Instantiating JavaCC Tokenizers/Parsers to Read from Unicode Source Files

Kenneth R. Beesley
Xerox Research Centre Europe
6, chemin de Maupertuis
38240 MEYLAN, France
ken.beesley@xrce.xerox.com

20 February 2005
Modified 21 April 2005
Modified 14 October 2005

Abstract

In this paper[1] I explain how to instantiate a JavaCC parser so that it reads from Unicode sources or sources in almost any industry-standard encoding.

This document reflects my own best understanding of and experience with Unicode processing in JavaCC (currently version 4.0beta1). I wrote the paper for my own future reference and in the hope that it would help other users.

Clarifications and corrections would be most welcome.

1 Introduction

1.1 JavaCC and Unicode

JavaCC is a popular parser-generator used to implement parsers for programming languages.[2]

In 2005, Unicode is a practical reality, and Unicode-capable text editors and Graphical User Interfaces (GUIs) are available on all popular platforms. This makes it possible to define new programming languages that contain Unicode strings or even Unicode identifiers and operators. Thus some JavaCC parsers need to read from Unicode files and other Unicode sources.

This paper explains how to instantiate a JavaCC parser to read from a source (file or whatever) that is not necessarily in the default character encoding of your operating system. In particular, I show how to make your parser read from Unicode files in the popular UTF-8 encoding.

[1] Source in my ../langcomp/rx/doc/papers/javacc_unicode/. XRCE Publication Release signed by Graham Button 6 December 2004.

[2] The home page for JavaCC is `https://javacc.dev.java.net`. A very useful FAQ can be found at `http://www2.engr.mun.ca/~theo/javaCC-FAQ/javacc-faq.htm`.

1.2 Beware the Existing JavaCC Documentation

While I'm very glad that JavaCC is available, and I'm grateful to the people who produced and maintain it, the state of the JavaCC documentation is lamentable. (In the JavaCC download, the documentation is found in the ../doc/ directory.) Beware of the following outdated documentation in the JavaCC 3.2 and JavaCC 4.0 distributions that can all too easily confuse the innocent:

- The entire `../doc/CharStream.html` file in the JavaCC download is completely obsolete and should be ignored. I have urged that this file be removed from the distribution, but without success.
- The JavaCC option `UNICODE_INPUT` appears, at least after my initial tests, to be unused and obsolete in JavaCC 4.0. (Corrections would be welcome.)
- The out-of-date documentation files

  ```
  ../doc/apiroutines.html
  ../doc/tokenmanager.html
  ```

  refer to the four stream classes

  ```
  ASCII_CharStream
  ASCII_UCodeESC_CharStream
  UCode_CharStream
  UCode_UCodeESC_CharStream
  ```

  which have also been obsolete since JavaCC 2.1.[3] Ignore all references to these obsolete stream classes; they should have been edited out of the documentation long ago. The current automatically generated Unicode-savvy "char-stream" classes, described below, are

  ```
  SimpleCharStream
  JavaCharStream
  ```

1.3 General Mind-Tuning about JavaCC and Unicode

- Java chars are Unicode characters, and Java String objects consist of Unicode characters. Inside Java programs, all text, Strings, and characters are Unicode.
- The tokenizer and parser generated by JavaCC from your JavaCC source files are Java programs.
- JavaCC tokenizer specifications are written in terms of Unicode characters. Text read from an input file, e.g. a source file representing a program in your new language, is converted to Unicode, *one way or another*, before it gets to your tokenizer.
- An InputStream is a Java object that is a source of raw bytes. InputStream is an abstract class, implemented by the concrete classes FileInputStream (used in the present examples), PipedInputStream, StringBufferInputStream, etc. System.in is a built-in static Java InputStream, i.e. it's also a source of raw bytes. Wherever FileInputStream appears in the examples below, you could substitute other implementations of the InputStream interface as appropriate for your application.

[3] See `../doc/javaccreleasenotes.html`.

