstract boolean isByteText ();
   public byte[] rawByteText ();
   public char[] rawCharText ();
   public final int tokenStart ();
   public final void setTokenStart (int newTokenStart);
   public final int dataEnd ();
   public abstract String rawTextToString (int off, int len);
   public abstract char rawTextCharAt (int off);
   public abstract void rawTextToChars (int off, int len,
      char[] dst, int dstOff);

   // Method used by token factories to count lines
   public final void countLine ();

   // Other methods for use by token factories or clients
   public final ScannerClient client ();
   public final ScannerTable scannerTable ();
   public final int tokenIndex ();
}
```

`Scanner` is the Invisible Jacc scanner. Its function is to read a stream of input characters, and break it into tokens.

Creation

When you create a `Scanner` object, you must provide the following:

- An object of type `ScannerClient`. This provides callback functions that the scanner can use to inform the client of significant events. These include scanning errors and end-of-file.
- An object of type `Prescanner`. This provides the input to the scanner.
- An object of type `ScannerTable`. This contains all the tables required for the scanning algorithm. In addition, it contains pointers to the token factories that are called during the scan.
- A `String` that contains the filename. The filename is inserted into the `file` field of each Token object.
- Two integers that contain the initial line and column numbers. These are used to initialize the `line` and `column` fields of the Token objects.
- An integer that specifies the size of the scanner's internal text buffer.
- An `Object` which is used to initialize the public variable `clientParams`. This can be `null`.

After creating a `Scanner` object, the client should not alter the `ScannerTable` object that was used to create it.

The `clientParams` variable may be used by the client and the token factories for any desired purpose. It is provided as a convenient place for the client to store per-file information that needs to be accessed during the scan.

Notice that because `Scanner` is an abstract class, you can't write "`new Scanner`" to create a `Scanner`. Instead, you have to use the `makeScanner` factory method.

Invoking the Scanner

Once the `Scanner` object is created, you invoke it by calling the `nextToken` method. When you call the `nextToken` method, the scanner analyzes the input and returns a `Token` object that represents the next token in the input.

When the scanner reaches the end of the input, it calls `ScannerClient.scannerEOF`. Then, it returns an end-of-file token.

If the input does not match any token, the scanner calls `ScannerClient.scannerUnmatchedToken` to report the error. Then, the scanner skips ahead in the input to look for a valid token.

Scanner Operation

The scanner operates using a pair of deterministic finite automata. Technical details of the algorithm can be found in references [1] and [2].

The function of the scanner is to read a stream of input characters and break the input into *tokens*. Each token is a string of one or more consecutive input characters.

Each token is defined by a regular expression, which is called the *token regular expression*. Also, a token definition may optionally include a second regular expression, which is called the *right context regular expression*.

The scanner maintains a *current position* in the input. The general idea is to find the longest possible token beginning at the current position. In case of a tie, the scanner gives priority to the token definition that appears earliest in the grammar specification file.

On each call to `nextToken`, the scanner finds the next token as follows:

1. Each token definition is marked *active* or *inactive*, depending on the scanner's start condition. Inactive token definitions do not participate in the scan.
2. The scanner finds the *context string*.
   - The context string is the longest string, beginning at the current position, that matches any of the active token definitions.
   - If a token definition does not have a right context, we say that a string *matches* the token definition if it matches the token regular expression.
   - If a token definition has a right context, we say that a string *matches* the token definition if it matches the catenation of the token regular expression with the right context regular expression.
   - If no such string exists, the scanner informs its client that an error occurred, by calling the method `ScannerClient.scannerUnmatchedToken`. Then, the scanner advances the current position by one character, and goes back to step 1 to begin a new scan.
3. The scanner selects the *current token definition*.
   - The current token definition is the token definition that is matched by the context string.
   - If there is more than one token definition that is matched by the context string, then the scanner chooses the token definition that appears earliest in the grammar specification file.
4. The scanner finds the *token string*.
   - The token string is a string, beginning at the current position, that matches the token regular expression for the current token definition.
   - If the current token definition does not have a right context, then the token string is the same as the context string.
   - If the current token definition has a right context, then the scanner splits the context string into two substrings, one that matches the token regular expression and one that matches the right context regular expression. The first substring becomes the token string. If there is more than one way to split the context string, then the scanner chooses the split that produces the longest token string.
5. The scanner calls the token factory for the current token definition. The scanner makes the token string and context string available to the token factory. The token factory can assemble the token, discard the token, or reject the token.
6. If the token factory assembles the token, the scanner's current position is advanced to the end of the token string. Then, the `nextToken` method returns the resulting token, in the form of a `Token` object. The token's `number` field identifies the type of token that was recognized. The token's `value` field contains the value supplied by the token factory.

7. If the token factory discards the token, the scanner's current position is advanced to the end of the token string. Then, the scanner goes back to step 1 and begins a new scan.
8. If the token factory rejects the token, the scanner finds the next best combination of context string, token definition, and token string. (In choosing the new combination, the scanner does not re-scan the input, but rather uses the stored results of the previous scan.)
   - If possible, the scanner keeps the same context string and token definition, but uses a shorter token string. This can occur only if the current token definition has a right context, and the context string can be split in more than one way. If this is possible, the scanner chooses the next-longest token string, and goes to step 5.
   - Otherwise, if possible, the scanner keeps the same context string, but uses a different token definition. This can occur only if there is more than one token definition that is matched by the context string. If this is possible, the scanner chooses the next matching token definition in the grammar specification file, and goes to step 4.
   - Otherwise, if possible, the scanner uses a shorter context string. This can occur only if there is more than one string, beginning at the current position, that matches an active token definition. If this is possible, the scanner chooses the next-longest context string, and goes to step 3.
   - Otherwise, there is an error. The scanner reports the error to its client by calling the method `ScannerClient.scannerUnmatchedToken`. Then, the scanner advances the current position by one character, and goes to step 1 to begin a new scan.

Whenever the scanner advances its current position in the input, it also advances the `column` field of the `Token` object by the same amount. Thus, the scanner automatically keeps track of the column number as it processes tokens. It is the responsibility of the token factories to recognize line-ends and advance the line number.

When the scanner reaches the end of the file, it notifies the client by calling `ScannerClient.scannerEOF`. Then, `nextToken` returns the end-of-file token.

Technical note: If you are familiar with the Unix program Lex, you should be aware of a subtle difference between Lex and Invisible Jacc in the handling of right contexts. When Lex splits a context string into two substrings, Lex makes the first substring equal to the longest initial substring that matches the token regular expression. With Lex, there is no guarantee that the second substring actually matches the right context regular expression. In contrast, when Invisible Jacc splits a context string into two substrings, the first substring always matches the token regular expression, and the second substring always matches the right context regular expression.

