jacc: just another compiler compiler for Java
A Reference Manual and User Guide

Mark P. Jones
Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
20000 NW Walker Road, Beaverton, OR 97006, USA

January 30, 2003

1 Introduction

jacc is a parser generator for Java [3] that is closely modeled on Johnson's classic yacc parser generator for C [6]. It is easy to find other parser generators for Java including CUP [4], Antlr [10], JavaCC [8], SableCC [12], Coco/R [9], BYACC/Java [5], and the Jikes Parser Generator [11]. So why would you want to use jacc instead of one of these other fine tools?

In short, what makes jacc different from other tools is its combination of the following features:

- Close syntactic compatibility with Johnson's classic yacc parser generator for C (in so far as is possible given that the two tools target different languages);
- Semantic compatibility with yacc—jacc generates bottom-up/shift-reduce parsers for LALR(1) grammars with disambiguating rules;
- A pure Java implementation that is portable and runs on many Java development platforms;

- Modest additions to help users understand and debug generated parsers, including HTML output and tests for LR(0) and SLR(1) conflicts;
- Generated parsers that use the technique described by Bhamidipaty and Proebsting [1] for creating very fast yacc-compatible parsers by generating code instead of encoding the specifics of a particular parser in a set of tables as the classic yacc implementations normally do.

If you are looking for a yacc-compatible parser generator for Java, then I hope that `jacc` will also meet your needs, and that these notes will help you to use it!

In particular, these notes describe basic operation of `jacc`, including its command line options and the syntax of input files. They do not attempt to describe the use of shift/reduce parsing in generated parsers or to provide guidance in the art of writing yacc-compatible grammars or the process of understanding and debugging any problems that are reported as conflicts. For that kind of information and insight, you should refer to other sources, such as: the original yacc documentation [6], several versions of which are easily found on the web; the documentation for Bison [2], which is the GNU project's own yacc-compatible parser generator; or the book on Lex & Yacc by Levine, Mason, and Brown [7]. Please note that this is an early version of the documentation for `jacc`; I welcome any comments for suggestions that might help to improve either the tool or this documentation.

2 Command Line Syntax

The current version of `jacc` is used as a command line utility, using simple text files for input and output. The input to `jacc`—a context-free grammar, annotated with semantic actions, `jacc` directives, and auxiliary code fragments—should be placed in a file called `X.jacc`, for some prefix `X`. The parser generator is invoked with a simple command of the form:

```
jacc X.jacc
```

By default, `jacc` will generate two output files, one called `XParser.java` containing the implementation of a parser as a Java class `XParser`, and the

other a file `XTokens.java` that defines an interface called `XTokens` that specifies integer codes for each of the token types in the input grammar. Note that `jacc` writes all output files in the same directory as the input file, automatically replacing any existing file of the same name. `jacc` will also display a warning message if the input grammar results in any conflicts. Such conflicts can be investigated further by running `jacc` with either the `-v` or `-h` options described below.

The `jacc` command accepts several command line options that can be used to modify its basic behavior.

`-p` Do not attempt to write the `XParser.java` file. This option is typically used together with `-t` to test that a given input file is well-formed, and to detect and report on the presence of conflicts, without generating the corresponding parser and token interface.

`-t` Do not attempt to write the `XTokens.java` file.

`-v` Write a plain text description of the generated machine in the file `X.output`. The output file provides a description of each state, and concludes with brief statistics for the input grammar and generated machine. The following example shows the output that is generated for a state containing a shift/reduce conflict (the classic "dangling else" problem). The description: begins with a description of the conflict; lists the corresponding set of items (parenthesized numbers on the right correspond to rule numbers in the input grammar); and concludes with a table that associates each input symbol with an appropriate shift, reduce, or goto action (the period, '.', identifies a default action).

```
49: shift/reduce conflict (shift 53 and red'n 31) on ELSE
state 49 (entry on stmt)
   stmt : IF '(' expr ')' stmt_    (31)
   stmt : IF '(' expr ')' stmt_ELSE stmt    (32)

   ELSE shift 53
   . reduce 31
```

`-h` Generate a description of the generated machine in HTML in the file `XMachine.html`. The generated file uses the same basic output format

as `X.output`, but includes hyperlinks that can be used to link between generated states. You can also use a browser's back button to simulate the effect of reduce actions: if the right hand side of the rule has n symbols, then click the back button n times. As a result, the generated `XMachine.html` file can be used to step through the behavior of the generated machine on a particular sequence of input tokens.

`-a` Uses the LALR(1) strategy to resolve conflicts. This is the default behavior for `jacc`, and the most powerful strategy that it provides for using lookahead information to resolve any conflicts detected in the input grammar. If `jacc` does not report any conflicts when this strategy is used, then the input grammar is said to be LALR(1).

`-s` Uses the SLR(1) strategy to resolve conflicts; If `jacc` does not report any conflicts when this strategy is used, then the input grammar is said to be SLR(1). For practical purposes, and noting that SLR(1) is weaker than LALR(1), this option is only useful only for understanding the formal properties of an input grammar.