- A Reader is a Java object that is a source of Unicode characters. Reader is an abstract class, implemented by the concrete classes InputStreamReader (used in the present examples), BufferedReader, StringReader, FileReader, etc. Readers are Unicode-savvy and know how to convert from a large set of industry-standard encodings to Java's internal Unicode characters. Wherever InputStreamReader appears in the examples below, you could substitute other implementations of the Reader interface as appropriate for your application.

- SimpleCharStream and JavaCharStream are classes automatically generated by JavaCC that wrap a Reader and provide the bridge between a stream of Unicode characters (coming from that Reader) and your XXXTokenManager. The XXXTokenManager calls the SimpleCharStream or JavaCharStream every time it needs the next Unicode character. The XXXTokenManager maps a stream of characters into a stream of tokens, according to the tokenizer definitions in your XXX.jj or XXX.jjt file. (More about all this below.)

- Finally, your XXX syntactic parser calls the XXXTokenManager whenever it needs the next token.

This paper is dedicated to explaining how JavaCC parsers can be instantiated so that they read from source files in various encodings, especially UTF-8, so that the characters are properly converted to Unicode before they are seen by the tokenizer.

2 JavaCC Options and Unicode

In the examples that follow, let's assume that your new language is named XXX and so is defined in a JavaCC source file named XXX.jj; if you are using JJTree as well, then your source file will be XXX.jjt.

The `options` specified at the top of your JavaCC source file affect the generation of your JavaCC parser in many ways. The overall syntax is

```
options {
   option_name = option_value ;
   option_name = option_value ;
   ...
}
```

and most of the `options` have built-in default values that are appropriate for most users. They also have some potentially confusing interdependencies. See `../doc/javaccgrm.html`, with caution, for general documentation of these `options`. The `options` most important for the understanding of the handling of Unicode are the following:

2.1 USER_TOKEN_MANAGER

The default value of USER_TOKEN_MANAGER is false, which is what most users want. When

```
USER_TOKEN_MANAGER = false ;  // default value is false
```

JavaCC will automatically generate a token manager class definition for you. If your language is named XXX, i.e. if your source file is named XXX.jj or XXX.jjt, then the

JavaCC compiler generates a file named XXXTokenManager.java. This, I repeat, is what most users want.

Mind Tuning: You typically define the tokenizer using SKIP, TOKEN, MORE and SPECIAL_TOKEN declarations in your XXX.jj or XXX.jjt source file. The automatically generated XXXTokenManager.java is based on these declarations and contains methods that "manage" tokens. An XXXTokenManager object maps from a stream of input characters, i.e. Unicode characters, to a stream of tokens, according to your declarations, and maintains a queue of available tokens to send to the syntactic parser.[4] The parser calls the XXXTokenManager whenever it needs the next token.

For experts only: If you set

```
USER_TOKEN_MANAGER = true ;   // for experts only
```

then only an *interface* named TokenManager.java is generated, rather than the ready-to-use XXXTokenManager.java; and then you, the "USER", have to hand-write your own class that implements this TokenManager.java interface; you probably don't want to try that unless you're an expert. In the rest of this document I assume that the value of USER_TOKEN_MANAGER is left as false, the default value.

Beware: The class defined in the automatically generated XXXTokenManager.java file does not, and is not supposed to, implement the TokenManager.java interface. The TokenManager.java interface is only for writing hand-crafted, user-defined token managers, and that's something best left to the experts.

2.2 USER_CHAR_STREAM

The default value of USER_CHAR_STREAM is false, which is what most users want. When

```
USER_CHAR_STREAM = false ;  // default value is false
```

JavaCC will automatically generate either SimpleCharStream.java or JavaCharStream.java, depending on the setting of `JAVA_UNICODE_ESCAPE` (see below).