```
public Object clientParams;
```

The client parameters. Client code and token factories may use this `public` variable for any desired purpose.

```
public static Scanner makeScanner (ScannerClient client,
   Prescanner source, ScannerTable scannerTable, String file,
   int line, int column, int bufSize, Object params)
```

Create a new scanner. The `client` argument is the object that the scanner will call to report errors and end-of-file.

The `source` argument is the input source from which the scanner will read characters. Although the function signature shows `source` as being of type `Prescanner`, in fact `source` must have type `PrescannerByte` or `PrescannerChar`. The `makeScanner` method automatically creates the appropriate concrete scanner class for the given type of `source`.

The `scannerTable` argument contains the scanning tables.

The `file` argument is the filename that is inserted in the `file` field of each `Token` object. Note, in particular, that this filename is *not* used to open a file. The filename can be `null`.

The `line` and `column` arguments are the initial line and column numbers. Typically, each of these equals `1`.

The `bufSize` argument is the size of the scanner's internal text buffer, in characters. The text buffer is used to buffer up the text that matches a token definition. It also allows the scanner to read a large amount of input on each call to the prescanner, which makes I/O operations more efficient. A typical buffer size is `4000` characters.

The `params` argument is the initial value of the `clientParams` field; it can be `null`.

Refer to the documentation following the class summary for additional information.

`public abstract Token` **`nextToken`** `() throws IOException, SyntaxException`

Reads the next token from the input, and returns it as a `Token` object. Refer to the documentation after the class summary for details on how this method operates.

The `number` and `value` fields of the returned `Token` object contain the token type and value, as returned by the token factory. The `file` field contains the filename specified in the `makeScanner` function. The `line` and `column` fields contain the line and column number where the token is located. (Since the column number is advanced before `nextToken` returns, the line and column number actually refer to the first character *after* the end of the token string.)

When end-of-file is reached, `nextToken` returns an end-of-file token. This is indicated by a `Token` object whose `number` field is zero. (Note that `Token.EOF` is defined to be zero.)

`public abstract void` **`close`** `() throws IOException`

Closes the contained `Prescanner` object.

`public final int` **`condition`** `()`

Returns the current start condition. The start condition is an integer, that can range from zero to one less than the number of start conditions.

Start conditions are assigned consecutive integer values, beginning with zero, in the order that they are listed in the %conditions section of the grammar specification file. You can use `ScannerTable.lookupCondition` to find the integer value of a start condition, given its name. When a scanner is first created, the start condition is initialized to zero.

`public final void` **`setCondition`** `(int newCondition)`

Sets the current start condition. The `newCondition` argument is an integer, that can range from zero to one less than the number of start conditions.

Start conditions are assigned consecutive integer values, beginning with zero, in the order that they are listed in the %conditions section of the grammar specification file. You can use `ScannerTable.lookupCondition` to find the integer value of a start condition, given its name. When a scanner is first created, the start condition is initialized to zero.

`public final int` **`tokenLength`** `()`

Returns the length of the token string. The token string is the string that matches the token regular expression.

This method may be used only by a token factory.

`public final void` **`setTokenLength`** `(int newTokenLength)`

Sets the length of the token string. A token factory can use this method to inform the scanner of the true length of the token, in cases where a regular expression can't determine the true length.

The `newTokenLength` argument must be greater than or equal to zero, and less than or equal to `textLength()`.

A token factory that calls this function may not reject the token.

This method may be used only by a token factory.

`public final String` **`tokenToString`** `()`

Creates a new `String` object, which contains the entire token string.

This method is equivalent to `tokenToString(0, tokenLength())`.

This method may be used only by a token factory.

`public final String` **`tokenToString`** `(int off, int len)`

Creates a new `String` object, which contains a substring of the token string. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring.

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `tokenLength()`.

This method is equivalent to `tokenToString().substring(off, off+len)`.

This method may be used only by a token factory.

`public final char` **`tokenCharAt`** `(int off)`

Obtains one character from the token string, and returns it as a `char`. The `off` argument is the position of the character.

An exception is thrown if `off` is less than zero, or greater than or equal to `tokenLength()`.

This method is equivalent to `tokenToString().charAt(off)`.

This method may be used only by a token factory.