`-0` Uses the LR(0) strategy to resolve conflicts; If `jacc` does not report any conflicts when this strategy is used, then the input grammar is said to be LR(0). For practical purposes, and noting that LR(0) is weaker than both SLR(1) and LALR(1), this option is only useful only for understanding the formal properties of an input grammar.

Multiple command line options can be combined into a single option. For example `jacc -pt X.jacc` has the same effect as `jacc -p -t X.jacc` If no arguments are specified, then `jacc` displays the following brief summary of command line syntax:

```
usage: jacc [-ptvhas0] file.jacc
 -p  do not generate parser
 -t  do not generate token specification
 -v  output plain text description of machine
 -h  output HTML description of machine
 -a  treat as LALR(1) grammar (default)
 -s  treat as SLR(1) grammar
 -0  treat as LR(0) grammar
```

3 Input File Syntax

The basic structure of a `jacc` input file is as follows:

... directives section ...
%%
... rules section ...
%%
... additional code section ...

The second %% and the additional code section that follows it can be omitted if it is not required. Comments may be included in any part of a `.jacc` file using the standard conventions of C++ and Java: the two characters `//` introduce a comment that spans to the end of the line in which it appears; the two characters `/*` introduce a C-style comment that spans all characters, possibly over multiple lines, up until the next occurrence of a closing comment marker `*/`.

3.1 The Directives Section

The opening section of a `.jacc` file is a sequence of directives that can be used to customize certain aspects of the generated Java source file (Section 3.1.1), to specify the interface between lexical analysis and parsing (Section 3.1.2), and to describe properties of the terminal and nonterminal symbols in the input grammar (Section 3.1.3).

3.1.1 Customizing the Generated Parsers

In this section we describe the directives that are used to specify and customize Java-specific aspects of `jacc`-generated parsers:

- The `%package` directive, which should be followed by a single qualified name, is used to specify the package for the parser class and token interface that are generated by `jacc`. For example, if an input file `Lang.jacc` contains the directive

```
%package com.compilersRus.compiler.parser
```

then each of the jacc-generated Java source files will begin with the declaration:

```
package com.compilersRus.compiler.parser;
```

- A code block is introduced by the sequence %{ and terminated by a later %}. All of the code in between these two markers is included at the beginning of the XParser.java file, immediately after any initial package declaration. This is typically used to specify any import statements that is needed by the code that appears in semantic actions or at the end of the .jacc source file. The following example shows a typical use:

  ```
  %{
  import java.io.File;
  import mycompiler.Lexer;
  import java.net.*;
  %}
  ```

  This declaration could also be used to provide definitions for auxiliary classes that are needed by the main parser, but this is not recommended; such definitions could instead be placed in a separate .java file, and including them in the .jacc source instead could distract a reader from more important aspects of the parser's specification.

  Code blocks like this should not be used to introduce a Java package declaration into the generated code. The %package directive provides a better way to specify the package because it will generate an appropriate declaration in both the parser source file and the tokens interface.

  Note that jacc does not attempt to determine if the text in a code block is valid; errors will not be detected until you attempt to compile the generated source files.

- A %class directive, followed by a single identifier, is used to change the name of the class that is used for the generated parser. For example, if the source file X.jacc specifies %class Y, then the generated parser will be called Y and will be written to the file Y.java (instead of the default behavior, which is to create a class XParser in the file XParser.java). Of course you should ensure that the parser class and the token interface have distinct names.

- An `%interface` directive, followed by a single identifier, is used to change the name of the interface that records numeric codes for input tokens. For example, if the source file `X.jacc` specifies `%interface Y`, then the generated interface will be called `Y` and will be written to the file `Y.java` (instead of the default behavior, which is to create an interface `XTokens` in the file `XTokens.java`).

- An `%extends` declaration is used to specify the super class for the parser. For example, if `Lang.jacc` specifies `%extends Phase`, then the generated parser in `LangParser.java` will begin with the line:

  ```
  class LangParser extends Phase implements LangTokens
  ```

  If no `%extends` directive is included in a `jacc` source file, then there will be no `extends` clause in the generated Java file either (i.e., the parser will be a direct subclass of `java.lang.Object`).

- The `%implements` directive, which should be followed immediately by the name of a class, is used to specify which interfaces are implemented by the generated parser. For example, if `Lang.jacc` specifies `%implements IX`, then the generated parser in `LangParser.java` will begin with the line:

  ```
  class LangParser implements IX, LangTokens
  ```

  Note that the tokens interface, in this case `LangTokens`, is automatically included in the list of implemented interfaces to ensure that the generated parser has access to the symbolic codes that are used to represent token types. Multiple `%implements` declarations can be included in a `.jacc` input file to specify multiple implemented interfaces.

3.1.2 Customizing the Lexer/Parser Interface

In this section, we describe the directives that are used to specify and customize the interface between lexical analysis and `jacc`-generated parsers.

