JavaCC Tutorial

Compilers

University of Porto/FEUP

Department of Informatics Engineering

ver. 1.2, February 2020

Content

1. Implementation of Syntactic Analyzers using JavaCC (Part 1) 2
   Example 2
2. Implementation of Syntactic Analyzers using JavaCC (Part 2) 4
   Example 4
   Tree Annotations 6
   Expression Evaluation using the Annotated Tree 10
   Simplification of the Generated Tree and Use of the Node Type to Represent the Operators 10
3. Proposed Exercises 13
   Exercise 1 13
   Exercise 2 13
   Exercise 3 13
   Exercise 4 14
4. References 14
   Other Tools to Generate Parsers: 14

Goal: To be able to build syntactic analyzers (parsers) using the JavaCC tool.

Strategy: Build an arithmetic expressions interpreter taking into account several options.

---

1. Implementation of Syntactic Analyzers using JavaCC (Part 1)

The goal of this document is to initiate learning of the JavaCC syntactic analyzers generator [1][2]. Knowledge about predictive top-down syntax analysis will contribute to improve your perception on how parser generators such as JavaCC work.

The JavaCC tool requires the description of the **tokens** and **grammar rules** in a file with the **.jj** extension. The tool generates a set of Java classes[1], including one with the same name as the parser. Figure 1 shows a generic view of the JavaCC flow, in which a **.jj** file describing the parser is given as input and the tool generates the supporting classes.

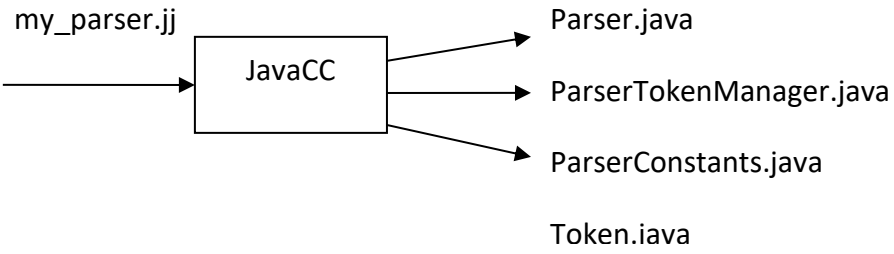

Figure 1. Inputs and Outputs of JavaCC, assuming that the parser was named "Parser" inside the *my_parser.jj* file.

The "**.jj"** files are composed by:

1. Options settings (optional): where the global settings are defined, including the global **lookahead** level.
2. Java Compilation Unit (PARSER_BEGIN(Parser) ... PARSER_END(Parser)) which defines the parser main class and where the main method may be included.
3. Grammar rules (accepts the symbols +, *, and ? with the same meaning as in regular expressions)

Example

Suppose that we intend to implement a syntactic analyzer that recognizes arithmetic expressions composed of a single positive integer or an addition or subtraction of two positive integers (e.g., 2+3, 1-4, 5). Terminal symbols (tokens) are specified using regular expressions as follows:

```
INTEGER = [0-9]+       // terminal symbol
Aritm → INTEGER [("+" | "-") INTEGER]  // grammar rule
```

[1] While you get acquainted with JavaCC you will also learn the meaning of these supporting classes.

To develop a parser implementing this grammar using JavaCC we have to create a new file describing the parser. In this example we name it as **Example.jj**.