```
public final void tokenToChars (int off, int len,
   char[] dst, int dstOff)
```

Obtains a substring of the token string, and copies its characters into a `char` array. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring. The characters are stored into array `dst`, beginning at array index `dstOff`. (Thus, `dst[dstOff]` receives the first character of the substring.)

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `tokenLength()`, or if `dstOff` is negative, or if `dstOff+len` is greater than `dst.length`.

This method is equivalent to `tokenToString().getChars(off, off+len, dst, dstOff)`.

This method may be used only by a token factory.

`public final int` **`contextLength`** `()`

Returns the length of the context string.

If a token is defined with a right context, then the context string matches the catenation of the token regular expression and the context regular expression. For example, if the token definition is “`a+/b+`”, and the next six input characters are “`aaabbc`”, then the context string is “`aaabb`” and the token string is “`aaa`”.

If a token is defined without a right context, then the context string is the same as the token string.

This method may be used only by a token factory.

`public final String` **`contextToString`** `()`

Creates a new `String` object, which contains the entire context string.

This method is equivalent to `contextToString(0, contextLength())`.

This method may be used only by a token factory.

`public final String` **`contextToString`** `(int off, int len)`

Creates a new `String` object, which contains a substring of the context string. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring.

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `contextLength()`.

This method is equivalent to `contextToString().substring(off, off+len)`.

This method may be used only by a token factory.

`public final char` **`contextCharAt`** `(int off)`

Obtains one character from the context string, and returns it as a `char`. The `off` argument is the position of the character.

An exception is thrown if `off` is less than zero, or greater than or equal to `contextLength()`.

This method is equivalent to `contextToString().charAt(off)`.

This method may be used only by a token factory.

```
public final void contextToChars (int off, int len,
   char[] dst, int dstOff)
```

Obtains a substring of the context string, and copies its characters into a `char` array. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring. The characters are stored into array `dst`, beginning at array index `dstOff`. (Thus, `dst[dstOff]` receives the first character of the substring.)

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `contextLength()`, or if `dstOff` is negative, or if `dstOff+len` is greater than `dst.length`.

This method is equivalent to `contextToString().getChars(off, off+len, dst, dstOff)`.

This method may be used only by a token factory.

```
public final int textLength ()
```

Returns the number of characters currently available in the scanner's input buffer.

Because the scanner reads input in large gulps, the scanner's input buffer typically contains much more input text than the context string. This function returns the amount of input currently available. If the amount is insufficient, you can call `readData` to obtain additional input text.

This method may be used only by a token factory.

```
public final String textToString (int off, int len)
```

Creates a new `String` object, which contains a substring of the text currently available in the scanner's input buffer. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring.

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `textLength()`.

For example, `textToString(0, textLength())` returns a `String` containing the entire contents of the scanner's input buffer.

This method may be used only by a token factory.

```
public final char textCharAt (int off)
```

Obtains one character from the scanner's input buffer, and returns it as a `char`. The `off` argument is the position of the character.

An exception is thrown if `off` is less than zero, or greater than or equal to `textLength()`.

This method is equivalent to `textToString(0, textLength()).charAt(off)`.

This method may be used only by a token factory.

```
public final void textToChars (int off, int len,
   char[] dst, int dstOff)
```

Obtains a substring of the scanner's input buffer, and copies its characters into a `char` array. The `off` argument is the position of the first character in the substring, and the `len` argument is the length of the substring. The characters are stored into array `dst`, beginning at array index `dstOff`. (Thus, `dst[dstOff]` receives the first character of the substring.)

An exception is thrown if `off` is less than zero, or if `off+len` is greater than `textLength()`, or if `dstOff` is negative, or if `dstOff+len` is greater than `dst.length`.

This method is equivalent to `textToString(0, textLength()).getChars(off, off+len, dst, dstOff)`.

This method may be used only by a token factory.

public abstract int **readData** () throws IOException, SyntaxException

Reads some amount of additional input into the scanner's input buffer. The return value is the number of additional characters that were read. The return value is zero if the input is at end-of-file.

Calling this method causes the value of `textLength()` to increase.

This method invalidates any previous return values obtained from the methods `tokenStart`, `dataEnd`, `rawByteText`, and `rawCharText`. If you are using these methods (which is not recommended), you must call them again to obtain updated values.

public abstract boolean **isByteText** ()

Returns `true` if the scanner's input buffer has type `byte[]`. Returns false if the scanner's internal buffer has type `char[]`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

For example, the following function returns a string that contains the entire contents of the scanner's input buffer.