- A `%next` directive is used to specify the code sequence (a single Java expression) that should be used to invoke the lexer and return the integer code for the next token. By default, `jacc` uses `lexer.nextToken()`

for this purpose, with the assumption that the `lexer` will be defined as an instance variable of the parser class, and that it will provide a method `int nextToken()`. Different mechanisms for retrieving input tokens can be set using a suitable `%next` directive. For example, in classic yacc parsers, the code for the lexer is invoked using a call to `yylex()`. To use the same method with `jacc`, we must include the following directive:

```
%next yylex()
```

and then add a suitable implementation for `yylex()` as a method in the parser class. A `%next` directive extends to the end of the line on which it appears. The generated parser will either fail to compile, or else give incorrect results if the expression specified by `%next` is not well-formed. In generated code, the expression used to read the next token will always be enclosed in parentheses (to avoid the possibility of a precedence-related misparse) and will always be the last thing on the line (to avoid any problems that might occur if the `%next` string were to end with a single line comment).

- A `%get` directive is used to specify the code sequence (a single Java expression) that should be used to obtain the integer code for the current token without advancing the lexer to a new token. By default, `jacc` uses `lexer.getToken()` for this purpose, with the assumption again that the `lexer` will be defined as an instance variable of the parser class, and that it will provide a suitable method `int getToken()`. Different mechanisms can be implemented using a suitable `%get` directive. For example, if the integer code for the current token is recorded in a instance variable `token` of the parser class, then the following directive should be used:

  ```
  %get token
  ```

  The `%get` directive uses the same syntactic conventions as `%next`; see above for further details.

- A `%semantic` directive is used to specify the type of the semantic values that are passed as token attributes from the lexer or constructed during parsing when reduce actions are executed. By default, `jacc` uses the

`java.lang.Object` type for semantic values, but a different type can be specified using an appropriate `%semantic` directive, as in the following example:

```
%semantic int
```

In yacc, the same effect is most commonly achieved by means of a `#define YYSTYPE int` preprocessor directive or by using a `%union` directive. Neither Java or `jacc` support unions, but the same effect can be achieved by defining a base class `Semantic` with a subtype for each different types of semantic value that is needed.

An additional colon followed by a code string can be used to specify an expression for reading the semantic value of the current token. By default, `jacc` uses `lexer.getSemantic()` for this purpose. The following example shows how a different method can be used, in this case assuming, as in classic yacc, that the lexer stores the semantic value of each token as it is read in a variable called `yylval`:

```
%semantic int: yylval
```

Once again, `jacc` uses the same syntactic conventions for the code sequence specified here as as used for the `%next` and `%get` directives.

3.1.3 Specifying Token and Nonterminal Properties

In this section, we describe the `jacc` directives that are used to specify properties of the terminal and nonterminal symbols in the input grammar. As much as possible, `jacc` uses the same syntax as yacc for these directives.

- A `%start` directive, followed immediately by the name of a nonterminal, is used to specify the start symbol for the grammar. If there is no `%start` directive in an input file, then the first nonterminal that is mentioned in the rules section of the input is used as the start symbol.
- The `%token` directive is used to define terminal symbols that are used in the grammar. By convention, terminals are usually written using only upper case letters and numeric digits. (Although, in theory, any

Java identifier could be used.) The following example uses a `%token` directive to define six tokens that might be used in the parser for a programming language like C or Java:

```
%token IF THEN ELSE FOR WHILE DO
```

The tokens interface that `jacc` generates will also use these same identifiers as the names for token codes, assigning arbitrary, but distinct small integer constants to each one. Any part of a program that needs access to these symbolic constants—most likely in those parts of the code having to do with lexical analysis—should include this interface in the `implements` clause of the corresponding classes. This will allow the code to use these symbolic constants directly, without the need for a qualifying class name prefix.

It is also possible to use single character literals as terminal symbols, which can make grammars a little easier to read. It is not actually necessary to declare such tokens explicitly in a `%token` definition, but it is usually considered good practice to do so for the benefit of documenting the symbols that are used, as in the following example:

```
%token '(' '[' '.' ';' ',' ']' ')'
```

For examples like these, `jacc` uses the corresponding integer code for each character as the token code (and automatically avoids using that same code for symbol token names like `IF`, `THEN`, and `ELSE` in the example above). For example, a lexer might indicate that it has seen an open parenthesis token by executing `return '(';`.