Example.jj file:

```
PARSER_BEGIN(Example)

// Java code invoking the parser
    public class Example {
        public static void main(String args[]) throws ParseException {
            // Object instantiation using the constructor with an argument
            // that reads from the standard input (keyboard)
            Example parser = new Example(System.in);
            parser.Aritm();
        }
    }
PARSER_END(Example)

// Symbols that must be skipped during the lexical analysis
SKIP :
{
    " " | "\t" | "\r"
}

// token definition (terminal symbols)
TOKEN :
{
    < INTEGER : (["0" – "9"])+ >
    | < LF : "\n" >
}

// Production definition
void Aritm() : {}
{
    <INTEGER> ( ("+" | "-") <INTEGER> )? <LF>       // "(...)?" equivalent to "[...]"
}
```

To generate the *parser*:

javacc Example.jj

To compile the generated Java code:

javac *.java

To execute the syntactic analyzer:

java Example

We can add arbitrary Java code sections in the production definition. For example, the following modifications allow printing the numbers read by the parser to the console:

```
void Aritm() : {Token t1, t2;}
{
    t1=<INTEGER> {
        System.out.println("Integer = "+t1.image);
    }
    ( ("+" | "-")  t2=<INTEGER> {
        System.out.println("Integer = "+t2.image);
    }
    )? (<LF>)
}
```

For each INTEGER symbol, a Java code line to print the token value read was inserted (the **image** field of the **Token** class returns a string representing the token value[2]).

Insert these modifications in the **Example.jj** file and repeat the process until the parser is executed. See what happens.

How could we print in the console the operator (+ or -)?

2. Implementation of Syntactic Analyzers using JavaCC (Part 2)

After a first approach to JavaCC, we are now going to see how we can automatically generate a syntax tree. For that, we are now going to use first JJTree [5] instead of JavaCC. JJTree is a pre-processing tool integrated in the JavaCC parser generator tool. JJTree generates Java classes and a JavaCC file that, on top of the parser description, integrates Java code for the syntax tree generation. The JJTree input file, which has the **.jjt** extension, specifies the grammar in the same way it is specified in JavaCC and additionally includes directives for the generation of tree nodes. Figure 2 shows the inputs and outputs of JJTree.

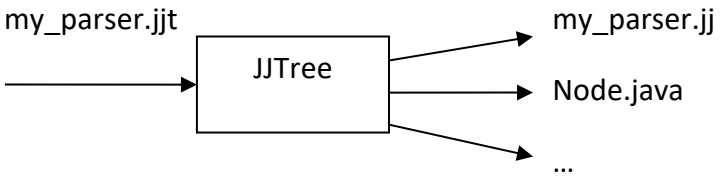

Figure 2. Inputs and outputs of JJTree.

Example

To test JJTree consider the challenge of building a calculator for simple arithmetic expressions with integer numbers, using the syntax tree generated by the parser to perform the calculations.

Consider the following tokens for the grammar:

```
LF="\n"
INTEGER=[0-9]+
```

and the following grammar for the arithmetic expressions:

```
Expression → Expr1 LF
Expr1 → Expr2 [("+" | "-") Expr2]
Expr2 → Expr3 [("*" | "/") Expr3]
Expr3 → INTEGER
```

[2] Other methods will be presented later.

Expr3 → "-" Expr3

Expr3 → "(" Expr1 ")"

To generate the syntax tree, we specify the original grammar in a file with the **jjt** extension and we add the directives that instruct JJTree to generate the code needed to build the tree.

The file presented next contains the previous grammatical rules and includes modifications introduced to generate the tree, identified in bold. The *dump(String)* method was also modified to print the generated tree for each input arithmetic expression.

Calculator.jjt file:

```
options
{
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException {
        System.out.println("Parser for a calculator that accepts expressions with integers, +,-,*,/,(,
and ).");
        System.out.println("Write an arithmetic expression:");
        Calculator myCalc = new Calculator(System.in);
        SimpleNode root = myCalc.Expression(); // returns reference to root node

        root.dump(""); // prints the tree on the screen
   }
}

PARSER_END(Calculator)

SKIP :
{
   " " | "\r" | "\t"
}

TOKEN:
{
    < INTEGER: (["0"-"9"])+ >
   |   < LF: "\n" >
}

SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;} // Java code inside brackets
}

void Expr1(): {}
{
  Expr2() [("+" | "-") Expr2()]
}

void Expr2(): {}
{
  Expr3() [("*" | "/") Expr3()]
}
```

```
void Expr3(): {}
{
   <INTEGER>
   | "-" Expr3()
   | "(" Expr1() ")"
}
```

To execute JJTree:

jjtree Calculator.jjt

To generate Java code with JavaCC:

javacc Calculator.jj

To compile the generated Java code:

javac *.java

To execute the syntax analyzer:

java Calculator

Now let's check the trees generated for the expressions you introduced.

