The Java Parsing System

Pouria Shaker Theodore S. Norvell

Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland

pouria@engr.mun.ca theo@engr.mun.ca

October 12, 2004

Abstract

The Java Parsing System (JPS) is a tool that works in conjunction with the JavaCC parser generator to generate recursive-decent parsers written in Java, for languages specified by extended context free grammars. Generated parsers communicate parser-events to client applications through a standard API. JPS currently supports four modes of operation: two such modes make it backward compatible with its predecessor JJTree; a third mode generates SAX conformant parsers; and a final mode gives client applications full control over their dealings with parse events. Unlike parser generators such as yacc and JavaCC, JPS provides a clean separation of concerns between grammar writing and the processing of the input language.

1 Introduction

Processing input data is a component of many software problems. The processing of data only becomes meaningful when the receiving and sending sides have agreed upon a structured way of communication. If data conforms to the agreed structure, the receiving end may extract and use the information present in the data. To r*ecognize* data is to determine whether input data conforms to the agreed structure; to *parse* data is to recognize data and produce output based on information present in the data.

Despite the existence of well-known techniques for writing parsers, the process is mechanical, tedious, and error-prone. For this reason, tools have been developed to automatically generate parsers from formal descriptions of rules governing the syntax of allowable data sequences. One such tool is JavaCC; JavaCC generates parsers written in Java. The default parser generated by JavaCC is a mere recognizer; to make the parser produce output, the programmer must augment the syntax rules with additional instructions in the form of Java code.

The Java Parsing System (JPS) is an extension to JavaCC that allows programmers to specify instructions related to the parser output separately from the syntax rules; this allows for grammar re-use.

2 JavaCC [1]

JavaCC is a parser-generator; given a context free grammar, it generates a recursive-decent parser written in Java. A recursive descent parser defines a subroutine for each nonterminal in the grammar. Each subroutine is responsible for recognizing a sequence of terminal symbols produced by its nonterminal, and for removing that sequence of terminal symbols from the input sequence. One can augment the subroutines with additional code to define a parser that produces some output in the event that the input is recognized.

A grammar specification for JavaCC consists of a set of lexical analysis rules mapping character strings to named tokens, and a set of productions defined as a set of methods as explained above.

Consider an extended context free grammar G defined by the following elements: The set of terminals {a, b, c}, the set of non-terminals {S, A} with S designated as the starting non-terminal, and the following set of productions P:

S → a A
A → (a | b) A
A → c

Figure 1 is a JavaCC specification of the lexical rules of this grammar while Figure 2 shows a pair of JavaCC production definitions for non-terminals S and A; note the Java code augmentations in bold. Admittedly, the output of the parser of Figure 2 is not very interesting. It prints a trace of the parsing process by reporting the consumption of each terminal.

```
TOKEN :
{
        < A: "a" > |
        < B: "b" > |
        < C: "c" >
}
```

Figure 1: JavaCC Token Declaration Example

```
void S()
{
    Token t;
}
{
    t = <A>
    A()
    {
        System.out.println("S found" + t.image);
    }
}
```

```
void A()
{
    Token t;
}
{
  (  ( t = <A> | t = <B> ) A() )
    |
  (  t = <C> )
    {
       System.out.println("A found" +  t.image);
    }
 }
```

Figure 2: JavaCC Production Declaration Example

The input to JavaCC is stored in a text file with the extension *jj*. Figure 3 illustrates JavaCC operation at the system level.

Figure 3: JavaCC System Level Operation

3 Why JPS?

JavaCC code augmentation is a powerful feature. It allows users to configure parsers to interact with other classes in their applications throughout the parse process. But there are two difficulties associated with this scheme:

1. Augmenting the grammar with Java code often clutters the grammar, reducing its readability and maintainability
2. Mode importantly, generating two parsers for the same grammar that produce different outputs would require the duplication of the entire grammar, with modifications made only to the code augmentations

The ideal solution to this problem is to separate the code from the grammar: JPS makes this possible. Figure 4 illustrates JPS operation at the system level.