```
String inputBufferToString (Scanner scanner)
{
   if (scanner.isByteText())
   {
      return new String (scanner.rawByteText(), 0,
         scanner.tokenStart(), scanner.dataEnd() – scanner.tokenStart());
   }
   else
   {
      return new String (scanner.rawCharText(),
         scanner.tokenStart(), scanner.dataEnd() – scanner.tokenStart());
   }
}
```

public byte[] **rawByteText** ()

If the scanner's input buffer has type `byte[]`, then this method returns the input buffer. Otherwise, it returns `null`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

public char[] **rawCharText** ()

If the scanner’s input buffer has type `char[]`, then this method returns the input buffer. Otherwise, it returns `null`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

public final int **tokenStart** ()

Returns the array index of the first character in the input buffer. This character is also the first character of the current token string.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

public final void **setTokenStart** (int newTokenStart)

Sets the array index of the first character in the input buffer.

This function can be used to advance the scanner’s current position in the input. Normally, the scanner advances its position automatically as tokens are processed. So you only need to call this function if you are accessing the input buffer directly.

The position can only be moved forward (not backward), and it can’t be moved past the end of the data currently in the input buffer. In other words, the `newTokenStart` argument must be greater than or equal to `tokenStart()`, and less than or equal to `dataEnd()`. If you call this method, you need to call `setTokenLength` to specify where scanning should resume, and you also need to adjust the line and column numbers. A token factory that calls this method must not reject the token.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

public final int **dataEnd** ()

Returns the array index of the last character in the input buffer, plus one. Thus, the number of characters in the input buffer is `dataEnd()-tokenStart()`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

public abstract String **rawTextToString** (int off, int len)

Copies text from the scanner’s input buffer into a `String` object. Text is copied beginning at array index `off`. The number of characters copied is `len`.

An exception is thrown if `off` is less than `tokenStart()`, or if `off+len` is greater than `dataEnd()`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

```
public abstract char rawTextCharAt (int off)
```

Obtains one character from the scanner's input buffer and returns it as a `char`. The character is obtained from array index `off`.

An exception is thrown if `off` is less than `tokenStart()`, or greater than or equal to `dataEnd()`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

```
public abstract void rawTextToChars (int off, int len,
   char[] dst, int dstOff)
```

Copies text from the scanner's input buffer into a `char` array. Text is copied beginning at array index `off`. The number of characters copied is `len`. The text is stored into the `dst` array, beginning at index `dstOff`.

An exception is thrown if `off` is less than `tokenStart()`, or if `off+len` is greater than `dataEnd()`, or if `dstOff` is negative, or if `dstOff+len` is greater than `dst.length`.

A token factory may use this method if it needs direct access to the input buffer. Direct access to the input buffer is not recommended, and should be avoided unless absolutely necessary.

```
public final void countLine ()
```

A token factory that recognizes a line-end can call this method to increment the line number. This method does the following:

- Increment the `line` field of the `Token` object.
- Set the `column` field of the `Token` object to `1-tokenLength()`.

Note that the scanner automatically adds `tokenLength()` to the `column` field after the token factory returns. Therefore, the above actions cause the first character after the end of the token string to be column 1 in the next line.

This method may be used only by a token factory.

```
public final ScannerClient client ()
```

Returns the `ScannerClient` object that was used to create this scanner.

```
public final ScannerTable scannerTable ()
```

Returns the `ScannerTable` object that was used to create this scanner.

```
public final int tokenIndex ()
```

If called by a token factory, this method returns the index number of the current token definition. If called by a client (that is, not during a call to `nextToken`), this method returns the index number of the last token definition that was recognized. Token definitions are assigned consecutive index numbers, beginning with zero, in the order they appear in the grammar specification file.

7.23 Interface ScannerClient

```
public interface ScannerClient
{
   // Methods
   public void scannerEOF (Scanner scanner, Token token);
   public void scannerUnmatchedToken (Scanner scanner, Token token);
}
```

`ScannerClient` is an interface that represents the client of a `Scanner` object.

`public void` **`scannerEOF`** `(Scanner scanner, Token token)`

The scanner calls this routine when it reaches end-of-file.

The `token` argument is set up as follows:

`token.number` = `Token.EOF`

`token.value` = `null`

`token.file` = source filename

`token.line` = current line number

`token.column` = current column number

A typical implementation of this method might check for run-on comments or unterminated blocks.

It is also possible for an implementation to insert additional tokens at the end of the file. Simply write the desired values into `token.number` and `token.value`. Be sure to set a flag if you do this (possibly using the `public` field `scanner.clientParams`), because doing so will make the scanner call `scannerEOF` again. The method `scannerEOF` must eventually allow the scanner to return an end-of-file token.

See the documentation for method `scannerEOF` in class `CompilerModel` for an example of how to implement this method.

`public void` **`scannerUnmatchedToken`** `(Scanner scanner, Token token)`

The scanner calls this routine when it cannot match a token.

The `token` argument is set up as follows:

`token.file` = source filename

`token.line` = current line number

`token.column` = current column number

A typical implementation of this method would print an error message, using the fields of the `token` argument to pinpoint the position of the error.

See the documentation for method `scannerUnmatchedToken` in class `CompilerModel` for an example of how to implement this method.

7.24 Class ScannerTable

```
public class ScannerTable implements Cloneable
{
   //Constructor
   public ScannerTable ();