Building the syntax tree is not enough to develop the arithmetic expression calculator. First, the generated tree does not store the values of the processed integers. Second, it is necessary to program the procedure for traversing the tree and calculating the value for the arithmetic expression represented by the tree. We are going to see next the required steps to include annotations in the tree.

Tree Annotations

To solve the first problem, we have to do some modifications. JJTree only generates the *SimpleNode*[3] class if it does not exist. This class is used to represent the syntax tree nodes[4] and class includes the following methods:

Some methods of the class that represents the syntatic tree nodes:	Description:
public Node jjtGetParent()	Returns the parent node of the current node
public Node jjtGetChild(int i)	Returns child node no. *I* of current node
public int jjtGetNumChildren()	Returns number of children of current node
Public void dump()	Prints on screen the syntax tree starting at the node that invoked the method

New methods and/or fields can be added to the *SimpleNode* class after it has been previously generated by JJTree. At the moment, we are interested in adding two fields that allow storing the value of the integer (in the leaf nodes) [5] and the operations to perform[6].

[3] Name of the class defined as instance to be returned by the *myCalc()* function.
[4] It is also possible to generate one class per node (see MULTI option).
[5] It would be possible to add two methods that allow accessing the field (assign and return its value).
[6] We could use the node ID (see field "id" of the "SimpleNode" class and the node types for each of the grammars in the class "CalculatorTreeConstants").

For this example, we define a class with the required operations as follows:

MyConstants.java file:

```
public class MyConstants {

    public static final int ADD =1;
    public static final int SUB =2;
    public static final int MUL =3;
    public static final int DIV =4;

    public static final char[] ops = {'+','-','*','/'};
}
```

The *SimpleNode* class is presented next with the required modifications (in bold).

Notice that instead of annotating the nodes with the operations to perform, we could have instructed JJTree to associate tree nodes to the respective arithmetic operation (using directives such as "#Add"). We will see later on how to use these directives.

SimpleNode.java file:

```
/* Generated By:JJTree: Do not edit this line. SimpleNode.java */
public class SimpleNode implements Node {

  protected Node parent;
  protected Node[] children;
  protected int id;
  protected Object value;
  protected Calculator parser;

  //  added
  public int val;
  public int Op=0;

  public SimpleNode(int i) {
    id = i;
  }

  public SimpleNode(Calculator p, int i) {
    this(i);
    parser = p;
  }

  public void jjtOpen() {}

  public void jjtClose() {}

  public void jjtSetParent(Node n) { parent = n; }
  public Node jjtGetParent() { return parent; }

  public void jjtAddChild(Node n, int i) {
    if (children == null) {
      children = new Node[i + 1];
    } else if (i >= children.length) {
      Node c[] = new Node[i + 1];
      System.arraycopy(children, 0, c, 0, children.length);
      children = c;
```

```
    }
    children[i] = n;
  }

  public Node jjtGetChild(int i) {
    return children[i];
  }

  public int jjtGetNumChildren() {
    return (children == null) ? 0 : children.length;
  }

  public void jjtSetValue(Object value) { this.value = value; }
  public Object jjtGetValue() { return value; }

  /* You can override these two methods in subclasses of SimpleNode to
     customize the way the node appears when the tree is dumped.  If
     your output uses more than one line you should override
     toString(String), otherwise overriding toString() is probably all
     you need to do. */

  public String toString() { return CalculatorTreeConstants.jjtNodeName[id]; }
  public String toString(String prefix) { return prefix + toString(); }

  /* Override this method if you want to customize how the node dumps
     out its children. */

  public void dump(String prefix) {
    System.out.println(toString(prefix));

    if(this.Op != 0)
        System.out.println("\t[ "+MyConstants.ops[this.Op-1]+" ]");
    if(children == null)
        System.out.println("\t[ "+this.val+" ]");

    if (children != null) {
      for (int i = 0; i < children.length; ++i) {
        SimpleNode n = (SimpleNode)children[i];
        if (n != null) {
          n.dump(prefix + " ");
        }
      }
    }
  }
}
```