It is not uncommon for different token types to be associated with different types of semantic value. For example, the numeric value of an integer literal token might be captured in an object of type `Integer` (i.e., `java.lang.Integer`), while the text of a string literal or an identifier might be represented by a `String` object. Information like this can be recorded by including the desired type between `<` and `>` symbols immediately after the `%token` directive, as in the following examples:

```
%token <Integer> INTEGER
%token <String>  STRING_LITERAL IDENTIFIER
%token <java.net.URL> URL_LITERAL
%token <String[]>     PATH_STRING
```

Type annotations like this make sense only if all of the declared types are subtypes of the `%semantic` type that has been specified for the grammar. For the examples above, `java.lang.Object` is the only valid choice for the semantic type, because is it the only type that has `Integer`, `String`, `java.net.URL`, and `String[]` as subclasses. No explicit declaration of `%semantic` is needed in this case however because `java.lang.Object` is the default. Note from the examples above that it is possible to use qualified names and array types in these annotations. It is also possible to use primitive types, such as `int`, in declarations like this, but that is unlikely to be useful in practice: it would require a `%semantic` directive with the same type, and then the same type would automatically be used for all tokens in the grammar. (This is a consequence of the fact that there are no non-trivial subtyping relationships between primitive types in Java. It is also the reason why we used `Integer` for the `INTEGER` token, instead of a simple `int`.)

If a specific type has been declared for a given token, then `jacc` will automatically insert an appropriate cast into any generated code that refers to the semantic value for a token of that type. This can be very convenient because it saves programmers from having to write these casts explicitly. But it is also quite risky because the cast might fail at run-time if the actual semantic value does not have a compatible type. (This might occur, for example, if the lexer simply returns the wrong type of value, or if it fails to update the variable that records 'the semantic value of the current token' when a new token is read.) You should therefore be careful to balance the convenience of type annotations against the risks. It is the programmers responsibility to ensure that the annotations are correct because there is no way for `jacc` to do that!

- The `%left`, `%right`, and `%nonassoc` directives work just like `%token` directives, except that they also declare a *fixity*—a combination of precedence and associativity/grouping—for each of the tokens that are mentioned. This is particularly useful for describing the syntax of expressions using infix operators where fixity information can be used as an alternative to more verbose grammars that encode precedence and associativity requirements implicitly in their structure.

  As an example, a simple grammar for a arithmetic expressions might

include the following three directives to specify fixities for addition, subtraction, multiplication, division, and exponentiation:

```
%left     '+' '-'
%right    '*' '/'
%nonassoc '^'
```

To illustrate the different possibilities, we have declared the first two operators as associating to the left (so an expression like `1-2-3` will be parsed in the same way as `(1-2)-3`), the next two operators as associating to the right (so an expression like `1/2/3` will be parsed in the same way as `1/(2/3)`), and the last operator is nonassociative (so an expression like `1^2^3` will be treated as a syntax error).

Note that there is no explicit way to specify precedence values. Instead, `jacc` assigns the lowest precedence to all of the tokens mentioned in the first fixity declaration, the next highest precedence to the tokens mentioned in the next fixity declaration, and so on. In the example above, `+` and `-` have the lowest precedence, `*` and `/` have higher precedence, and `^` has the highest precedence. There is no way to specify that two operators should have the same precedence but different associativities.

In fact `jacc` can use fixity information in a more general way than these examples might suggest to resolve shift/reduce conflicts that seem to have little or nothing to do with infix operators. For example, the classic 'dangling else' problem can be resolved by assigning suitable fixities for the `THEN` and `ELSE` tokens. However, the resulting grammars can be harder to read, so this is not a technique that we would recommend, and we will not describe it any further here.

- A `%type` directive works just like a `%token` directive except that it is used to define nonterminal symbols that are used in the grammar. It is not strictly necessary to define nonterminals using `%type` because any identifier that is used on the left hand side of a production in the rules section of the input will be treated as a nonterminal. However, `%type` directives are still useful in practice, both to document the set of nonterminals that are used, and to associate types with nonterminals using the optional type annotations, as in the following example:

  ```
  %type <Expr> literal expr unary primary atom
  ```

As in the case of `%token` directives, `jacc` uses these annotations to guide the insertion of casts in the translation of semantic actions. In this case, however, the annotations indicate the type of value that is produced by the semantic actions that are associated with a particular nonterminal. Given the directives above, for example, the programmer should ensure that each production for the `expr` nonterminal assigns a value of type `Expr` (which includes any subclass of `Expr`) to `$$`.

3.2 The Rules Section

The rules section of the input to `jacc`, which follows immediately after the first `%%` marker, specifies a context free grammar for the language that the generated parser is intended to recognize. In addition, it associates each production with a fragment of code called a *semantic action* that the parser will execute each time that production is used. These semantic actions are able to access the semantic values corresponding to each of the terminal and nonterminal symbols on the right hand side of the rule, and are typically used to construct a portion of a parse tree, or else perform some other computation as appropriate.