   // Methods
   public int lookupCondition (String name);
   public void linkFactory (String tokenName, String linkName,
      TokenFactory factory);
   public void setTrace (ErrorOutput traceOut);
   public void writeToStream (DataOutput stream) throws IOException;
   public void readFromStream (DataInput stream) throws IOException;
   public String checkInvariant ();
   public boolean writeToJavaSource (PrintStream stream, String packageName,
      String className, boolean useRLE);
   public Object clone ();
}
```

`ScannerTable` holds the tables required for `Scanner`.

Generating Tables

The class `GenFrontEnd` contains functions to generate scanner tables. Using `GenFrontEnd`, you supply a specification of the tokens, and then `GenFrontEnd` creates the resulting `ScannerTable` object.

Once you have a `ScannerTable` object, you will probably want to save it for future use. There are two ways to do this.

- You can use `writeToStream` to write the `ScannerTable` to an output stream. This can be used to save the tables into a file.
- Alternatively, you can use `writeToJavaSource` to create a subclass of `ScannerTable` that contains all the tables as literals. This produces a Java source file that you can compile.

The class `GenGUI` provides a graphical user interface that invokes the parser generator, and then writes the scanner tables either to a file, or to Java source code. The class `GenMain` provides a simple command-line interface that invokes the parser generator, and then writes the scanner tables either to a file, or to Java source code.

Using the Tables

To use the tables, you first have to retrieve them from where they are stored.

- If the tables are stored in a file, you can create a new `ScannerTable` object and then call `readFromStream` to load the tables from an input stream. Refer to method `readGenFile` in class `CompilerModel` for an example of how to do this.
- If the tables are stored as literals in a subclass, you can create a new instance of the subclass, and cast the result to type `ScannerTable`. When you create the subclass object, the tables are automatically loaded from the literals.

After retrieving the tables, you need to link in the token factories. To do this, call `linkFactory` once for each combination of token name and link name that has a token factory. `ScannerTable` automatically constructs an internal table containing all the token factories. (If you don't supply factories for all token definitions, `ScannerTable` automatically supplies default factories. If a token definition's parameter is nonzero, the default factory assembles a token whose number is the parameter, and whose value is `null`; otherwise, the default factory discards the token.)

If you want to enable tracing, call `setTrace` and specify the destination for tracing output. If you enable tracing, `ScannerTable` inserts code to write a message every time a token definition is recognized. This is useful for debugging.

Next, if you are using multiple start conditions, you need to link the start conditions. To do this, use `lookupCondition` to get the number of each start condition, given its name. You need to save these numbers. These start condition numbers are the numbers you must pass to `Scanner` when setting the start condition. (Note that unlike with token factories, `ScannerTable` does not construct and save a start condition linkage table, because `Scanner` doesn't require one.)

Finally, pass the `ScannerTable` object to `Scanner`.

Other Functions

`ScannerTable` defines other functions that can be used for special purposes.

You can call `clone` to create a copy of a `ScannerTable` object. This is a shallow copy which contains the same tables. This is useful if you want to create two different `ScannerTable` objects, with the same scanning tables but different sets of token factories. A multi-pass compiler might need to do this, supplying a different set of token factories for each pass. Note that for this to work, you must call `clone` before making any calls to `linkFactory`.

You can call `lookupCondition()` to get the number of any start condition, given its name.

You can call `checkInvariant` to make `ScannerTable` check its tables for internal consistency. This can be used after retrieving the tables from a file, if you want to check that the tables were read properly. It can also be used after linking factories to verify that every factory actually corresponds to at least one token definition.

`public` **`ScannerTable`** `()`

Create a new `ScannerTable` object. The object contains no scanning tables, and it has no linked token factories. Tracing is disabled.

`public int` **`lookupCondition`** `(String name)`

Given the name of a start condition, this method returns the corresponding condition number. If the name is invalid, this method returns `-1`. Start condition numbers are assigned consecutively, beginning with zero, in the order that start conditions are listed in the %categories section of the grammar specification file.

```
public void linkFactory (String tokenName, String linkName,
   TokenFactory factory)
```

Links a token factory into the scanner tables.

The `tokenName` argument is the name of the token.

The `linkName` argument is the token definition's link name. If the token definition does not have a link name, this should be an empty string (not `null`).

The `factory` argument is the token factory for the token definition. Note that if there is more than one token definition with the same token name and link name, this factory will be used for all of them.

If no factory is provided for a given token, then a default factory is used. If the token definition's parameter is zero, the default factory discards the token. If the token definition's parameter is nonzero, the default factory assembles a token whose number equals the parameter, and whose value is `null`.

Note that there is no way to "unlink" a token factory once it has been linked.

`public void` **`setTrace`** `(ErrorOutput traceOut)`

Sets the destination for tracing output. If the argument is `null`, then tracing is disabled.

If this function is never called, then tracing is disabled.

When tracing is enabled, the token name, link name, parameter, position, and text are written out every time a token definition is recognized. This lets you view the operation of the scanner. Caution: This produces a large amount of output!

`public void` **`writeToStream`** `(DataOutput stream) throws IOException`

Write the scanner tables to the `stream`, in binary format.

The binary format is fairly complicated. If you need to know the format, look at the source code for the class `ScannerTable`.

`public void` **`readFromStream`** `(DataInput stream) throws IOException`

Read the scanner tables from the `stream`, in binary format.

This function performs only a cursory check of data integrity. If you want to fully check data integrity, call `checkInvariant` after calling `readFromStream`.

`public String` **`checkInvariant`** `()`

This method checks the object to make sure that its contents are internally consistent.

If the contents are consistent, the method returns `null`.

If any inconsistency is found, the method returns a `String` describing the problem.