For experts only: If you set

```
options {
  USER_CHAR_STREAM = true ;    // default is false
  // JAVA_UNICODE_ESCAPE  is ignored
  ...
}
```

then neither SimpleCharStream.java nor JavaCharStream.java is generated, the setting of option JAVA_UNICODE_ESCAPE is ignored, and only an *interface* named CharStream.java is generated. Then you, the "USER", have to hand-write your own user-defined class that implements this interface. You probably don't want to try that unless you're an expert. The rest of this paper assumes that the option USER_CHAR_STREAM is left as false, and that we want to use one of the two automatically generated classes: SimpleCharStream or JavaCharStream. (The choice between them is explained in the next section.)

[4] In most cases, the tokenizer should be thought of as working independently of the syntactic parser, and may well have tokenized ahead of where the parser is, keeping available tokens on the queue until the parser calls for them.

Mind Tuning: Your XXXTokenManager object gets a stream of Unicode characters from a char-stream object, which may be one of three separate types: SimpleCharStream, JavaCharStream, or a hand-crafted, user-defined class (e.g. BobsCharStream or CarolsCharStream) that implements the CharStream.java interface.

Beware: Note that the automatically generated SimpleCharStream.java and JavaCharStream.java do not, and are not supposed to, implement the CharStream.java interface.[5]

2.3 JAVA_UNICODE_ESCAPE

The JAVA_UNICODE_ESCAPE option is false by default, but you may want to set it to true.

Your XXXTokenManager object gets a stream of Unicode characters from a char-stream object, which in turn gets a stream of Unicode characters from a Reader object. The stream of Unicode characters sent by the Reader object to the char-stream object may contain 6-char sequences of the form \uHHHH, where H is a hex character. For example, the Arabic taa' character happens to have the Unicode code point value 0x062A and can be represented in a Java program as \u062A. Such "Java escape sequences" are the Java-language convention for designating Unicode characters where typing the actual Unicode character is either inconvenient or impossible (e.g. if your editor is limited to ASCII). Java compilers automatically detect 6-char sequences like \u062A in a Java source file and collapse them down to single Unicode characters.

If you want to steal the Java-escape convention for use in your own new language, then specify

```
options {
    JAVA_UNICODE_ESCAPE = true ;  // the default value is false
}
```

and such 6-char sequences will be intercepted and collapsed to a single Unicode character before being passed to your XXXTokenManager object. The effect is best shown in the real examples that follow.

There is nothing sacred about the Java-language escape convention \uHHHH, which is also used in Python. In Perl, for example, the equivalent escape convention is \x{HHHH}. JavaCC makes it easy, via the `JAVA_UNICODE_ESCAPE = true` option, to borrow/steal the Java convention for use in your own language. If you want to use a different convention, e.g. Perl's \x{HHHH}, then leave `JAVA_UNICODE_ESCAPE` as false, and then the individual characters will be passed uncollapsed to your XXXTokenManager, which will need to tokenize them explicitly.

3 Unicode and JavaCC 4.0

3.1 JavaCC and SimpleCharStream.java

Let's assume that you want your parser to read from a Unicode source file that is in the UTF-8 encoding. In JavaCC, the key option that you need to understand is `JAVA_UNICODE_ESCAPE`, which is set to false by default.

[5] Here be dragons. Note that the automatically generated SimpleCharStream.java and JavaCharStream.java files contain a comment at the top which claims that "[this class is] An implementation of interface CharStream where the stream is assumed to contain only ASCII characters". This comment is false and misleading: the files are *not* in fact implementations of the CharStream interface, and they are *not* limited to ASCII characters. Just ignore these comments.

```
options {
  USER_TOKEN_MANAGER = false ;  // default is false
  USER_CHAR_STREAM = false ;    // default is false
  JAVA_UNICODE_ESCAPE = false ; // default is false
                                //   false value causes the file
                                //   SimpleCharStream.java
                                //   to be generated instead of
                                //   JavaCharStream.java
  ...
}
```

When JAVA_UNICODE_ESCAPE is set to false (the default value), JavaCC automatically generates the file SimpleCharStream.java (rather than JavaCharStream.java), and a SimpleCharStream object will *not* intercept 6-char sequences of the form \uHHHH, where H is a hex digit, in the input stream and collapse them to one Unicode character; rather it will leave them as 6-char sequences and pass them unchanged to your XXXTokenManager.