3.2.1 Describing the Grammar

In fact the format of the rules section of a `jacc` input file can be described by a set of rules written in that same format—which conveniently doubles as a simple example (albeit without any semantic actions):

```
rules     : rules rule                 // a list of zero or more
          | /* empty */                // rules
          ;
rule      : NONTERMINAL ':' rhses ';'  // one rule can represent
          ;                            // several productions

rhses     : rhses '|' rhs              // one or more rhs'es
          | rhs                        // separated by "|"s
          ;
rhs       : symbols optPrec optAction  // the right hand side of
          ;                            // a production
```

```
symbols   : symbols symbol              // a list of zero or more
          | /* empty */                 // symbols
          ;
symbol    : TERMINAL                    // the union of terminals
          | NONTERMINAL                 // and nonterminals
          ;

optPrec   : PREC TERMINAL               // an optional precedence
          | /* empty */
          ;

optAction : ACTION                      // and optional action
          | /* empty */
          ;
```

The tokens in this grammar are `NONTERMINAL` (representing nonterminal symbols), `TERMINAL` (representing terminal symbols, which includes both identifiers and single character literals), `PREC` (which stands for the token `%prec`, and is used to assign a precedence—actually, a fixity—level to a given production), and `ACTION` (representing fragments of Java code that begin with `{` and ends with `}`). Of course the `':'`, `';'`, and `'|'`, symbols used in the grammar above are also simple terminal symbols.

This example also illustrates several common idioms that are used in `jacc` grammars to describe lists of zero or more items (e.g., `rules` and `symbols` describe lists of zero or more `rule` and `symbol` phrases, respectively); optional items (e.g., `optPrec` and `optAction` describe optional precedence annotations and actions, respectively); and lists of one ore more elements with an explicit separator (e.g., `rhses` describes a list of `rhs` phrases, each of which is separated from the next by a `|` token). There is nothing special about the uses of `/* empty */` in this example; they are just standard comments, but serve to emphasize when the right hand side of a production is empty.

Unlike `jacc`, the classic yacc allows semantic actions to appear between the symbols on the right hand side of a production, but this can sometimes result in strange behavior and lead to confusing error messages. Moreover, any example that is described using this feature of a yacc grammar can easily be translated into a corresponding grammar that does not (see the yacc documentation for details).

3.2.2 Adding Semantic Actions

As mentioned previously, semantic actions take the form of a fragment of Java code, enclosed between a matching pair of open `{` and close `}` braces. In the interests of readability, it is usually best to keep such code fragments short, moving more complex code into separate methods of the class so as to avoid obscuring the grammar. `jacc` does not make any attempt to ensure that the text between the braces is well-formed Java code; errors in the code will not be detected until the generated parser is compiles.

In fact the only thing that `jacc` does as it copies text from the original actions to the generated parser is to look for special tokens such as `$$`, `$1`, `$2`, and so on, which it replaces with references to the semantic value of the token on the left hand side of the production (for `$$`), the first token on the right (for `$1`), the second token on the right (for `$2`), and so on. The following example shows how these symbols might work in a parser for arithmetic expressions:

```
expr : expr '+' expr    { $$ = new AddExpr($1, $3); }
     | expr '-' expr    { $$ = new SubExpr($1, $3); }
     ;
```

The semantic actions shown here do not use the semantic value for the operator symbols in this grammar; that is, neither one mentions `$2`. However, both actions use `$1` to refer to the left operand and `$3` to refer to the right operand of the expression that is being parsed. Each of these names is replaced in the generated code with an expression that extracts the corresponding semantic value from the parser's internal stack. (If a type, `E`, has been declared for the `expr` nonterminal, then the generated code will also attempt to cast the values retrieved from the stack to values of type `E`.) On the other hand, the reference to `$$` will be replaced with a local variable that the generated parser uses, temporarily, to record the result of a reduce action. Normally, the code for a semantic action should only attempt to read the positional parameters `$1`, `$2`, etc..., and should only attempt to write to the result parameter `$$`, as in the examples above. In the parlance of attribute grammars, `$1` and `$2` are *inherited attributes*, while `$$` is a *synthesized attribute*.

The original yacc allows semantic actions to make use of parameters like `$0`, `$-1`, and `$-2` with zero or negative offsets as another form of inherited attribute. In cases like these, the parameter strings are replaced by references

into parts of the parser's internal stack that access values from the context of the current production rather than the right hand side of the production itself. This feature must be used with care and requires a fairly deep understanding of shift-reduce parsing internals to ensure correct usage and avoid subtle bugs. It is not supported in the current version of `jacc`.

If the action for a given production is omitted, then the generated parser behaves as if the action `{ $$ = $1; }` had been specified.

3.3 The Additional Code Section

The final section of the input to `jacc` follows the second `%%` marker, and provides code that will be copied into the body of the generated parser class. `jacc` does not attempt to check that this code is valid, so syntax errors in this portion of the code will not be detected until you attempt to compile the generated Java source files.