```
public boolean writeToJavaSource (PrintStream stream, String packageName,
   String className, boolean useRLE)
```

This method writes a Java source file for a subclass of `ScannerTable`. The subclass contains all the generated scanner tables, written as literal constants.

The constructor for the subclass automatically sets all the scanner tables to the literal constants. This saves the time that would otherwise be required to read the tables in from a file.

The Java source file is written to the specified `stream`. The `packageName` argument is the name of the subclass's package, and the `className` argument is the name of the subclass itself.

If `useRLE` is `true`, the tables are compressed with RLE encoding. The tables are automatically expanded the first time an object of this class is constructed. This reduces the size of the class file, at the expense of the time required for RLE decoding. (Note, however, that the Java VM Specification implies a limit of approximately 8000 array initializer elements per class. Thus, RLE encoding may be mandatory for large tables.)

The return value is `stream.checkError`, which is `true` if there was an I/O error.

`public Object` **`clone`** `()`

Creates a copy of the tables. This is a shallow copy. The clone object shares its contained tables with the original.

7.25 Class SyntaxException

```
public class SyntaxException extends Exception
{
   // Constructors
   public SyntaxException ()
   public SyntaxException (String s)
}
```

`SyntaxException` is the root exception class for all exceptions that result from scanning and parsing. `SyntaxException` (or one of its subclasses) is thrown when an error is encountered that makes it impossible to continue scanning or parsing the input.

`public` **`SyntaxException`** `()`

Creates a new exception object with `null` as its error message string.

`public` **`SyntaxException`** `(String s)`

Creates a new exception object with `s` as its error message string.

7.26 Class Token

```
public class Token
{
   // Fields that define the contents of the token
   public int number;
   public Object value;
   public String file;
   public int line;
   public int column;

   // Special value for line and column numbers
   public static final int noPosition = ErrorOutput.noPosition;

   // Special token types
   public static final int EOF = 0;
   public static final int escapeInsertStream = -1;

   // Constructors
   public Token ();
   public Token (int number, Object value, String file,
      int line, int column);
   public Token (Token other);

   // Method
   public void copyFrom (Token other);
}
```

A `Token` object contains a token. A token consists of:

- An integer that represents its type.
- An `Object` that represents its value.
- A `String` that is the name of the file where the token came from.
- Two integers that identify the line number and column number where the token came from.

`Token` objects are used throughout Invisible Jacc. A scanner produces a stream of `Token` objects as output, a preprocessor filters a stream of `Token` objects, and a parser accepts a stream of `Token` objects as input. In addition, `Token` objects are used to specify the position in the source file where an error occurs.

Data fields within a `Token` object are declared `public` to allow efficient access.

```
public int number;
```

The token number. This indicates the type of token.

- A positive integer denotes an ordinary token.
- Zero denotes an *end-of-file token*.
- A negative integer denotes an *escape token*.

`public Object **value**;`

The token's value. This can be an arbitrary object, and it can be `null`.

`public String **file**;`

The source file where the token originated. This can be `null`.

`public int **line**;`

The line number within the source file where the token originated. This can be `noPosition` to indicate that the line number is unknown or unspecified.

It is expected that this field may be an approximation of the token's actual position.

`public int **column**;`

The column number within the line where the token originated. This can be `noPosition` to indicate that the column number is unknown or unspecified.

It is expected that this field may be an approximation of the token's actual position. Also, negative values may be used to indicate a position prior to the specified line.

`public static final int **noPosition** = ErrorOutput.noPosition;`

Constant that defines an unknown or unspecified line or column.

`public static final int **EOF** = 0;`

Constant that defines the end-of-file token.

`public static final int **escapeInsertStream** = -1;`

Constant that defines the insert-stream escape token. For this token, the field `value` contains a `TokenStream` object. The preprocessor class `PreprocessorInclude` recognizes this escape token and replaces it with all the tokens of the stream.

`public **Token** ()`

Create a new `Token` object initialized as follows: `number` is set to zero, `value` is `null`, `file` is `null`, `line` equals `noPosition`, and `column` equals `noPosition`.

```
public Token (int number, Object value, String file,
   int line, int column)
```

Create a new `Token` object, and initialize its fields from the constructor's arguments.

```
public Token (Token other)
```

Create a new `Token` object, and initialize its fields by copying the fields of the `other Token` object.

```
public void copyFrom (Token other)
```

Copy all the fields from the `other Token` object into this `Token` object.

7.27 Class TokenFactory

```
public abstract class TokenFactory
{
   // Return values
   public static final int assemble = 0;
   public static final int discard = 1;
   public static final int reject = 2;

   // Method
   public abstract int makeToken (Scanner scanner, Token token)
      throws IOException, SyntaxException;
}
```

`TokenFactory` is used by the scanner to create `Token` objects.

Every token definition has an associated object of class `TokenFactory`. Whenever the scanner recognizes a token definition, it calls the `makeToken` method of the corresponding `TokenFactory` object. The `makeToken` method can do one of three things:

- It can *assemble* the token. In this case, the scanner returns the assembled token to its client.
- It can *discard* the token. In this case, the scanner skips past the matching string and then attempts to recognize another token.
- It can *reject* the token. In this case, the scanner calls the next candidate token factory.

A compiler typically defines several subclasses of `TokenFactory`, one subclass for each token definition that requires special action. Each subclass of `TokenFactory` is called a *token factory*.

It is recommended that a `TokenFactory` object should not store any state information. State information should be stored either in a "global" client object, or in the scanner's `clientParams` variable. In Java 1.1, a `TokenFactory` could be declared as an inner class nested inside the global client, which would give the `TokenFactory` convenient access to the global client's variables and methods.