We will see next the file that allows to obtain the syntactic analyzer that includes the generation of the annotated syntax tree.

To use the syntax tree it is important that we can verify the type of a given node. With that goal in mind, one can annotate each node with the information related with its respective type. In this example, we use the constants defined in the *MyConstants* class.

New Calculator.jjt file:

```
options
{
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)
```

```
public class Calculator
{
   public static void main(String args[]) throws ParseException {
       System.out.println("Parser for a calculator that accepts expressions with integers, +,-,*,/,(,
and ).");
       System.out.println("Write an arithmetic expression:");
       Calculator myCalc = new Calculator(System.in);
       SimpleNode root = myCalc.Expression(); // returns reference to root node

       root.dump(""); // prints tree on screen
   }
}

PARSER_END(Calculator)

SKIP :
{
   " " | "\r" | "\t"
}

TOKEN:
{
    < INTEGER: (["0"-"9"])+ >
   |   < LF: "\n" >
}

SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;}
}

void Expr1(): {}
{
  Expr2(1)
  [
   ("+" {jjtThis.Op = MyConstants.ADD;}
   | "-" {jjtThis.Op = MyConstants.SUB;}
   )
   Expr2(1)
   ]
}

void Expr2(int sign): {}    // 1: positive; -1: negative, because of the '-' unitary operator
{
  Expr3(sign)
  [
   ("*" {jjtThis.Op = MyConstants.MUL;}
   | "/" {jjtThis.Op = MyConstants.DIV;}
   )
   Expr3(1)
  ]
}

void Expr3(int sign):  {Token t;}
{
   t=<INTEGER>
   {
    jjtThis.val = sign *Integer.parseInt(t.image);
   }
   | "-" Expr3(-1)
   | "(" Expr1() ")"
}
```

Expression Evaluation using the Annotated Tree

Relying on the annotation of the integers and the operations in the syntax tree, we can program a method able to determine the value of an arithmetic expression given by the user. We can do this including a recursive function such as follows:

```
int eval(SimpleNode node) {

    if(node.jjtGetNumChildren() == 0) // leaf node with integer value
        return node.val;
    else if(node.jjtGetNumChildren() == 1) // only one child
         return this.eval((SimpleNode) node.jjtGetChild(0));

     SimpleNode lhs = (SimpleNode) node.jjtGetChild(0); //left child
     SimpleNode rhs = (SimpleNode) node.jjtGetChild(1); // right child

     switch(node.Op) {
         case MyConstants.ADD : return eval( lhs ) + eval( rhs );
         case MyConstants.SUB : return eval( lhs ) - eval( rhs );
         case MyConstants.MUL : return eval( lhs ) * eval( rhs );
         case MyConstants.DIV : return eval( lhs ) / eval( rhs );
         default : // abort
             System.out.println("Ilegal operator!");
             System.exit(1);
     }
     return 0;
 }
```

The method above can be placed between:

```
"PARSER_BEGIN(Calculator)" and "PARSER_END(Calculator)".
```

Then, we can invoke it in "*main*", as illustrated by the instruction:

```
System.out.println("Expression value: "+myCalc.eval(root));
```

Simplification of the Generated Tree and Use of the Node Type to Represent the Operators

The syntax trees generated in the presented examples are designated as concrete syntax trees (they represent the derivations by the grammar productions faithfully). The simplification of these trees results in abstract syntax trees (ASTs).