If you use `jacc` as a tool for exploring the properties of different grammars (which is something that you might do in the early stages of prototyping a new parser or language design), then you will probably not be interested in executing the parsers that `jacc` generates. In such cases, there is no need to include any additional code, and you can even omit the second `%%` marker. For example, the following shows the complete text of an input file that you could use with `jacc` to explore the 'dangling else' problem (this same text could be used without any changes as input to yacc):

```
%token IF THEN ELSE expr other
%%
stmt : IF expr THEN stmt ELSE stmt
     | IF expr THEN stmt
     | other
     ;
```

In practice, however, if you want to run the parsers that you obtain from `jacc`, then you will need to add at least the definition of a method:

```
void yyerror(String msg) {
   ... your code goes here ...
}
```

Every jacc-generated parser includes a reference to `yyerror()`, and will potentially invoke this method as a last resort if the parser encounters a serious error in the input from which it cannot recover. As such, a yacc-generated parser that does not include this method will not compile. The additional code section of a `jacc` input file is often a good place to provide a definition for `yyerror()` (although it could also be obtained by inheriting a definition from a superclass if you have also used the `%extends` directive).

It is also common practice to use the additional code section of an input file to provide: constructors for the parser class; local state variables (and accessor functions/getters) that record the results of parsing; definitions for helper functions that are used in the semantic actions; and a local variable, `lexer`, that provides a link to the lexical analyzer. In fact for simple cases—such as the example in Section 4.1—we might even include the full code for the lexer in the additional code section of the `.jacc` file.

4 Examples: `jacc` in practice

This section describes two example programs using `jacc`. Both are versions of a simple interactive calculator that reads sequences of expressions from the standard input (separated by semicolons) and display the results on the standard output. The first version (Section 4.1) is written as a single `jacc` source file, while the second (Section 4.2) shows how more realistic applications can be constructed from a combination of `jacc` and Java source files.

4.1 The Single Source File Version

This section describes a simple version of the calculator program in which all of the source code is placed in a single file called `simpleCalc.jacc`. The file begins with the following set of directives that specifies the name of the generated parser as `Calc`, defines a simple interface to lexical analysis, and lists the tokens that will be used:

```
%class     Calc
%interface CalcTokens
%semantic  int : yylval
%get       token
```

```
%next      yylex()

%token '+' '-' '*' '/' '(' ')' ';' INTEGER
%left  '+' '-'
%left  '*' '/'
%%
```

The rules section of `simpleCalc.jacc` gives the productions for the grammar, each of which is annotated with an appropriate semantic action.

```
prog : prog ';' expr  { System.out.println($3); }
     | expr           { System.out.println($1); }
     ;
expr : expr '+' expr  { $$ = $1 + $3; }
     | expr '-' expr  { $$ = $1 - $3; }
     | expr '*' expr  { $$ = $1 * $3; }
     | expr '/' expr  { $$ = $1 / $3; }
     | '(' expr ')'   { $$ = $2; }
     | INTEGER        { $$ = $1; }
     ;
%%
```

In this version of the program, we interleave evaluation of the input expression with parsing by using integers as `%semantic` values, and by executing the appropriate arithmetic operation as each different form of expression is recognized. This example illustrates very clearly how syntax (such as the symbolic token `'+'` in the production) is translated into semantics (in this case, the `+` operator on integer values) by the parsing process.

The additional code that is included in the final section of our input file is needed to turn the `jacc`-generated parser into a self-contained Java application. In a more realistic program, much of this functionality would be provided by other classes. However, every `jacc`-generated parser must include at least the definition of a `yyerror()` method that the parser will call to report a syntax error. In this example, we provide a very simple error handler that displays the error message and then terminates the application:

```
private void yyerror(String msg) {
  System.out.println("ERROR: " + msg);
  System.exit(1);
}
```

Next we describe a simple interface for reading source input, one character at a time, from the standard input stream. The variable `c` is used to store the most recently read character, and the `nextChar()` method is used to read the next character in the input stream, provided that the end of file (i.e., a negative character code in `c`) has not already been detected:

```
private int c;

/** Read a single input character from standard input.
 */
private void nextChar() {
  if (c>=0) {
    try {
      c = System.in.read();
    } catch (Exception e) {
      c = (-1);
    }
  }
}
```

The biggest section of code in our example program is used to implement a simple lexical analyzer. The lexer stores the code for the most recently read token in the `token` variable, and the corresponding integer value (for an `INTEGER` token) in the `yylval` variable. The lexer is implemented by the `yylex()` method. Note that this agrees with the settings specified by the `%get`, `%semantic`, and `%next` directives at the beginning of this example program.

```
int token;
int yylval;

/** Read the next token and return the
 *  corresponding integer code.
 */
int yylex() {
  for (;;) {
    // Skip whitespace
    while (c==' ' || c=='\n' || c=='\t' || c=='\r') {
      nextChar();
    }
```

```
    if (c<0) {
      return (token=ENDINPUT);
    }
    switch (c) {
      case '+' : nextChar();
                 return token='+';
      case '-' : nextChar();
                 return token='-';
      case '*' : nextChar();
                 return token='*';
      case '/' : nextChar();
                 return token='/';
      case '(' : nextChar();
                 return token='(';
      case ')' : nextChar();
                 return token=')';
      case ';' : nextChar();
                 return token=';';
      default  : if (Character.isDigit((char)c)) {
                   int n = 0;
                     do {
                       n = 10*n + (c - '0');
                       nextChar();
                     } while (Character.isDigit((char)c));
                     yylval = n;
                     return token=INTEGER;
                   } else {
                     yyerror("Illegal character "+c);
                     nextChar();
                   }
    }
  }
}
```

Notice that the lexer returns the symbol `ENDINPUT` at the end of the input stream. Every `jacc`-generated token interface defines this symbol, with integer value `0`. As in this example, the lexer should return this code to the parser when the end of the input stream is detected.