```
public static final int assemble = 0;
```

The `makeToken` method returns `assemble` if it assembles a token.

```
public static final int discard = 1;
```

The `makeToken` method returns `discard` if it discards a token.

```
public static final int reject = 2;
```

The `makeToken` method returns `reject` if it rejects a token.

```
public abstract int makeToken (Scanner scanner, Token token)
   throws IOException, SyntaxException
```

Fill in a `Token` object.

The `scanner` argument is the `Scanner` object. You can use this argument to make calls back to the scanner, as described below.

The `token` argument is a `Token` object into which you should assemble the token. The scanner initializes the `Token` object as follows:

`token.number` = the token parameter from the scanner tables

`token.value` = `null`

`token.file` = the file name

`token.line` = the current line number

`token.column` = the current column number

If this token corresponds to a terminal symbol in the grammar, then the token parameter is typically the numerical value of the terminal symbol. Therefore, it is normally not necessary for the token factory to explicitly fill in the `token.number` field.

The line number and column number identify the position of the first character in the token.

Assembling a Token

If `makeToken` assembles a token, it must return `assemble`. In addition, it must set `token.number` to the token number (if the scanner-supplied value is not correct), and it must set `token.value` to the token value (if the value is not `null`).

After `makeToken` returns, the scanner adds the length of the token string to `token.column`, and also advances its input pointer by the length of the token string. This has the effect of moving past the token. Then, the scanner returns the assembled token to its client.

There is no restriction on the type of token that can be assembled. The token can be a normal token, an end-of-file token, or an escape token. However, normally a token factory would not return an end-of-file token, since the scanner generates one automatically when it reaches the end of its input.

Discarding a Token

If `makeToken` discards a token, it must return `discard`.

After `makeToken` returns, the scanner adds the length of the token string to `token.column`, and also advances its input pointer by the length of the token string. This has the effect of moving past the token. Then, the scanner searches for another token.

Rejecting a Token

If `makeToken` rejects a token, it must return `reject`.

After `makeToken` returns, the scanner calls the next candidate token factory. This is useful if token recognition cannot be accomplished with regular expressions alone.

Note that rejecting a token does not cause the scanner to re-scan the input. Rather, the scanner remembers the results of the prior scan. For this reason, `makeToken` must not call `scanner.setTokenStart` or `scanner.setTokenLength` if it rejects the token (because those two methods mess up the stored scan results).

Also note that rejecting a token may cause the same token factory to be called again, with a different (usually shorter) token string.

Getting the Token String

The scanner provides methods for the token factory to obtain the string that matches the token regular expression. This is called the *token string*.

- The token factory can call `scanner.tokenToString` to obtain a `String` object which contains all or part of the matching token string.
- The token factory can call `scanner.tokenCharAt` to obtain a single character of the matching token string.
- The token factory can call `scanner.tokenToChars` to copy the matching token string into a `char` array.
- The method `scanner.tokenLength` returns the length of the matching token string.

Getting the Context String

The scanner also provides methods to match the *context string*. If a token definition includes a right context, then the context string matches the catenation of the token regular expression with the right context regular expression. (For example, if the token definition is “a+/b+” and the next six input characters are “aaabbc”, then the context string is “aaabb” and the token string is “aaa”.) If a token definition does not have a right context, then the context string is the same as the token string.

- The token factory can call `scanner.contextToString` to obtain a `String` object which contains all or part of the matching context string.
- The token factory can call `scanner.contextCharAt` to obtain a single character of the matching context string.
- The token factory can call `scanner.contextToChars` to copy the matching context string into a `char` array.
- The method `scanner.contextLength` returns the length of the matching context string.

Altering Line and Column Position

Optionally, `makeToken` can alter the values of `token.line` and/or `token.column`. This is typically done if the matching token string contains an end-of-line. For example, a token that recognizes a single end-of-line could increment `token.line` and set `token.column` to `1-scanner.tokenLength()`.

The token factory can call `scanner.countLine` to increment `token.line` and to set `token.column` to `1-scanner.tokenLength()`. The method `scanner.countLine` is provided as a convenience to the token factory; there is nothing wrong with the token factory modifying `token.line` and `token.column` directly.

Altering Token Length

The length of the matching token string can be altered by calling `scanner.setTokenLength`. This is useful if the actual token length differs from the length recognized by the regular expression. Note that it is legal to specify a length of zero, however caution must be used in this case not to create an infinite loop.

Note that calling `scanner.setTokenLength` alters the value returned by subsequent calls to `scanner.tokenLength`.

Setting the Start Condition

The start condition can be changed by calling `scanner.setCondition`. The new start condition takes effect beginning with the next token scan.

Additionally, `scanner.condition` can be called to retrieve the current start condition.

Getting the Token Index Number

The token factory can call `scanner.tokenIndex` to get the index number of the current token definition. Token definitions are assigned consecutive index numbers, beginning with zero, in the order they appear in the grammar specification file.

Accessing Client Parameters

The token factory can access the public variable `scanner.clientParams`. This variable is dedicated to the client's use. The token factory and client can use it for any desired purpose. The variable `scanner.clientParams` would be the appropriate place to store any per-file state information.

Getting Additional Input Text

The scanner maintains an internal input buffer. Conceptually, the buffer begins at the start of the current token string, and it includes at least the token string and context string. Typically, it also includes additional input text beyond the context string.

- The function `scanner.textLength` returns the number of characters currently available in the input buffer.
- The token factory can call `scanner.textToString` to obtain a `String` object which contains part of the currently available input.
- The token factory can call `scanner.textCharAt` to obtain any single character of the currently available input.
- The token factory can call `scanner.textToChars` to copy part of the currently available input into a `char` array.
- If the token factory needs to read beyond the currently available input, it can call `scanner.readData`, which causes the scanner to read some amount of additional input data into the buffer.

Direct Access to the Input Buffer

In unusual cases, it may be necessary for the token factory to have direct access to the scanner's input buffer. The following functions allow this.

The scanner's input buffer may have type `byte[]` or `char[]`. Therefore, a token factory that wants to access the buffer directly must be able to handle either type of buffer. First, call `scanner.isByteText` to determine which type of buffer is in use. If the buffer type is `byte[]`, call `scanner.rawByteText` to obtain the internal input buffer. If the buffer type is `char[]`, call `scanner.rawCharText` to obtain the internal input buffer.

The method `scanner.tokenStart` returns the array index within the input buffer where the current token string begins. The method `scanner.dataEnd` returns the array index of the last available character in the input buffer, plus `1`. The token factory may only access the part of the buffer between these two indexes.

The token factory may call `scanner.readData` to read additional data into the buffer. Since `scanner.readData` can reallocate the buffer, any values previously obtained from `scanner.rawByteText`, `scanner.rawCharText`, `scanner.tokenStart`, and `scanner.dataEnd` become invalid. After calling `scanner.readData`, the token factory must call these other functions again to obtain updated values.

The token factory can call `scanner.setTokenStart` to discard input text. This function advances the scanner's current position in the input. The token factory can use this function to scan ahead an arbitrary distance in the input, perhaps to discard comments or embedded data blocks. If the token factory does this, it will almost certainly want to call `scanner.setTokenLength` to specify where scanning is to resume, and it will want to adjust `token.line` and `token.column` accordingly.

7.28 Interface TokenStream

