Getting Started with JavaCC

2006-01-31

Jonas Lundberg
Software Technology Group
Institute of Mathematics and System Engineering,
Växjö University

(email: Jonas.Lundberg@msi.vxu.se)

Abstract

This short text is targeted to students that wants to start using JavaCC. The aim is to bridge the gap from "never used JavaCC" to being able to read the instructions provided at `http://javacc.dev.java.net`. It also contains a simple exercise suitable for making you more comfortable with `JavaCC`.

Our start kit is available at

`http://w3.msi.vxu.se/users/jonasl/javacc`

Introduction

JavaCC (Java Compiler Compiler) is a parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.

Getting Started

First of all, make sure that you have Java installed. We have tested JavaCC with Java v1.4.2 but any reasonable new version will probably do. Remember to update your enviromental variables according to:

```
CLASSPATH:  "other paths"; .../jdk1_4_2/jre/lib/rt.jar
PATH: "other paths"; ... /jdk1_4_2/bin
```

The exact syntax depends on what operating system you are using. Verify that they work by typing `java`, and `javac` in a terminal window.

Next, install JavaCC. It can downloaded from

`http://w3.msi.vxu.se/users/jonasl/javacc`

or from the official JavaCC home page at

`http://javacc.dev.java.net`

You must once again, after unzipping, update your `PATH` enviromental variable.

```
PATH: "other paths"; ... javacc-4.0/bin
```

Once that is done, you can verify that it works by typing:

```
prompt% javacc
```

This should result in a short text describing how to use the `javacc` command. You can also try the two programs `jjtree` and `jjdoc` that we will use later on.

Getting Started using JavaCC

We have put together a tiny getting started kit that you can download from

`http://w3.msi.vxu.se/users/jonasl/javacc`

The kit contains a zipped directory named `simple_parser` with the following content:

```
Main.java   DFSPrinter.java   SimpleParser.jjt   testing/
```

(plus a number of other Java files). The most important files are:

1. `SimpleParser.jjt`: This is a simple parser specification. This file can be used to *generate* a set of Java source files that implements a parser for a very simple programming language.

2. `Main.java` A simple driver class (containing the main method) that initiates and starts the generated parser.

3. `DFSPrinter.java` A class that prints an indented version of the resulting syntax tree.

The directory `testing` contains a few very simple test programs that can be used for testing purposes. For example, the file `testing/basics.prgm` looks like:

```
program B {
  //Declarations
  int a;
  int b;
  int c;
  int d;

  //Statements
  a = 1;
  b = a+1;
  c = a+3*b;
  d = (a+b)/(a-b);
  print(a/b+c*d);
}
```

The Example Application

The getting starting kit includes the complete code of a parser for a simple programing language. It implements a parser for the following grammar:

```
Program             ::= <PROGRAM> Id <LB> Body <RB>
Body                ::= Declaration* Statement*
Declaration         ::= Type Id <SC>
Statement           ::= AssignStmt | PrintStmt
AssignStmt          ::= id <ASGN> AdditativeExp <SC>
PrintStmt           ::= <PRINT> <LP> AdditativeExp <RP> <SC>
AdditativeExp       ::= MultiplicativeExp ( AddOp MultiplicativeExp )*
MultiplicativeExp   ::= UnaryExp ( MultOp UnaryExp )*
UnaryExp            ::= Id | IntLiteral | ParentizedExp | NegativExp
ParentizedExp       ::= <LP> AdditativeExp <RP>
NegativExp          ::= <MINUS> UnaryExp
MultOp              ::= <MULT> | <DIV>
AddOp               ::= <PLUS> | <MINUS>
Type                ::= <INT>
Id                  ::= <IDENTIFIER>
IntLiteral          ::= <INTEGER_LITERAL>
```

Here `Program` is the start symbol and everything enclosed in `<>`-brackets are terminals. The JavaCC specification for this grammar can be found in the file `SimpleParser.jjt`. The specification can be used to generate a parser implementation (`Parser.java`) together with a number of classes used by that class.

The application entry point is the file `Main.java`. It starts by reading the program to be parsed. The connection to the parser part of the program looks something like:

```java
// Create parser and parse expression
Parser parser = new Parser(inputStream);
ASTProgram root = parser.Program();
System.out.println("Program parsed successfully.");

// Print an indented list of the AST nodes
DFSPrinter printer = new DFSPrinter();
printer.print(root);
```

That is, we create a parser instance by adding the current input stream to the parser constructor (`new Parser(inputStream)`). In the line

```java
ASTProgram root = parser.Program();
```

we do the actual parsing by invoking a method corresponding to the start symbol (`Program`) of the grammar. The return value of the parsing command is a reference to the root of the generated syntax tree. That is, an instance of the class `ASTProgram.java` which, like all node types, inherits from the class `SimpleNode` which implements the behaviour that is common to all nodes in the syntax tree. This step concludes the actual parsing.