N.B. the generated file SimpleCharStream.java still contains the following erroneous comment in the header:

```
/* Generated By:JavaCC: Do not edit this line.
   SimpleCharStream.java Version 4.0 */
/**
 * An implementation of interface CharStream,
 * where the stream is assumed to
 * contain only ASCII characters (without unicode processing).
 */
```

In reality, this automatically generated SimpleCharStream.java does *not* implement the CharStream interface (and it's not supposed to),[6] and the stream is *not* limited to only ASCII characters. Just ignore this confusing comment.

JavaCC 4.0 offers a new parser constructor (not available in 3.2) that takes as arguments a standard Java InputStream object and a String representing the encoding. For example, if your language is named XXX, to cause your tokenizer/parser to read from a file named `input.utf8` that is in the UTF-8 encoding, you could instantiate the parser in the following way.

```
  XXX parser = null ;
  try {
    parser = new XXX(new FileInputStream("input.utf8"), "UTF-8") ;
  }
  catch (FileNotFoundException e) {
    System.out.println("File not found. Exiting.") ;
    System.exit(0) ;
  }
```

When JAVA_UNICODE_ESCAPE is set/left at false, this new constructor call is apparently equivalent to the following more verbose constructor call (already available in JavaCC 3.2), which explicitly uses the generated XXXTokenManager and SimpleCharStream classes, and a standard Java InputStreamReader:

[6] The interface CharStream.java is generated only when the option USER_CHAR_STREAM is set to true (the default value is false, which is what most users want). If this option is set to true, then the user must hand-write a char-stream class that implements the CharStream.java interface; and this is best left for the experts.

```
  XXX parser = null ;
  try {
    parser = new XXX(new XXXTokenManager
                     (new SimpleCharStream
                       (new InputStreamReader
                         (new FileInputStream("input.utf8"), "UTF-8")
                       )
                     )
                    ) ;
  }
  catch (UnsupportedEncodingException e) {
    System.out.println("Encoding unknown. Exiting.") ;
    System.exit(0) ;
  }
  catch (FileNotFoundException e) {
    System.out.println("File not found. Exiting.") ;
    System.exit(0) ;
  }
```

Either way, the net effect is to open the file `input.utf8`, interpret the stream of bytes as UTF-8 and convert them accordingly to Java Unicode characters, and then pass the characters to the XXXTokenManager, which in turn supplies tokens to the parser.

Note that in JavaCC 4.0 (as in JavaCC 3.2), you *cannot* instantiate a JavaCC parser in the following way:

```
// DANGER: This does NOT work in JavaCC 4.0 (or 3.2)
XXX parser = new XXX(new SimpleCharStream
                       (new InputStreamReader
                          (  new FileInputStream("inputfile"),
                             "UTF-8"
                          )
                       )
                     ) ;
// DANGER: This does NOT work in JavaCC 4.0 (or 3.2)
```

The code just above does not work because there is no constructor that allows a JavaCC parser to be implemented directly on a SimpleCharStream or JavaCharStream object. (Remember that SimpleCharStream.java and JavaCharStream.java do *not* implement the CharStream.java interface.)

Corrections and comments would be appreciated.

3.2 JavaCC 4.0 and JavaCharStream.java

Now let's assume that you want your parser to read from a Unicode source file that is in the UTF-8 encoding, and that you do want 6-char sequences of the form \uHHHH, where H is a hex digit, in the input stream to be intercepted and collapsed by the char-stream object into one Unicode character before they are passed the XXXTokenManager. This collapsing trick is performed by a JavaCharStream object.