```
public interface TokenStream
{
   // Methods
   public Token nextToken () throws IOException, SyntaxException;
   public void close () throws IOException;
}
```

A `TokenStream` object represents a stream of tokens. Each token may be read from the stream once and only once. Reading a token automatically advances the internal stream pointer to the next token.

Each token is packaged in a `Token` object.

End-of-stream is indicated by returning token number `0`, which is indicated symbolically as `Token.EOF`.

```
public Token nextToken () throws IOException, SyntaxException
```

Get the next token in the stream, and remove it from the stream.

When the end of the stream is reached, this method returns a `Token` object with `number = Token.EOF`. The end-of-stream token may be retrieved only once. (`TokenStream` differs from `Preprocessor` on this point.) If this method is called again after end-of-stream has been returned, its behavior is undefined.

In order to avoid the overhead of creating a `Token` object for each token, this method may return the same `Token` object repeatedly, and simply update the fields of the `Token` object for each call. Therefore, the caller must treat the returned `Token` object as read-only, and may use the returned `Token` object only until the next call to `nextToken`.

```
public void close () throws IOException
```

Close the token stream.

Typically, this method closes the file from which tokens are being obtained.

8 References

In writing Invisible Jacc, we relied on the following references:

[1] Alfred Aho and Jeffrey Ullman, *Principles of Compiler Design*, Addison-Wesley 1977, ISBN 0-201-00022-9.

[2] Charles Fischer and Richard LeBlanc, Jr., *Crafting a Compiler*, Benjamin/Cummings 1988, ISBN 0-8053-3201-4.

[3] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley 1996, ISBN 0-201-63451-1.

[4] Bell Laboratories, *Unix Programmer's Manual*, *Volume 2*, Holt, Rinehart and Winston, 1983, ISBN 0-03-061743-X

References [1] and [2] both provide a solid theoretical and practical background for writing scanners and parsers.

Reference [1] has a particularly good exposition of lexical analysis, regular expressions, and finite automata. If you can’t find a copy of reference [1], there is a new book by Aho, Sethi, and Ullman which covers essentially the same material.

Reference [2] has an especially good exposition of LR parsing. The book also describes the LM error repair algorithm, and Pager’s algorithm for optimized LR(1) grammars, neither of which is covered in reference [1].

Reference [3], in addition to being an essential reference for Java programmers, also contains an excellent example of an LALR(1) grammar. In particular, the LALR(1) grammar for Java shows how to resolve operator precedence, associativity, the “dangling else”, and other language problems, all without using an ambiguous grammar.

Reference [4] contains the user manuals for Yacc and Lex, which are probably the best-known and most widely used parser and scanner generators.