Last, but not least, we include a `main()` method that uses `nextChar()` to read the first character in the input stream, then `yylex()` to read the first

token, and then calls `parse()` to do the rest of the work:

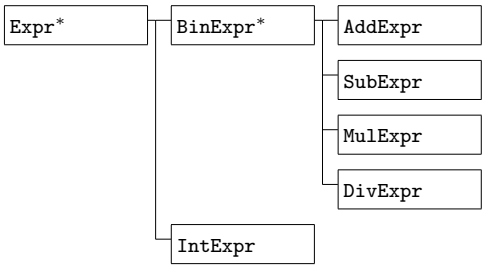
```
public static void main(String[] args) {
  Calc calc = new Calc();
  calc.nextChar(); // prime the character input stream
  calc.yylex();    // prime the token input stream
  calc.parse();    // parse the input
}
```

4.2 The Multiple Classes Version

This section presents a second version of our simple calculator program where the code is distributed across multiple classes, and in which the structure of the input expressions is captured explicitly in an intermediate data structure (representing the so-called *abstract syntax*). The resulting program is more representative of the way that parser generators like `jacc` are used in practice although, in this particular case, the program is still just a toy, and it would be hard to justify the extra overhead compared with the first version.

4.2.1 Abstract Syntax

Our first task is to define the classes that we need to capture the essential structure, or *abstract syntax*, of input expressions as concrete data values. For this, we choose a standard technique in Java programming with a hierarchy of classes, each of which represents a particular form of expression, and all of which are subclasses of an abstract base class `Expr`. The following diagram shows the inheritance relationships between the different classes in graphical form (abstract classes are marked by an asterisk):



The code that defines the classes in this small hierarchy is shown below. For an application as simple as our calculator program, this particular approach will likely seem unnecessarily complex and verbose—but it does at least scale to more realistic applications. Note that the only special functionality we build in to these classes is an ability to evaluate `Expr` values using the `eval()` method:

```
abstract class Expr {
  abstract int eval();
}

class IntExpr extends Expr {
  private int value;
  IntExpr(int value) { this.value = value; }
  int eval() { return value; }
}

abstract class BinExpr extends Expr {
  protected Expr left, right;
  BinExpr(Expr left, Expr right) {
    this.left = left;  this.right = right;
  }
}

class AddExpr extends BinExpr {
  AddExpr(Expr left, Expr right) { super(left, right); }
  int eval() { return left.eval() + right.eval(); }
}
class SubExpr extends BinExpr {
  SubExpr(Expr left, Expr right) { super(left, right); }
  int eval() { return left.eval() - right.eval(); }
}
class MulExpr extends BinExpr {
  MulExpr(Expr left, Expr right) { super(left, right); }
  int eval() { return left.eval() * right.eval(); }
}
class DivExpr extends BinExpr {
  DivExpr(Expr left, Expr right) { super(left, right); }
  int eval() { return left.eval() / right.eval(); }
}
```

4.2.2 Lexical Analysis

Our implementation of lexical analysis in this version of the calculator is a fairly simple modification of the corresponding code in the first version. We have wrapped the necessary code in a class called `CalcLexer`; declared that it should implement the token interface `CalcTokens`; and added methods `nextToken()` to read the next token, `getToken()` to retrieve the current token code, and `getSemantic()` to return the semantic value for the current token. (The latter being valid only if the current token is an integer literal.) These names have been chosen to coincide with the defaults that `jacc` assumes for the `%next`, `%get`, and `%semantic` directives.

```
class CalcLexer implements CalcTokens {
  private int c = ' ';

  /** Read a single input character from standard input.
   */
  private void nextChar() {
    if (c>=0) {
      try {
        c = System.in.read();
      } catch (Exception e) {
        c = (-1);
      }
    }
  }

  private int     token;
  private IntExpr yylval;

  /** Read the next token and return the
   *  corresponding integer code.
   */
  int nextToken() {
    for (;;) {
      while (c==' ' || c=='\n' || c=='\t' || c=='\r') {
        nextChar();      // Skip whitespace
      }
      if (c<0) {
        return (token=ENDINPUT);
```

```
      }
      switch (c) {
        case '+' : nextChar();
                   return token='+';
        case '-' : nextChar();
                   return token='-';
        case '*' : nextChar();
                   return token='*';
        case '/' : nextChar();
                   return token='/';
        case '(' : nextChar();
                   return token='(';
        case ')' : nextChar();
                   return token=')';
        case ';' : nextChar();
                   return token=';';
        default  : if (Character.isDigit((char)c)) {
                     int n = 0;
                       do {
                         n = 10*n + (c - '0');
                         nextChar();
                       } while (Character.isDigit((char)c));
                       yylval = new IntExpr(n);
                       return token=INTEGER;
                     } else {
                       Main.error("Illegal character "+c);
                       nextChar();
                     }
      }
    }
  }

  /** Return the token code for the current lexeme.
   */
  int getToken() { return token; }

  /** Return the semantic value for the current lexeme.
   */
  IntExpr getSemantic() { return yylval; }
}
```