In the final part we print an indented list of the syntax tree nodes. The printing is done by a class named `DFSPrinter` which method `print()` traverses the tree in depth-first order and prints the name of each visisted node.

Build and Run

The building process is done in three steps:

```
prompt> jjtree SimpleParser.jjt
prompt> javacc SimpleParser.jj
prompt> javac Main.java
```

1. The command `jjtree` reads the file `SimpleParser.jjt` and a) generates the Java classes needed to construct a parse tree. Each non-terminal node type is represented by a seperate class, b) it generates a JavaCC (`.jj`) file named `SimpleParser.jj` that will be used in the next step to generate the parser.

2. The command `javacc` reads the file `SimpleParser.jj` and generates the Java classes needed to construct the parser.

3. The final command `javac` compiles all the Java source files to an executable program.

You can now run your parser as an ordinary Java program

```
prompt> java Main testing\basics.prgm
```

If everything works, you should see a textual representation of the generated parse tree.

JavaCC - an Introduction

This is just a brief introduction to parser specification using JavaCC. Further details can be found at the JavaCC home page at `javacc.dev.java.net`. The actual specification takes place in a JJTree file with postfix `.jjt`. A JJTree file can roughly be divided into three parts:

1. A **class template** where you decide the properties (e.g. name, visibility, etc.) of the Java parser class to be generated. In our example it looks like:

```
PARSER_BEGIN(Parser)
import java.io.*;

public class Parser {}
PARSER_END(Parser)
```

2. A section containing the **lexical specification** (the tokens to be recognized). Here you specify the token names and their corresponding regular expressions. A piece of our example specification looks like:

```
/* LITERALS */
TOKEN :
{
  < INTEGER_LITERAL : "0" | ["1"-"9"] (["0"-"9"])* >
}

/* OPERATORS */
TOKEN :
{
      < PLUS: "+" >
  |   < MINUS: "-" >
  |   < MULT: "*" >
  |   < DIV: "/" >
}
```

   This means that integers have the token `INTEGER_LITERAL` with corresponding regular expression $0 \mid [1-9][0-9]*$ and that the addition operator has the token `PLUS` with corresponding regular expression $+$. You can use the extended set of operators *concat*, $|$, $*$, $+$, $?$, $[\,]$ to define your regular expressions.

3. The final section contains the **context-free grammar** defining the syntax analysis. JavaCC uses the Extended Backus-Nauer Form (EBNF) where the standard notation (BNF) is extended with $*$, $+$, $?$. A piece of our example specification looks like:

```
void AdditativeExp() #AddExp:
{}
{
   ( MultiplicativeExp() (AddOp() MultiplicativeExp())*)
}

void MultiplicativeExp() #MultExp:
```

```
{}
{
   (UnaryExp() (MultOp() UnaryExp())*)

}
```

This corresponds to the following two productions:

$$\begin{aligned} AdditativeExp &\rightarrow MultiplicativeExp\ (AddOp\ MultiplicativeExp)* \\ MultiplicativeExp &\rightarrow UnaryExp\ (MultOp\ UnaryExp)* \end{aligned}$$

The parts `#AddExp` and `#MultExp` informs JJTree to name the corresponding AST node classes to `ASTArithExp.java` and `ASTMultExp.java`.

Additional JavaCC Features

Generating HTML-grammars using `jjdoc`

JavaCC comes with a nice tool (`jjdoc`) that reads a JavaCC file and generates a readable grammar in HTML format. This is done by executing the command.

```
prompt> jjdoc SimpleParser.jj
```

The resulting file `SimpleParser.html` shows a nice printout of the grammar specified in `SimpleParser.jjt`.

Collapsing Nodes in the AST

Once you have a correct parser you can reduce the size of the syntax tree by collapsing some nodes that contains no valuable information. This can be done by some minor modifications of the grammar specification in the JJTree file. For example, if you compare

```
void MultiplicativeExp() #MultExp(>1):
{}
{
   (UnaryExp() (MultOp() UnaryExp())*)

}
```

with the version displayed above you can see that we have replaced `#MultExp` in the header with a `#MultExp(>1)`. This means that a `MultExp` node is only added to the syntax tree if it has more than one child. By using this approach cleverly (but with some caution) we can avoid the frequently occuring long non-branching subtrees in the syntax tree. Furthermore, by removing `#MultExp` all together you instruct JJTree to *never* add any such nodes. More information about how to change the shape of the generated tree can be found in the JJTree tutorial at the JavaCC home page.