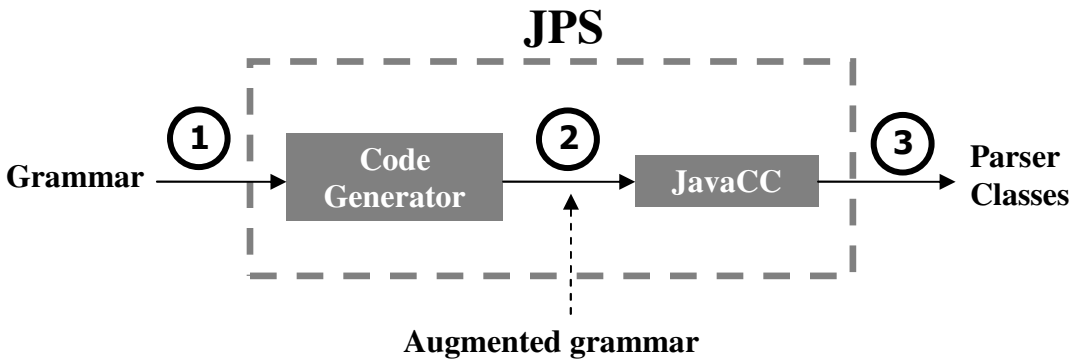

Figure 4: JPS System Level Operation

From a user's point of view, JPS operates the same as JavaCC; the input is a grammar specification and the output is set of parser classes. The difference lies in the fact that the generated parser is now guaranteed to raise parse-events at well-defined points of the input. Client applications may now use different event-handlers to produce different output from the same parser.

Internally JPS augments the grammar with Java code to produce call-backs to a *builder* object. The builder object is a member of a class implementing the `Builder` interface; think of the call-backs as the raised parse-events and of the `Builder` implementation as the event-handler. Once again, changing the output of the parser is simply a matter of registering a new builder implementation with the parser (Figure 5).

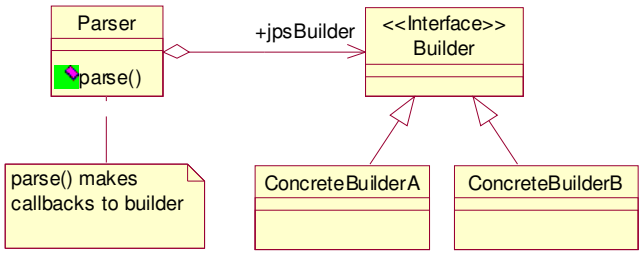

Figure 5: A JPS Generated Parser's Basic Class Relationships

Figure 6 shows a sample production going through the three transitions of Figure 4. Call-backs to the builder have been underlined in the code sample for transition 2.

```
void Start() #Start : {}
{
    Expression() ";"
}
```

1- Original Production

```
void Start() : {
  Token jpst000;
  boolean jpsc000 = true;
  jpsBuilder.startScope(JJTSTART, true);
}
{
  try {
    Expression() jpst000 = ";" { jpsBuilder.token(jpst000); }
  }
  catch(Exception jpse000) {
    jpsBuilder.abortScope(JJTSTART, jpse000, jpsc000);
    jpsc000 = false;
  }
  finally {
    if(jpsc000) {
      jpsBuilder.endScope(JJTSTART);
    }
  }
}
```

2- Augmented Production

```
static final public void Start() throws ParseException {
  Token jpst000;
  boolean jpsc000 = true;
  jpsBuilder.startScope(JJTSTART, true);
  try {
    Expression();
    jpst000 = jj_consume_token(14);
    jpsBuilder.token(jpst000);
  }
  catch (Exception jpse000) {
    jpsBuilder.abortScope(JJTSTART, jpse000, jpsc000);
    jpsc000 = false;
  }
  finally {
    if (jpsc000) {
      jpsBuilder.endScope(JJTSTART);
    }
  }
}
```