The *#void* directive is used to avoid the generation of a node for each grammar production specified in the **jjt** file. These directives are placed after the names of the procedures which we do not want to represent as a node in the syntax tree. This method allows to automatically generate ASTs without requiring the transformation of the concrete syntax tree into an abstract syntax tree (AST). The following file presents a version of the calculator that generates ASTs. We will use this example to illustrate how we can use the node types that are attributed automatically. The node type identifier is stored in the *id* attribute of each tree node (see *node.id* in the examples). For each tree node type, JJTree generates a file in which the constants identifying the nodes types that can appear in the syntax trees generated from the grammatical rules are specified (see the **CalculatorTreeConstants.java** file). The node type corresponding to the procedures related to the grammatical rules that do not use the "*#void*" directive and/or to the nodes specified in the directives to JJTree. This version illustrates the

association of different identifiers to grammatical rules in the same procedure (see, e.g., #Add(2), #Mul(2)).

New Calculator.jjt file:

```
options {
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)
public class Calculator
{
   public static void main(String args[]) throws ParseException {
        System.out.println("Calculator that accepts expressions with integers, +,-,*,/,(, and ).");
        System.out.println("Write an arithmetic expression:");
       Calculator myCalc = new Calculator(System.in);
       SimpleNode root = myCalc.Expression();
       root.dump("");

       System.out.println("Expression Value: "+myCalc.eval(root));
   }

int eval(SimpleNode node) {

    if(node.jjtGetNumChildren() == 0) // leaf node with integer value
        return node.val;
    else if(node.jjtGetNumChildren() == 1) // only one child
         return this.eval((SimpleNode) node.jjtGetChild(0));

    SimpleNode lhs = (SimpleNode) node.jjtGetChild(0); //left child
    SimpleNode rhs = (SimpleNode) node.jjtGetChild(1); // right child

     switch(node.id) {
         case CalculatorTreeConstants.JJTADD : return eval( lhs ) + eval( rhs );
         case CalculatorTreeConstants.JJTSUB : return eval( lhs ) - eval( rhs );
         case CalculatorTreeConstants.JJTMUL : return eval( lhs ) * eval( rhs );
         case CalculatorTreeConstants.JJTDIV : return eval( lhs ) / eval( rhs );
         default : // abort
             System.out.println("Ilegal operator!");
             System.exit(1);
     }
     return 0;
    }
}
PARSER_END(Calculator)

SKIP :
{
   " " | "\r" | "\t"
}

TOKEN:
{
   < INTEGER: (["0"-"9"])+ >
   |   < LF: "\n" >
}

SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;}
}
```

```
void Expr1() #void: {}
{
  Expr2(1)
  [
   ("+" Expr2(1) #Add(2)
   | "-" Expr2(1) #Sub(2))
  ]
}

void Expr2(int sign) #void: {}    // 1: positive; -1: negative
{
  Expr3(sign)
  ("*" Expr3(1) #Mul(2)
   | "/" Expr3(1) #Div(2)
   )?          // (…)?equivalent to […]
}

void Expr3(int sign) #void:  {Token t;}
{
   t=<INTEGER>
   {
    jjtThis.val = sign * Integer.parseInt(t.image);
   } #Term
   | "-" Expr3(-1)
   | "(" Expr1() ")"
}
```