To cause the vital file JavaCharStream.java to be generated, you need to specify the option `JAVA_UNICODE_ESCAPE = true`.

```
options {
  USER_TOKEN_MANAGER = false ;  // default is false
```

```
  USER_CHAR_STREAM = false ;    // default is false
  JAVA_UNICODE_ESCAPE = true ;  // default is false
                                //   true value causes the file
                                //   JavaCharStream.java
                                //   to be generated instead of
                                //   SimpleCharStream.java
  ...
}
```

When JAVA_UNICODE_ESCAPE is set to true (the default value is false), JavaCC automatically generates the file JavaCharStream.java rather than SimpleCharStream.java.

N.B. the generated file JavaCharStream.java still contains the following erroneous comment in the header:

```
/* Generated By:JavaCC: Do not edit this line.
   JavaCharStream.java Version 4.0 */
/**
 * An implementation of interface CharStream,
 * where the stream is assumed to
 * contain only ASCII characters (without unicode processing).
 */
```

In reality, this JavaCharStream.java does *not* implement the CharStream interface (and it's not supposed to), and the stream is *not* limited to only ASCII characters. Just ignore this confusing comment.

As already shown in the previous section, JavaCC 4.0 offers a new parser constructor (not available in 3.2) that takes as arguments an InputStream object and a String representing the encoding. For example, if your language is named XXX, to cause your tokenizer/parser to read from a file named input.utf8 that is in the UTF-8 encoding, you could instantiate the parser in the following way.

```
  XXX parser = null ;
  try {
    parser = new XXX(new FileInputStream("input.utf8"), "UTF-8") ;
  }
  catch (FileNotFoundException e) {
    System.out.println("File not found. Exiting.") ;
    System.exit(0) ;
  }
```

When JAVA_UNICODE_ESCAPE is set to true, this new constructor call is apparently equivalent to the following more verbose constructor call (already available in JavaCC 3.2), which explicitly uses the generated XXXTokenManager and JavaCharStream classes, and a standard Java InputStreamReader:

```
  XXX parser = null ;
    try {
      parser = new XXX(new XXXTokenManager
                       (new JavaCharStream
                         (new InputStreamReader
                           (new FileInputStream("input.utf8"), "UTF-8")
                         )
                       )
                     ) ;
```

```
}
catch (UnsupportedEncodingException e) {
  System.out.println("Encoding unknown. Exiting.") ;
  System.exit(0) ;
}
catch (FileNotFoundException e) {
  System.out.println("File not found. Exiting.") ;
  System.exit(0) ;
}
```

Either way, the net effect is to open the file `input.utf8`, interpret the stream of bytes as UTF-8 and convert them accordingly to Java Unicode characters, intercept and collapse any 6-char sequences of the form \uHHHH into single characters, and then pass the characters to the XXXTokenManager, which in turn supplies tokens to the parser.

Corrections and comments would be appreciated.

4 Input from Files in Other Industry-Standard Encodings

Your input files might in fact be in any of a large number of industry-standard encodings, e.g. UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, ISO-8859-1, ISO-8859-2, ISO-8859-6, ISO-8859-1-WINDOWS-3.1-LATIN-1, etc., etc., etc.[7] To instantiate a JavaCC parser that reads its input from a file encoded in ISO-8859-1 (also known as "latin1"), just replace the UTF-8 name in the above examples with the encoding name ISO-8859-1:

```
// read input from an ISO-5589-1-encoded file
XXX parser = new XXX(new XXXTokenManager
                        (new JavaCharStream  // or SimpleCharStream
                            (new InputStreamReader
                                (  new FileInputStream ("inputfile"),
                                   "ISO-8859-1"
                                )
                            )
                        )
                    ) ;
```

In JavaCC 4.0, that can be simplified greatly to

```
// read input from an ISO-5589-1-encoded file
XXX parser = new XXX(new FileInputStream ("inputfile"), "ISO-8859-1") ;
```

5 Using SimpleCharStream or JavaCharStream to Read from Files in the Default Encoding of Your Operating System

Recall that when Java programs (including parsers generated by JavaCC) read text from a file, that text always gets converted, *one way or another*, into Unicode characters

[7] For a long list of standard encoding names, and some recognized aliases, see `http://java.sun.com/j2se/1.4.2/docs/api/`.

inside Java. Some Java programmers remain blissfully unaware of this conversion because Java programs always "know" the default encoding of the operating system; unless explicitly instructed otherwise, they will assume that an input file is in the default encoding and will automatically convert it to Unicode characters accordingly.