Careful comparison of this code with the previous version will also reveal other small differences including the initialization of `c`—which avoids the need for a call to the lexer's `nextChar()` method, now hidden as a `private` method of `CalcLexer`—and a change in the type of semantic value, for reasons that will be explained in the next section.

4.2.3 Parsing

Because most of the functionality of the calculator has been moved out to other classes, our input to `jacc` is much shorter in this version. We assume that the following parser specification is placed in a file called `Calc.jacc` so that the generated parser and tokens interface will be given the (default) names `CalcParser` and `CalcTokens`, respectively.

```
%semantic Expr
%token '+' '-' '*' '/' '(' ')' ';' INTEGER
%left  '+' '-'
%left  '*' '/'
%%
prog : prog ';' expr    { System.out.println($3.eval()); }
     | expr             { System.out.println($1.eval()); }
     ;
expr : expr '+' expr    { $$ = new AddExpr($1, $3); }
     | expr '-' expr    { $$ = new SubExpr($1, $3); }
     | expr '*' expr    { $$ = new MulExpr($1, $3); }
     | expr '/' expr    { $$ = new DivExpr($1, $3); }
     | '(' expr ')'     { $$ = $2; }
     | INTEGER          { $$ = $1; }
     ;
%%
  private CalcLexer lexer;
  CalcParser(CalcLexer lexer) { this.lexer = lexer; }

  private void yyerror(String msg) { Main.error(msg); }
```

Notice that this version of the parser uses the constructors for the `AddExpr`, `SubExpr`, `MulExpr`, and `DivExpr` classes to build a data structure that describes the structure of the expression that is read. There are no calls to the constructor for `IntExpr` here because the lexer takes care of packaging up

the semantic values for integer literals as `IntExpr` objects. This is important because it means that they can be used as semantic values in the parser (note that we have declared `Expr` as the `%semantic` type for this parser, and that `IntExpr` is a subclass of `Expr`.)

One alternative would have been to use the default semantic type `Object`; to arrange for the lexer to return the value of each integer constant as an `Integer` object; and to have the parser handle the construction of `IntExpr` values by changing the action for the last production in the grammar to:

```
{ $$ = new IntExpr($1.intValue()); }
```

Such a change would also require us to declare types for `INTEGER` and `expr` (or else to rewrite the semantic actions to include explicit casts):

```
%token <Integer> INTEGER
%type  <Expr>
```

4.2.4 Top-level Driver

To complete this second version of the calculator program we need a small driver that constructs and initializes a `lexer` object, uses that to build a suitable parser object, and then invokes the parser's main `parse()` method.

```
class Main {
  public static void main(String[] args) {
    CalcLexer  lexer  = new CalcLexer();
    lexer.nextToken();
    CalcParser parser = new CalcParser(lexer);
    parser.parse();
  }

  static void error(String msg) {
    System.out.println("ERROR: " + msg);
    System.exit(1);
  }
}
```

Note also that we have had to introduce an `error()` method that can be shared between the parser and the lexer. This sharing of services—in this

case, for error handling—is common in any system where the same functionality is required in components that are logically distinct.

References

[1] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast yacc-compatible parsers (for very little effort). *Software—Practice & Experience*, 28(2):181–190, February 1998.

[2] Free Software Foundation. Bison. (`http://www.gnu.org/manual/bison-1.35/bison.html`).

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, June 2000.

[4] Scott E. Hudson. The CUP parser generator for Java. (`http://www.cs.princeton.edu/~appel/modern/java/CUP/`).

[5] Bob Jamison. BYACC/Java. (`http://troi.lincom-asg.com/~rjamison/byacc/`).

[6] S.C. Johnson. Yacc — yet another compiler compiler. Technical Report Computer Science Technical Report Number 32, Bell Laboratories, July 1975.

[7] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition.* O'Reilly, 1992.

[8] Sun Microsystems. Java compiler compiler (JavaCC)—the Java parser generator. (`http://www.webgain.com/products/java_cc/`).

[9] H. Mössenböck. Coco/R for Java. (`http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Java/`).

[10] Terrence Parr. Antlr, another tool for language recognition. (`http://www.antlr.org/`).

[11] David Shields and Philippe G. Charles. The Jikes parser generator. (`http://www-124.ibm.com/developerworks/projects/jikes/`).

[12] The Sable Research Group. SableCC. (`http://www.sablecc.org/`).