Annotating the AST

If you look in the example code at the end of the grammar specification you will find the following piece of code:

```
void AddOp() #AddOp:
{Token t;}
{
    (t=<PLUS> | t=<MINUS>)  {jjtThis.setLexem(t.image);}
}

void Integer_Literal() #Int :
{Token t;}
{
   t=<INTEGER_LITERAL> {jjtThis.setLexem(t.image);}
}
```

In order to understand this we must first realize that JavaCC creates a new AST node (if not instructed otherwise - see the previous section) during the parsing process every time it uses a production. The variable `jjtThis` holds a reference to this newly created node object. In the above case we use this reference to annotate the node with some information. The methods used are defined in the different AST node classes. For example:

```java
/* Generated By:JJTree: Do not edit this line. ASTAddOp.java */

public class ASTAddOp extends SimpleNode {
  public ASTAddOp(int id) {
    super(id);
  }

  public ASTAddOp(Parser p, int id) {
    super(p, id);
  }

  // My added members
  String lexem = "";

  public void setLexem(String lex) {lexem = lex;}
  public String getLexem() {return lexem;}

  //Override super class handling
  public String toString() {return super.toString()+": "+lexem;}
}
```

The above code consists of two parts: The upper part generated by the `jjtree` command and a hand crafted section defining a few members. That is, we have added a few lines of code to the node class `ASTAddOP` that was generated by JJTree. This is the way we usually export "extra" information from the actual parsing to the resulting syntax tree. This information can later be accessed when traversing the tree. Our `DFSPrinter` uses the overwritten method `toString()` when printing the syntax tree.

Lookahead handling

The default lookahead size for a JavaCC generated parser is 1. However, in the grammar specification you can instruct the parser to temporarily increase this value. This approach is often useful when left-recursion appears (and you don't want to rewrite the grammar). For example, in the following production we instruct the parser to use lookahead 2 when trying to resolve which branch of `Type` to chose.

```
void Type() #Type:
{}
{
LOOKAHEAD(2)
  "int" "[" "]"
| "int"
}
```

This approach should be used with some caution since using *lookahead* > 1 will slow down the compiler considerably. More information about how to use `LOOKAHEAD` can be found in *lookahead tutorial* at the JavaCC home page.

Sources of Information

JavaCC has a home page (http://javacc.dev.java.net) where everything you need can be found. There you can:

- Download JavaCC
- Read tutorials. Example, tutorials for
  - `jjtree`,
  - `jjdoc`,
  - `lookahead` handling.
- Find examples
- Look at grammars for the complete Java and many other languages

Start Kit

We have put together a very simpe start kit involving this text, the example `SimpleParser`, and a complete version of JavaCC (version 4.0). It can be found at

`w3.msi.vxu.se/users/jonasl/javacc`

Exercises

In this execise we should extend the parser `SimpleParser` to be able to parse programs like

```
program Compute {
  //Declarations
  int n = 100;
  int count = 2;
  int fib;
  int prev = 1;
  int prevPrev = 1;

  //Compute fibonacci(n)
  if (n > 2) {
    while (count < n) {
      fib = prev + prevPrev;
      prevPrev = prev;
      prev = fib;
      count = count + 1;
```

```
    }
  } else {
    fib = 1;
  }

  if (fib > 1) {
    print(fib);
  }
}
```

That is, we have kept the structure (declarations followed by statements) but added a number of new language constructs. We suggest the following approach:

1. **Identifiers and initialization**: a) Change the lexical specification for identifiers to accept all strings starting by a letter followed by zero or more letters or digits. Change the grammer so that we can assign a value to the variables when we are declaring them. The following type of declarations should be accepted:

   ```
   int first = 1;
   int second = first + 2;
   int third;
   ```

   Use the test program `testing\assignments.prgm` to check if it works.

2. **Boolean variables and expressions**: Introduce a new type `boolean`, the boolean litterals `true` and `false`, and the operations `<`, `>` and `&&` (logical `AND`). Make sure that correct operator priorities are explicit in the resulting syntax tree. The following type of program should be accepted:

   ```
   //Declarations
   int a = 1;
   boolean v1;
   boolean v2 = true;

   //Statements
   v1 = 8 < a + 1;           //Rhs Priority: 8 < (a+1)
   v2 = 4 < a && 5 < a;      //Rhs Priority: (4<a) && (5<a)
   print(true && false);
   ```

   Use the test program `testing\boolean.prgm` to check if it works.

3. **While and If statements**: Introduce `while-` and `if`-statements that looks like:

   ```
   while ( ... ) {  |   if ( ... ) {    |    if ( ... ) {
     ...            |     ...           |      ...
   }                |   }               |    } else {
                    |                   |      ...
                    |                   |    }
   ```

   That is, the `else` part in the `if`-statements should be optional. It should be possible to have nested statements (statements within statements). Use the test program `testing\statements.prgm` to check if it works.

4. **Semantic Analysis**: The parser checks if the input is syntacticly correct and constructs a syntax tree. The next phase of a compiler is called the semantical analysis. This is where we find programming errors not caught by the parser. For example, the following code has correct syntax but is still wrong since we are using a variable `b` that we havn't declared.

   ```
   int a = 1;
   b = a+3;
   ```

   Write a class `CheckDeclared.java` that traverses the syntax tree and checks that all variables are declared before they are used. Hint: Take a look in `DFSPrinter.java` for how to traverse the tree.

5. **What remains?**: What more should be checked before we can say that the program is correct?