Similarly, when a Java program writes a Unicode String out to file, it will assume, unless explicitly told otherwise, that it should convert the Unicode characters into the default encoding for output.

Assume again that your language is named XXX. If you have set your `options` so that the file SimpleCharStream.java is automatically generated, then you can instantiate an XXX parser with the following incantation:

```
options {
  USER_TOKEN_MANAGER = false ;   // this is the default value
  USER_CHAR_STREAM = false ;     // this is the default value
  JAVA_UNICODE_ESCAPE = false ;  // causes SimpleCharStream.java
                                 //   to be generated rather than
                                 //   JavaCharStream.java
}
...

// instantiating an XXX parser to read from a file in
// the default encoding of the operating system, whatever
// that might be

XXX parser = new XXX(new XXXTokenManager
                         (new SimpleCharStream
                              (new FileReader("inputfile")))) ;
```

or, equivalently

```
XXX parser = new XXX(new XXXTokenManager
                         (new SimpleCharStream
                              (new FileReader
                                   (new File("inputfile"))))));
```

In these and subsequent examples, the `"inputfile"` should be replaced by the name of the source file you want to read from or a String variable set to that name.

In these cases, the Reader object (here a FileReader) assumes that the input file is encoded in the default encoding of the operating system, whatever that might be, and converts the stream of bytes coming from that file from the default encoding into Unicode characters. The SimpleCharStream object wraps the FileReader and is the bridge between the Reader and the XXXTokenManager. Your token manager will call the SimpleCharStream object each time that it needs the next Unicode character.

The default Java conversion from and into the default encoding of the host operating system is generally a Good Thing, allowing programs to be more portable. To find the default encoding of your operating system, copy the following trivial Java program to

a file named FindDefaultEncoding.java.

```
public class FindDefaultEncoding {
    public static void main(String[] args) {
        String s = System.getProperty("file.encoding") ;
        System.out.println(s) ;
    }
}
```

Then compile it (the dollar sign represents your command-line prompt):

```
$ javac FindDefaultEncoding.java
```

which should create the file FindDefaultEncoding.class. Then execute the class file thus

```
$ java FindDefaultEncoding
```

and the default encoding should be displayed.

It is important to understand that regardless of the default encoding of your operating system, inside any Java program, including the Java-language parsers generated from your JavaCC definitions, all characters and Strings are Unicode,[8] so any text input to your parser is going to be converted to Unicode characters, *one way or another*, before your tokenizer sees it. In the examples above using FileReader, the FileReader "knows" what the default character encoding of your operating system is, and it converts the stream of raw bytes coming from the input file from that system-default encoding (whatever it might be) to Unicode characters. If in fact that input file is not in the default encoding (e.g. if your system encoding is CP437, and the file is really stored as UTF-8), then disaster is likely to follow. Sections 3 and 4 show you how to instantiate a JavaCC parser to read from files in encodings that are not the system-default encoding.

Use the following incantation when you want to read from a file in the default encoding, and you want a JavaCharStream to collapse Java-unicode-escape-sequences of the form \uHHHH to single Unicode characters:

```
options {
  USER_TOKEN_MANAGER = false ;  // default is false
  USER_CHAR_STREAM = false ;    // default is false
  JAVA_UNICODE_ESCAPE = true ;  // default is false
                                //   true cause JavaCharStream.java
                                //   to be generated instead of
                                //   SimpleCharStream.java
}
...

XXX parser = new XXX(new XXXTokenManager
                       (new JavaCharStream
                          (new FileReader("inputfile")))) ;
```

or, equivalently

[8] In Java 1.4, each Unicode character is stored internally as a 16-bit `char`. This Unicode encoding, known as UCS-2, cannot handle supplemental Unicode characters, the characters that go beyond the Basic Multilingual Plane. Java 1.5 handles supplemental characters in an encoding known as UTF-16.

```
XXX parser = new XXX(new XXXTokenManager
                       (new JavaCharStream
                          (new FileReader
                             (new File("inputfile")))));
```

Again, the result is to open the indicated file, assume that it is in the default encoding of the operating system, perform the conversion to Unicode accordingly, and (the new bit) collapse Java-unicode-escape-sequences to single Unicode characters before passing them to the XXXTokenManager.