3- Generated Parser

Figure 6: Parser Generation Phases

4 The Builder Interface

The definition of the `Builder` interface is shown Figure 7. The reader is encouraged to refer to Figure 6 while reading descriptions of the methods in the `Builder` interface.

```
public interface Builder {
  /** callback for start of scope */
  public void startScope(int aKind, boolean aIsUnconditional);

  /** callback for end of scope for unconditional scopes */
  public void endScope(int aKind);

  /** callback for end of scope for indefinite scopes */
  public void endScope(int aKind, boolean aCondition);

  /** callback for end of scope for definite scopes */
  public void endScope(int aKind, int aNumChildren);

  /** callback for aborting a scope */
  public void abortScope(int aKind, Exception aException,
                         boolean aScopeOpen);

  /** callback for token encounter */
  public void token(Token aToken);

  /** callback for querying arity */
  public int arity();

  /** callback for obtaining reference to object representing
      scope (using jjthis in custom, simple, and multi modes
      produces this callback */
  public Object getJJTThis(boolean aScopeOpen);

  /** callback for reinitializing builder */
  public void reset();
}
```

Figure 7: The Builder Interface

4 JPS Modes of Operation

JPS currently supports four modes of operation:

- *Simple* mode and *multi* mode are provided for backward compatibility with JJTree. JJTree is a pre-processor for JavaCC; it augments a JavaCC language specification with code for parse-tree generation. JJTree provides users with some control over the tree-generation process by introducing additional annotations to the grammar specification [2]. In simple mode and multi mode, JPS provides an implementation of the builder interface that generates parse-trees in a manner identical to JJTree; it also generates the same supporting files as JJTree.

- In *SAX* mode, JPS generates SAX conformant parsers. SAX, or the Simple API for XML, is a set of interfaces designed for communication between XML parsers and client applications. SAX is widely used and is supported in several programming languages including Java. A SAX conformant parser is one that implements `XMLReader`, an interface that allows an application to register event handlers for document processing, and initiating a document parse [3]. When set to operate in SAX mode, JPS modifies the parser class definition such that the parser implements `XMLReader`. In this mode, the builder is implemented to produce SAX calls. This allows for SAX-based tools to use input formats other than XML; in other words, the JPS generated parser poses as an XML parser to the SAX-based tool.
- In *custom* mode, the user provides the implementation of the builder interface. This allows users to handle parse events any way they want.

JPS is extensible; its strategy based design allows for a straight-forward implementation of new built-in modes of operation. One possibility for the future is the development of a mode for top-down tree construction (JJTree builds trees bottom up).

8 Potential Applications of JPS

The following are few interesting applications for JPS:

- *Text Converters*: given a grammar for a particular text format, it is possible to generate a parser for that text format using JPS, and to implement several builders that perform conversions to other text formats.
- *Code Analysis Tools*: it is possible to implement builders for performing code analysis operations such as software metrics measuring, anomaly checking, style checking, verification, indexing, etc.
- *Code Transformation Tools*: it is possible to implement builders for performing code transformation operations such as pretty printing, language augmentation (e.g. embedded SQL), parallelization, etc.
- *Front-end of Compilers*: the first step in the compilation process is to parse the input, and to generate a parse-tree. It is possible to use JPS to perform this step.
- *Special Purpose Languages*: JPS can be used in an application to provide support for support special purpose languages and formats such as scripting languages, configuration files, data files, email and news messages, address lists, etc.

9 Conclusion

A growing collection of useful grammars developed by JavaCC users is currently available; the re-usability of such grammars is limited by custom code augmentations; JPS alleviates this problem by separating grammar augmentation from grammar specification.

References

[1] Java Compiler Compiler [tm] - The Java Parser Generator, [online] https://javacc.dev.java.net/

[2] JavaCC [tm] - JJTree Reference Documentation, [online] https://javacc.dev.java.net/doc/JJTree.html

[3] About SAX, [online] http://www.saxproject.org/