JavaCC [tm]: Error Reporting and Recovery

This document describes the error recovery features introduced in Version 0.7.1. This document also describes how features have changed since Version 0.6.

The first change (from 0.6) is that we have two new exceptions:

- ParseException
- TokenMgrError

Whenever the token manager detects a problem, it throws the exception TokenMgrError. Previously, it used to print the message:

Lexical Error ...

following which it use to throw the exception ParseError.

Whenever the parser detects a problem, it throws the exception ParseException. Previously, it used to print the message:

Encountered ... Was expecting one of ...

following which it use to throw the exception ParseError.

In Version 0.7.1, error messages are never printed explicitly, rather this information is stored inside the exception objects that are thrown. Please see the classes ParseException.java and TokenMgrError.java (that get generated by JavaCC [tm] during parser generation) for more details.

If the thrown exceptions are never caught, then a standard action is taken by the virtual machine which normally includes printing the stack trace and also the result of the "toString" method in the exception. So if you do not catch the JavaCC exceptions, a message quite similar to the ones in Version 0.6.

But if you catch the exception, you must print the message yourself.

Exceptions in the Java [tm] programming language are all subclasses of type Throwable. Furthermore, exceptions are divided into two broad categories - ERRORS and other exceptions.

Errors are exceptions that one is not expected to recover from - examples of these are ThreadDeath or OutOfMemoryError. Errors are indicated by subclassing the exception "Error". Exceptions subclassed from Error need not be specified in the "throws" clause of method declarations.

Exceptions other than errors are typically defined by subclassing the exception "Exception". These exceptions are typically handled by the user program and must be declared in throws clauses of method declarations (if it is possible for the method to throw that exception).

The exception TokenMgrError is a subclass of Error, while the exception ParseException is a subclass of Exception. The reasoning here is that the token manager is never expected to throw an exception - you must be careful in defining your token specifications such that you cover all cases. Hence the suffix "Error" in TokenMgrError. You do not have to worry about this exception - if you have designed your tokens well, it should never get thrown. Whereas it is typical to attempt recovery from Parser errors - hence the name "ParseException". (Although if you still want to recover from token manager errors, you can do it - it's just that you are not forced to catch them.)

In Version 0.7.1, we have added a syntax to specify additional exceptions that may be thrown by methods corresponding to non-terminals. This syntax is identical to the Java "throws ..." syntax. Here's an example of how you use this:

```java
void VariableDeclaration() throws SymbolTableException, IOException :
{
    ...
}
```

Here, VariableDeclaration is defined to throw exceptions SymbolTableException and IOException in addition to ParseException.
Error Reporting

The scheme for error reporting is simpler in Version 0.7.1 (as compared to Version 0.6) - simply modify the file ParseException.java to do what you want it to do. Typically, you would modify the getMessage method to do your own customized error reporting. All information regarding these methods can be obtained from the comments in the generated files ParseException.java and TokenMgrError.java. It will also help to understand the functionality of the class Throwable (read a Java book for this).

There is a method in the generated parser called "generateParseException". You can call this method anytime you wish to generate an object of type ParseException. This object will contain all the choices that the parser has attempted since the last successfully consumed token.

Error Recovery

JavaCC offers two kinds of error recovery - shallow recovery and deep recovery. Shallow recovery recovers if none of the current choices have succeeded in being selected, while deep recovery is when a choice is selected, but then an error happens sometime during the parsing of this choice.

Shallow Error Recovery

We shall explain shallow error recovery using the following example:

```java
void Stm() :
{
  IfStm()
  |  WhileStm()
  |  error_skipto(SEMICOLON)
}
```

Let's assume that IfStm starts with the reserved word "if" and WhileStm starts with the reserved word "while". Suppose you want to recover by skipping all the way to the next semicolon when neither IfStm nor WhileStm can be matched by the next input token (assuming a lookahead of 1). That is the next token is neither "if" nor "while".

What you do is write the following:

```java
void Stm() :
{
  IfStm()
  |  WhileStm()
  |  error_skipto(SEMICOLON)
}
```

But you have to define "error_skipto" first. So far as JavaCC is concerned, "error_skipto" is just like any other non-terminal. The following is one way to define "error_skipto" (here we use the standard JAVACODE production):

```java
JAVACODE
void error_skipto(int kind) {
  ParseException e = generateParseException();  // generate the exception object.
  System.out.println(e.toString());  // print the error message
  Token t;
  do {
    t = getNextToken();
  } while (t.kind != kind);
  // The above loop consumes tokens all the way up to a token of
  // "kind". We use a do-while loop rather than a while because the
  // current token is the one immediately before the erroneous token
  // (in our case the token immediately before what should have been
  // "if"/"while".
```
That's it for shallow error recovery. In a future version of JavaCC we will have support for modular composition of grammars. When this happens, one can place all these error recovery routines into a separate module that can be "imported" into the main grammar module. We intend to supply a library of useful routines (for error recovery and otherwise) when we implement this capability.

**Deep Error Recovery**

Let's use the same example that we did for shallow recovery:

```java
void Stm() :
{}
{
  IfStm()
  | WhileStm()
}
```

In this case we wish to recover in the same way. However, we wish to recover even when there is an error deeper into the parse. For example, suppose the next token was "while" - therefore the choice "WhileStm" was taken. But suppose that during the parse of WhileStm some error is encountered - say one has "while (foo { stm; }" - i.e., the closing parentheses has been missed. Shallow recovery will not work for this situation. You need deep recovery to achieve this. For this, we offer a new syntactic entity in JavaCC - the try-catch-finally block.

First, let us rewrite the above example for deep error recovery and then explain the try-catch-finally block in more detail:

```java
void Stm() :
{}
{
  try {
    (IfStm()
    | WhileStm())
  catch (ParseException e) {
    error_skipto(SEMICOLON);
  }
}
```

That's all you need to do. If there is any unrecovered error during the parse of IfStm or WhileStm, then the catch block takes over. You can have any number of catch blocks and also optionally a finally block (just as with Java errors). What goes into the catch blocks is "Java code", not JavaCC expansions. For example, the above example could have been rewritten as:

```java
void Stm() :
{}
{
  try {
    (IfStm() | WhileStm())
  catch (ParseException e) {
    System.out.println(e.toString());
    Token t;
    do {
      t = getNextToken();
    } while (t.kind != SEMICOLON);
  }
```
Our belief is that it's best to avoid placing too much Java code in the catch and finally blocks since it overwhelms the grammar reader. It's best to define methods that you can then call from the catch blocks.

Note that in the second writing of the example, we essentially copied the code out of the implementation of error_skipto. But we left out the first statement - the call to generateParseException. That's because in this case, the catch block already provides us with the exception. But even if you did call this method, you will get back an identical object.