FileReader is in fact a convenience class that extends the more general InputStream-Reader class. You can get the same FileReader effect by doing the following:

```
XXX parser = new XXX(new XXXTokenManager
                       (new JavaCharStream   // or SimpleCharStream
                           (new InputStreamReader
                               (new FileInputStream("inputfile")))) ;
```

or equivalently

```
XXX parser = new XXX(new XXXTokenManager
                       (new JavaCharStream  // or SimpleCharStream
                           (new InputStreamReader
                               (new FileInputStream
                                   (new File("inputfile")))))) ;
```

In these examples, the FileInputStream object supplies a stream of raw bytes from the indicated file. The InputStreamReader is Unicode-savvy (all Java Reader and Writer objects are Unicode-savvy) and converts that stream of raw bytes from the operating system's default character encoding into a stream of Unicode chars. The JavaCharStream (or SimpleCharStream) wraps the InputStreamReader, providing the bridge to the XXXTokenManager; this bridge includes buffering. In the case of JavaCharStream, this bridge also intercepts each 6-char sequence of the form \uHHHH, where H is a hexadecimal digit, and collapses that sequence to a single Unicode char before outputting it to the XXXTokenManager object. The XXXTokenManager calls the JavaCharStream or SimpleCharStream object whenever it needs the next Unicode character.

Because reading from an InputStream in the default encoding is such a common requirement, JavaCC also provides a much abbreviated form of the constructor that just takes an InputStream as its argument

```
XXX parser = new XXX(new FileInputStream("inputfile")) ;
```

which assumes that the file is in the default encoding (and is to be converted into Unicode accordingly), and assumes the use of XXXTokenManager and either JavaCharStream or SimpleCharStream (depending on the setting of the JAVA_UNICODE_ESCAPE option).

Because System.in is a static InputStream, you can also instantiate a JavaCC parser to read input entered in a terminal:

```
XXX parser = new XXX(System.in) ;
```

6 A Few Warnings

6.1 TokenManager.java, XXXTokenManager.java, CharStream.java, SimpleCharStream.java and JavaCharStream.java

In the general Java world, implementations of an interface named Foo, or extensions of an abstract (incompletely defined) class named Foo, are conventionally named *Something*Foo. For example, Reader is an abstract Java class, extended by the concrete classes InputStreamReader, BufferedReader, StringReader, FileReader, etc.

However, in the JavaCC world, be aware that the automatically generated XXXTokenManager.java file (where XXX is the name of your language) does *not* implement the TokenManager.java interface, and it's not supposed to. Note that the TokenManager.java *interface* will be generated only if you specify the option `USER_TOKEN_MANAGER = true`, which is best left for experts; by default, `USER_TOKEN_MANAGER = false`, which is what most users want.

Similarly, the automatically generated SimpleCharStream.java and JavaCharStream.java files do *not* implement the interface CharStream.java, and they're not supposed to.[9] Note that the CharStream.java interface will be generated only if you specify the option `USER_CHAR_STREAM = true`, which is best left for experts; by default, `USER_CHAR_STREAM = false`, which is what most users want.