New SimpleNode.java file:

```
public class SimpleNode implements Node {

    protected Node parent;
    protected Node[] children;
    protected int id;
    protected Object value;
    protected Calculator parser;

    //  added
    public int val;
    …
    public void dump(String prefix) {
        System.out.println(toString(prefix));
        switch(this.id) {
        case CalculatorTreeConstants.JJTADD:
            System.out.println("\t[ + ]");break;
         case CalculatorTreeConstants.JJTSUB:
            System.out.println("\t[ - ]");break;
         case CalculatorTreeConstants.JJTMUL:
            System.out.println("\t[ * ]");break;
         case CalculatorTreeConstants.JJTDIV:
            System.out.println("\t[ / ]");break;
        }
        if(children == null)
            System.out.println("\t[ "+this.val+" ]");
        if (children != null) {
            for (int i = 0; i < children.length; ++i) {
                SimpleNode n = (SimpleNode)children[i];
                if (n != null) {
                    n.dump(prefix + " ");
                }
            }
        }
    }
```

```
}
```

CalculatorTreeConstants.java Generated File:

```
public interface CalculatorTreeConstants {

    public int JJTEXPRESSION = 0;
    public int JJTVOID = 1;
    public int JJTADD = 2;
    public int JJTSUB = 3;
    public int JJTMUL = 4;
    public int JJTDIV = 5;
    public int JJTTERM = 6;

    public String[] jjtNodeName = { "Expression",
"void",   "Add",   "Sub",  "Mul",  "Div",  "Term" };
}
```

Concepts to practice:

1. Definition of tokens in JavaCC (impact of the specification order, match based on the longest possible sequence of characters).
2. *Lookahead*: *lookahead* level, global and local.
3. Syntactic and semantic *Lookahead*.
4. Descendent recursive parsers and predictive. *Backtracking;*
5. Error handling and recovery.

3. Proposed Exercises

Exercise 1

Modify the previous grammar so that it accepts assignments to symbols and the use of symbols in the arithmetic expressions, as illustrated by the following example:

```
A=2;
B=3;
A*B;
```

Consider that the symbols are defined by the regular expression [A-Za-z][0-9A-Za-z]*.

Then, add the code to calculate the arithmetic expressions (the interpretation of the previous code must show the value 6 on the screen).

Exercise 2

The implemented calculator does not calculate directly expressions of the type -(2), -(2+3), etc. The '-' sign immediately before the '(' symbol is not considered in the calculations. What modifications can be done so that expressions such as the previous ones are evaluated correctly?

Exercise 3

The presented grammar does not accept arithmetic expressions that contain sequences of operations of the same type (e.g., 2+3+4 or 2*3*4). The only way to introduce expressions of this type is to use parenthesis to group the operations such as, for example, 2+(3+4) or 2*(3*4). The grammar modified to accept sequences of operations of the same type ("{...}" indicates 0 or more) is illustrated below.

Expression → Expr1 LF

Expr1 → Expr2 **{**("+" | "-") Expr2**} // modified**

Expr2 → Expr3 **{**("*" | "/") Expr3**} // modified**

Expr3 → INTEGER

Expr3 → "-" Expr3

Expr3 → "(" Expr1 ")"

Identify the problems that this grammar will introduce in the calculation of the arithmetic expressions using the syntax tree.

Exercise 4

Taking into account one of the previous implementations (for example, the one from Exercise 2), improve the way that grammatical errors are handled and presented (include error recovery [6]).

4. References

[1] JavaCC, https://javacc.org/

[2] Tom Copeland, *Generating Parsers with JavaCC*, Second Edition, Centennial Books, Alexandria, VA, 2009. ISBN 0-9762214-3-8, http://generatingparserswithjavacc.com/

[3] JavaCC™: Documentation Index, https://javacc.org/doc

[4] Oliver Enseling, "Build your own languages with JavaCC", Copyright © 2003 JavaWorld.com, an IDG company, http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools_p.html

[5] JavaCC™: JJTree Reference Documentation, https://javacc.org/jjtree

[6] JavaCC™: Error Reporting and Recovery, https://javacc.org/tutorials/errorrecovery

Other Tools to Generate Parsers:

[7] SableCC: http://sablecc.org/

[8] ANTLRWorks: The ANTLR GUI Development Environment, http://www.antlr.org/