So when the JavaCC documentation (../doc/apiroutines.html) lists constructors with signatures like:

```
TheParser.TheParser(CharStream stream)
TheParserTokenManager.TheParserTokenManager(CharStream stream)
```

these constructors cannot be instantiated with an object of type SimpleCharStream or JavaCharStream.[10]

6.2 Constructors Using SimpleCharStream or JavaCharStream

Since JavaCC 2.1, JavaCC has offered two poorly documented constructors with the following signatures:

```
TheParserTokenManager.TheParserTokenManager(SimpleCharStream scs)
TheParserTokenManager.TheParserTokenManager(JavaCharStream jcs)
```

i.e. you can instantiate an XXXTokenManager for your XXX language using either a SimpleCharStream object or a JavaCharStream object. You can also instantiate a JavaCC parser on an XXXTokenManager. This makes it possible to instantiate a

[9] Just to confuse you, the automatically generated SimpleCharStream.java and JavaCharStream.java files contain misleading comments at the top saying that they *are* implementations of the CharStream.java interface. This is misinformation. Just ignore these comments.

[10] Rather these constructors are intended for use by experts who set the option USER_CHAR_STREAM = true, causing the interface CharStream.java to be generated, and then write their own class, e.g. BobsCharStream.java or CarolsCharStream.java, that implements the CharStream.java interface. Most users will prefer to use the automatically generated SimpleCharStream or JavaCharStream classes, which do *not* implement the CharStream.java interface.

JavaCC parser as shown in the (verbose) examples in this paper, e.g.

```
XXX parser = new XXX(new XXXTokenManager
                        (new JavaCharStream   // or SimpleCharStream
                            (new InputStreamReader
                                (  new FileInputStream("inputfile"),
                                   "UTF-8"
                                )
                            )
                        )
                    ) ;
```

where the automatically generated token manager, here called XXXTokenManager, is instantiated on (i.e. "wraps") a SimpleCharStream or JavaCharStream object.

7 Syntactic Variations

7.1 Anonymous vs. Named Objects

The examples so far have consistently used a compact Java syntax that instantiates a number of objects and uses them anonymously; e.g. in the following example the XXXTokenManager, SimpleCharStream, InputStreamReader, and FileInputStream objects are all used without giving them handles (names).

```
XXX parser = new XXX(new XXXTokenManager
                        (new JavaCharStream   // or SimpleCharStream
                           (new InputStreamReader
                              ( new FileInputStream("inputfile"),
                                "UTF-8"
                              )
                           )
                        )
                    ) ;
```

Many insignificant syntactic variations are possible. e.g. the following code is functionally equivalent to the above.

```
FileInputStream fis = new FileInputStream("inputfile") ;
InputStreamReader isr = new InputStreamReader(fis, "UTF-8") ;
JavaCharStream jcs = new JavaCharStream(isr) ;
XXXTokenManager tm = new XXXTokenManager(jcs) ;
XXX parser = new XXX(tm) ;
```

(In real life, you would be forced to put some of the statements in `try` blocks and catch or throw Exceptions.)

7.2 Parameterizing Input File Names and Encodings

Java beginners should also be aware that the examples above appear to "wire-in" the input file name and the name of the encoding, but they can also be represented as String names, e.g.

```
String inputFileName = "inputfile" ;
   /* This inputFileName variable could be set in any
    * convenient way, e.g. from a command-line flag
    * or via user interaction
    */

String inputFileEncoding = "UTF-8" ;
   /* this inputFileEncoding variable could be set in any
    * convenient way, e.g. from a command-line flag,
    * from the extension of the input file, via user
    * interaction, etc.
    */

// Then these variables can be used as shown here

XXX parser = new XXX(new XXXTokenManager
                       (new SimpleCharStream  // or JavaCharStream
                          (new InputStreamReader
                             (  new FileInputStream(inputFileName),
                                inputFileEncoding
                             )
                          )
                       )
                    ) ;
```

or, in JavaCC 4.0

```
XXX parser = new XXX(new FileInputStream(inputFileName), inputFileEncoding) ;
```

8 Suggestions and Corrections are Welcome

I wrote this document after being thoroughly confused by the outdated and inadequate JavaCC documentation, and by the inaccurate comments in the automatically generated JavaCharStream.java and SimpleCharStream.java files. I hope it will be of use to other JavaCC users.

Corrections and suggestions for better presentation would be much appreciated.